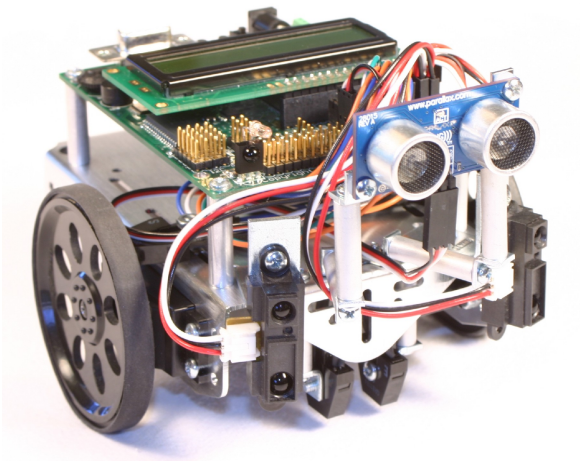
RidgeSoft™

Exploring Robotics with the IntelliBrain-Bot

An Introduction to Robotics and Java™ Programming



RidgeSoft, LLC
PO Box 482
Pleasanton, CA 94566
www.ridgesoft.com

Revision 2

Table of Contents

CHAPTER 1

Introducing the IntelliBrain-Bot

Throughout this book we will be using the IntelliBrain™-Bot Deluxe educational robot to learn about the emerging field of robotics. The IntelliBrain-Bot educational robot is a pre-designed mobile robot, which will allow us to focus our discussion primarily on robotics programming, using the companion Java™-enabled robotics software development environment, RoboJDE™. Before we get started programming the robot, let's first take a look at the mechanical and electronics components which make up the IntelliBrain-Bot educational robot.

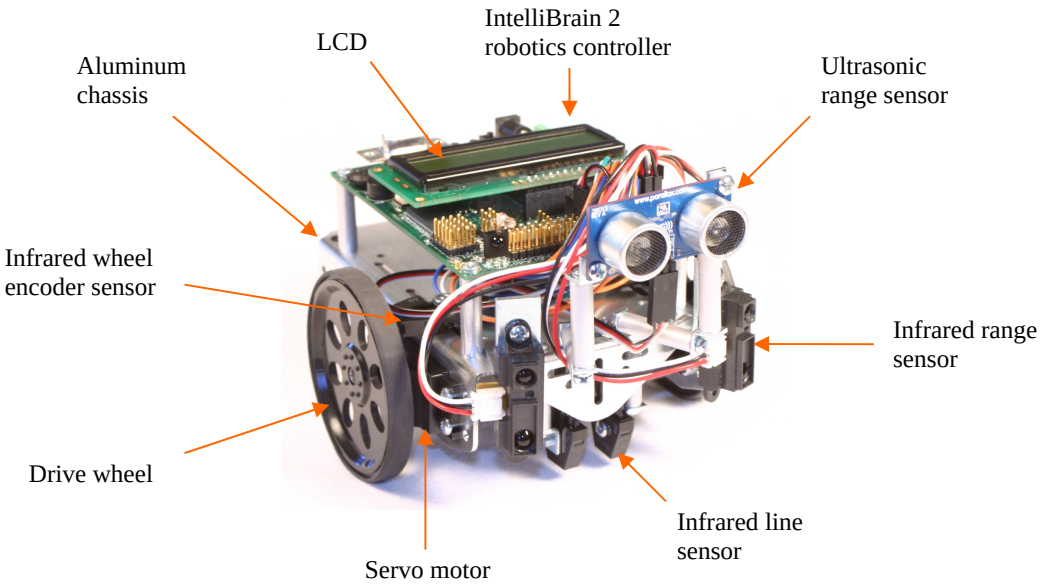


Figure 1-1 - IntelliBrain-Bot Deluxe Educational Robot

IntelliBrain-Bot Educational Robot Design

Figure 1-1 shows a fully assembled IntelliBrain-Bot Deluxe educational robot. As you can see in the figure, the robot is made up of the following major parts:

- IntelliBrain 2 robotics controller with LCD display
- aluminum chassis
- servo motors
- wheels
- assorted hardware
- sensors
- battery holder (not visible, under chassis)
- batteries (not visible, under chassis)

Mechanics

The IntelliBrain-Bot educational robot employs a simple mechanical design. An aluminum chassis fabricated from a single piece of sheet metal provides a sturdy central structure for the robot. Two motors mounted on the underside of the chassis drive the two large wheels, enabling the robot to move under its own power. A ball tail wheel supports the back end of the robot. The robotics controller, sensors, motors, tail wheel and battery holder mount directly on the chassis.

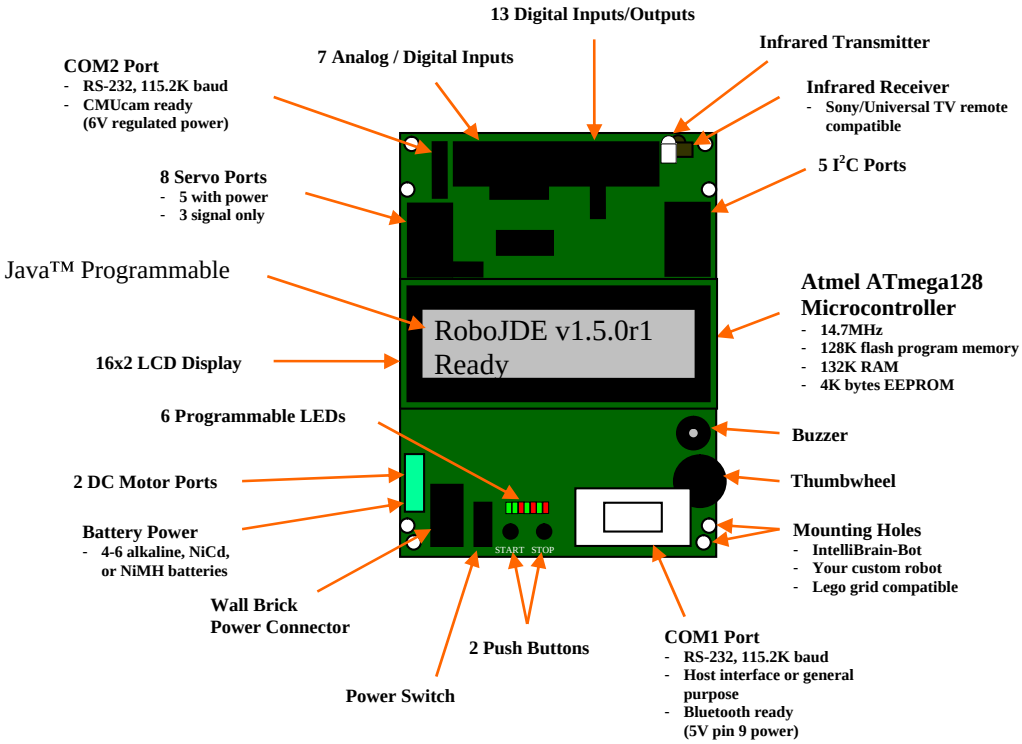


Figure 1-2 - IntelliBrain 2 Robotics Controller

IntelliBrain 2 Robotics Controller

The IntelliBrain 2 robotics controller, depicted in Figure 1-2, is the IntelliBrain-Bot educational robot's brain. A Java™ program executing on the IntelliBrain robotics controller enables the robot to function independently, analyzing input from the robot's sensors and by controlling the robot's motors to achieve desired results. By creating and loading different programs you can program the robot to perform a wide range of functions. A program can be very simple or quite complex. A program may be as trivial as displaying your name on the LCD screen or as complex as a program that controls a search and rescue robot.

Sensor and Motor Ports

Sensors and motors connect to the IntelliBrain 2 robotics controller via input and output ports. As you can see in Figure 1-1 and Figure 1-2, most of these ports consist of connector pins along the front edge (top edge in Figure 1-2) of the controller board. Each port has three or four pins: ground, power and signal pins. The ports are arranged side-by-side across the forward edge of the controller's circuit board. Each port is marked by a label on the controller board, which indicates the type of port and the number of the port. Table 1-1describes the ports available on the IntelliBrain 2 robotics controller.

Table 1-1 - IntelliBrain 2 Robotics Controller Ports

Port Type	Labels	Description
Analog	A1-A7	Analog ports use an analog to digital converter to read a voltage between 0 and 5 volts and convert it to an integer value between 0 and 1023, where 0 corresponds to 0 volts and 1023 corresponds to 5 volts.
Digital	IO1-IO13	Digital ports input or output a Boolean (on/off or true/false) signal. When configured as an output, a digital port outputs 0 volts when it is off (false) and 5 volts when it is on (true). When configured as an input, a digital port returns a false value when the signal is low (nearest 0 volts) and true when the signal is high (nearest 5 volts).
Servo	S1-S8	Servo ports interface directly to hobby servo motors. Servo motors were originally developed for use in model airplanes and, due to their light weight, low cost and simple electronics, are now commonly used in small robots.
Motor	M1, M2	Motor ports interface directly with conventional DC motors, using pulse width modulation (PWM) to vary the power output by each motor.
Serial	COM1, COM2	Serial ports interface to more complex sensors such as cameras and Global Positioning System (GPS) devices. By attaching a Bluetooth serial adapter to a serial port the robot can communicate wirelessly to a host computer or other robots. The COM1 port also serves as the connection to a PC when developing a program using the RoboJDE development environment.

Port Type	Labels	Description
I^2C	I2C	I^2C (pronounced i-two-c) ports interface to more advanced sensors and effectors such as compass sensors and speech synthesizers. There are five port headers for I^2C devices. The I^2C ports are not numbered individually because I^2C device addressing is controlled in software, not by the physical connection.
Infrared transmitter	none	The infrared transmitter provides a signal an on/off signal that can be detected by the infrared receiver (described below). This can be used for simple communication between robots to signal a Boolean (on/off) value, for example, signaling another robot to stop or go.

Human Interface Features

In addition to providing the ability to execute a Java program and to interface to sensors and motors, the IntelliBrain 2 robotics controller provides a number of features which make it easy to program the robot to interact with people. Table 1-2 lists these features.

Table 1-2 - Human Interface Features

Device	Description
Liquid Crystal Display	The Liquid Crystal Display (LCD) screen provides two, sixteen character lines of output. The LCD is useful to directly display sensor readings, providing an easy way to learn about and diagnose problems with sensors. The display is also useful for providing a simple menu-based user interface.
Push Button	Two push buttons, labeled START and STOP may be used like the buttons on a mouse, allowing a human to indicate choices. By default the buttons start and stop the Java program, but your program can use them for other purposes.
Thumbwheel	The thumbwheel works like a volume control knob on a car stereo. It provides a means for users to vary the setting of an analog value, such as motor speed, or a way to scroll through screens of output on the LCD display.
Buzzer	The buzzer can be programmed to beep, click or play a tune.
Universal remote control receiver	The universal remote control receiver allows the program to take input from a universal remote control, giving a human the ability to remotely control the robot or provide a keyboard type input for a more sophisticated human interface.
Light Emitting Diodes (LEDs)	Seven LEDs provide visual indications to humans. Six of these can be turned on, off or blinked under program control to provide a visual indication to the user. Three of the program controlled LEDs are green and three are red. The left most LED is a power indicator, which illuminates green when power is on. It cannot be controlled by the program. The LEDs are numbered 1 through 6, starting with the LED to the right of the power LED.

Sensors

The IntelliBrain-Bot Deluxe educational robot comes with seven sensors described in Table 1-3.

Table 1-3 – IntelliBrain-Bot Deluxe Educational Robot Sensors

Sensor	Description
Wheel Encoder Sensor (2)	Two wheel encoder sensors sense movement of the robot's wheels. These sensors use reflected infrared light to sense if a hole or a spoke is in front of the sensor.
Line Sensor (2)	Two line sensors mounted on the underside of the robot sense the presence of a non-reflective line. These sensors use infrared light to sense the reflectivity of the surface below the robot, allowing it to follow the course of a non-reflective, black line over a highly reflective, white background.
Infrared Range Sensor (2)	Two infrared range sensors sense the distance to objects between 4 and 30 inches in front of the robot. These sensors measure the angle of reflection of a narrow beam of infrared light to determine the distance to objects within their range. They can be used to detect and avoid obstacles in the robot's path.
Ultrasonic Range Sensor	One ultrasonic range sensor senses the distance to an object between 1.2 and 118 inches in front of the robot. This sensor measures distance by issuing a brief pulse of high frequency sound, a ping, and precisely measuring the time until the first echo returns. Using speed-of-sound calculations the distance to an object can be precisely determined. This sensor can be used to identify far off objects, walls and hallways. It is also can be used to create an invisible "tractor beam," as you will observe later in this lesson.

Batteries

Chemical energy in the four AA batteries mounted in the battery holder, on the underside of the chassis, provides the energy to power the motors and electronics.

A Hands-on Demonstration

Now for some fun, let's take a few minutes to see what the IntelliBrain-Bot educational robot can do.

Note: Your lab instructor should have loaded the IntelliBrain-Bot demo program into the flash memory of your robot and made sure the batteries are fully charged prior to this exercise.

The demo program provides examples of the robot using various sensors to perform different behaviors. It also provides a simple user interface, demonstrating the use of the IntelliBrain 2 robotics controller's human interface features, as well as a Sony compatible universal remote control. Finally, it provides a simple means to verify that each sensor is properly connected and functioning correctly.

Using the IntelliBrain-Bot Demo Program

The user interface of the IntelliBrain-Bot demo program enables you to toggle through a list of pre-programmed functions (behaviors) the robot can perform to select the function you want the robot to demonstrate. Try this by completing the following steps:

1. Switch the power on.

   The power switch is located toward the left side of the rear edge of the IntelliBrain 2 robotics controller, just left of the push buttons labeled START and STOP. Slide it toward the front of the robot.

2. Press the button labeled "START."

   This will start the demo program. You should see the following displayed on the LCD screen:

   ```
   Select Function
   Do Nothing
   ```

   The first line of text is telling you to select a function for the robot to perform. The second line is the name of the function.

3. Press the button labeled "STOP."

   The stop button is typically used to stop your Java program. However, it can also be used for other purposes. The demo program uses it to allow you to scroll through the list of functions.

   You should see the following displayed on the LCD screen:

   ```
   Select Function
   Play Tune
   ```

4. Press the STOP button repeatedly, observing the various functions which are available.

   The available functions are described in Table 1-4.

5. Continue to press the STOP button until the "Play Tune" function is displayed.

6. Press the START button.

   The robot will play Beethoven's tune, Ode to Joy, using the buzzer.

7. Switch the power switch to the off position.

Table 1-4 – IntelliBrain-Bot Demo Program Functions

Function	Description
Do Nothing	The robot does not attempt to move. This allows you to test the sensors. You will find this is extremely useful for debugging sensor problems.
Play Tune	Plays Beethoven's tune, Ode to Joy, using the buzzer.
Remote Control	Allows you to remotely control the robot using a Sony compatible infrared remote control. Use the channel up button to move the robot forward, the channel down button to move it backward, the volume up button to rotate right and the volume down button to rotate left. (Requires a Sony compatible infrared remote control. Most universal remote controls will work if programmed for a Sony television.)
Navigate Forward	Uses wheel encoder sensors and navigation classes (provided in the RoboJDE class library) to navigate the robot straight ahead 24 inches.
Rotate 180	Uses wheel encoder sensors and navigation classes to rotate the robot in place 180 degrees.
Navigate Square	Uses wheel encoder sensors and navigation classes to navigate the robot around a 16 inch square.
Random Dance	Uses software generated random numbers to perform a dance made up of random steps.
Follow Line	Uses line sensors to enable your robot to follow a black line on a white surface.
Avoid Obstacles	Uses wheel encoder sensors, navigation classes, and infrared range sensors to navigate your robot 24 inches forward and back to where it started, avoiding obstacles along the way.
Follow Object	Uses the ultrasonic range sensor to maintain a distance of 6 inches from an object in front of your robot, creating a "tractor beam" effect.

Testing Sensors

Once you have selected a function in the demo program and started it running, the LCD screen switches to displaying screens which give you a glimpse into the robot's view of the world. This will allow you to verify that all of the sensors are functioning properly.

Let's peer into the robot's brain to verify each sensor is operating properly.

1. Switch the power on.

2. Press START.

   "Do Nothing" should appear on the second line of the display. If it doesn't, press STOP repeatedly until it displays.

3. Press START again.

4. Use your finger to rotate the thumbwheel, observing the different screens which display as you move the wheel.

   The screens are described in Table 1-5.

Table 1-5 – IntelliBrain-Bot Demo Program Screens

Display	Description
IntelliBrainBot	Displays the program name and version.
L Wheel R Wheel	Displays the current raw analog values reported by the left and right wheel sensors. Turn a wheel and observe the change in the value reported by the associated sensor as spokes and holes pass in front of the sensor.
L Line R Line	Displays the current raw analog values reported by the left and right line sensors.
L Range R Range	Displays the current raw analog values reported by the left and right infrared range sensors.
Sonar Range	Displays the distance in inches to the nearest object in front of the sonar range sensor.
L Enc R Enc	Displays the current count values maintained by the encoders. Turn a wheel and observe the change in the count value. Note: the counter will not sense changes in direction when you turn the wheel by hand.
Pose	Displays the x and y coordinates of the robot in inches from the starting point and the heading in radians, with zero being the direction the robot was facing when the program started.

Wheel Encoder Sensors

5. Rotate the thumbwheel until you see the display referring to the wheel sensors.

6. Hold the robot in your right hand and use your left hand to slowly rotate the left wheel.

   Observe that the sensor reading displayed to the right of “L Wheel” varies between a low value of approximately 40 and a high value of approximately 1000 as you rotate the wheel. (The numbers are the readings of the sensors. In later chapters you will learn about how the IntelliBrain 2 controller uses its analog-to-digital converter to sample sensor readings.)

7. Switch hands and repeat the previous step, this time testing the right wheel sensor.

Line Sensors

8. Rotate the thumbwheel until you see the display referring to the line sensors.

9. Set the robot down with the sensors over a bright white surface. For example, the white area of the line following poster.

   Observe that both line sensors report a reading, below 300.

10. Set the robot down with the sensors over a non-reflective black surface. For example, the black line on the line following poser.

    Observe that both line sensors report a high reading, above 300.

11. Set the robot down with one sensor over a bright white surface and the other sensor over a non-reflective black surface. For example, straddling the line on the line following poster.

    Observe that the sensor over the white surface reads low, while the sensor of the black surface reads high.

Ultrasonic Range Sensor

12. Rotate the thumbwheel until you see the display referring to the left and right range sensors.

13. Hold the robot up such that there are no objects within four feet of the robot.

    Observe that both sensors read a very low value, typically less than 10.

14. Hold your hand approximately 3 inches in front of the left range sensor.

    Observe the left sensor reading is approximately 500.

15. Repeat the previous step for the right range sensor.

Sonar Range Sensor

16. Rotate the thumbwheel until you see the display referring to the sonar sensor.

17. Hold your hand in front of the sonar sensor.

    Observe as you move your hand the distance value displayed on the screen tracks the distance your hand is from the sensor.

18. Switch the power off.

"Tractor Beam" Demonstration

The "Follow Object" function of the demo program implements an invisible "tractor beam" by using the sonar range sensor to maintain a fixed distance of 6 inches between the robot and an object ahead of it.

1. Set the robot on the floor with a few feet of clear space around it.

2. Start the demo program and select the "Follow Object" function.

3. Place your hand approximately six inches in front of the robot.

4. Slowly move your hand away from the robot.

   Observe the robot follows your hand forward.

5. Slowly move your hand toward the robot.

   Observe the robot backs away from you hand.

6. Switch the power off.

Navigation Demonstration

The demo program includes three functions which demonstrate the IntelliBrain-Bot educational robot's ability to navigate, "Navigate Forward," "Rotate 180," and "Navigate Square." These functions use the wheel sensors to keep track of the robot's position. If it drifts off course, the program quickly compensates by adjusting power to the motors to steer it back on course.

1. Set the robot on the floor with at least 3 feet of clear space in front of it.

2. Start the demo program and select the "Navigate Forward" function.

   Observe the robot drives straight ahead for 2 feet, then stops.

3. Press STOP.

4. Set the robot on the floor and select the "Rotate 180" function.

   Observe the robot turns in place approximately 180 degrees.

5. Press STOP.

6. Set the robot on the floor with at least 3 feet of clear space in all directions.

7. Select the “Navigate Square” function.

   Observe the robot drives in a 16 inch square pattern.

8. Switch the power off.

Random Dance Demonstration

The “Random Dance” function of the demo moves the robot in a never ending series of small random moves. Because the robot has equal probability to move any direction, it will not drift far from where it started as it performs this unusual dance.

1. Set the robot on the floor with a few feet of clear space around it.

2. Start the demo program and select the “Random Dance” function.

   Observe the robot dances about randomly, but doesn’t drift far from where it started.

3. Switch the power off.

Collision Avoidance Demonstration

One of the primary uses of the infrared range sensor is to avoid collisions with objects in the robots path. The “Avoid Obstacles” function of the demo program demonstrates how the robot can steer around obstacles in its way.

1. Set the robot on the floor with three feet of clear space in front of it.

2. Place an object approximately the same size as the robot roughly one foot in front of the robot.

3. Start the demo program and select the “Avoid Obstacles” function.

   Observe the robot will drive to a point 2 feet ahead of it, detecting and steering around the obstacle in its path and then return to where it started.

4. Switch the power off.

Line Following Demonstration

In case you haven’t guessed it already, the line sensors enable the robot to follow a line on the floor. You will need either a line following poster or a one inch wide strip of non-reflective black electrical tape on a white surface to complete this demonstration.

1. Set the robot on the floor over the black line.

2. Start the demo program and select the “Follow Line” function.

Observe the robot follows the line.

3. Switch the power off.

Remote Control Demonstration

The IntelliBrain-Bot educational robot can receive input from a universal remote control. You will need a universal remote control configured to control a Sony television to complete this demonstration.

1. Set the robot on the floor with several feet of clear space around it.

2. Start the demo program and select the “Remote Control” function.

3. Press and hold the next channel button for a moment.

   Observe the robot moves forward while you hold the button down.

4. Press and hold the previous channel button for a moment.

   Observe the robot moves backward while you hold the button down.

5. Press and hold the increase volume button for a moment.

   Observe the robot rotates clockwise while you hold the button down.

6. Press and hold the decrease volume button for a moment.

   Observe the robot rotates counter clockwise while you hold the button down.

7. Steer the robot around the room using these four control buttons.

8. Switch power off.

Summary

You should now be familiar with the features of the IntelliBrain-Bot educational robot and its construction. Through the hands-on demonstration you have seen many of the capabilities you will learn to program yourself in subsequent chapters.

Exercises

1. Complete the parts list in Table 1-6 by inspecting the robot and filling in the missing information.

2. Locate the ports on the IntelliBrain 2 robotics controller and fill in Table 1-7.

3. Locate the human interface features of the IntelliBrain 2 robotics controller and fill in Table 1-8.

4. Trace wires from each sensor and motor to the port on the IntelliBrain 2 robotics controller it connects to. Record the label and type of the port in Table 1-9.

5. Using the demo program experiment with each sensor and record the minimum and maximum reading you observe as you experiment with the sensor in Table 1-10. Note the circumstances when you observed the minimum and maximum readings for each sensor.

Table 1-6 - IntelliBrain-Bot Deluxe Educational Robot Parts List

Qty	Part	Description
1		Acts as the robot's brain by executing a Java programs and interfacing to sensors, motors and humans.
1	Aluminum chassis	
2	Servo motor	
2		Coverts torque of from the motor shaft to force to move the robot forward or back.
1	Ball tail wheel	
	Tire	Provides traction (friction) so the wheels don't slip.
2		Uses reflected infrared light to enable the robot to sense and follow a line on the floor.
2		Uses reflected infrared light to enable the robot to sense the distance to an object between 4 and 30 inches away.
1		Measures the time between high frequency sound pulses and their echoes to sense the distance to an object between 1.8 and 118 inches away.
2		Uses reflected infrared light to sense rotation of a wheel, enabling the robot to track its position monitoring wheel movement.
	Battery holder	Holds the batteries on the underside of the chassis.
4	Batteries	
	Aluminum standoff	Used to mount the robotics controller, line sensors and ultrasonic range sensor.
	1" corner bracket	Used to mount infrared range sensors.
	Right angle bracket	Used to mount line sensors and ultrasonic range sensor.
	Screws	Used to fasten parts together.
	Nuts	Used to fasten parts together.
	Washer	Aluminum or nylon washer used in mounting sensors.
	Cotter pin	Used to attach tail wheel.

Table 1-7 – IntelliBrain 2 Robotics Controller Ports

Port Type	Label(s)	Location
Servo motor		
Analog		
Digital		
I^2C		
Motor		
Serial		

Table 1-8 – IntelliBrain 2 Robotics Controller Human Interface Features

Feature	Label(s)	Location
Liquid Crystal Display	- none -	
Push button		
Thumbwheel		
Buzzer		
Universal remote control receiver	- none -	

Table 1-9 – IntelliBrain-Bot Deluxe Educational Robot Sensor and Motor Connections

Sensor/Motor	Port (Label)	Port Type
Left servo motor		
Right servo motor		
Left wheel encoder sensor		
Right wheel encoder sensor		
Left infrared range sensor		
Right infrared range sensor		
Left line sensor		
Right line sensor		
Ultrasonic range sensor		

Table 1-10 – Sensor Readings

Sensor	Min	Max	Notes
Left wheel encoder sensor			
Right wheel encoder sensor			
Left infrared range sensor			
Right infrared range sensor			
Left line sensor			
Right line sensor			
Ultrasonic range sensor			

CHAPTER 2

Programming the IntelliBrain-Bot

In the previous chapter you became familiar with the hardware features of IntelliBrain™-Bot Deluxe educational robot. You also observed the robot in action by working with the demo program. In this chapter you will begin to learn about the software features of the robot, as well as robotics programming concepts, the focus of this book. You will use the RoboJDE™ Java™-enabled robotics software development environment to create, build, load and run your first program. You will also learn debugging techniques which will help you quickly resolve problems with your program.

Note: The RoboJDE development environment should be installed on the computer you will be using prior to proceeding with the hands-on activities in this chapter. Your lab instructor has most likely already taken care of this. However, if you are working on your own, follow the instructions in the *RoboJDE User Guide* to install the RoboJDE software.

Creating a New Project

To begin a new project you must create a new RoboJDE project file to store the project's properties. RoboJDE uses project files to make it easy for you to switch between different robotics software projects. Use the following procedure to create a project named "MyBot:"

1. Start the RoboJDE development environment from the Windows start menu. The default location on the start menu is start->All Programs->RoboJDE->RoboJDE.

   The RoboJDE Graphical User Interface (GUI) will appear.

2. Select File->New Project menu item in the RoboJDE GUI.

   The Project Properties dialog will appear.

3. Click the browse button to the right of the "Project folder" field.

   The Choose File dialog will appear.

4. Browse to and select the folder in which you want to create your project.

   Note: You can create a new folder by browsing to the location where you want to create a new folder then clicking on the create folder button. A folder titled "New Folder" will appear. Click on the new folder's name and change it to a name you choose. Then click on the folder icon to the left of the name to select it. Click OK.

5. Enter the name "MyBot" in the "Main class" field.

6. Click OK.

   The MyBot class will be created, as shown in Figure 2-1.

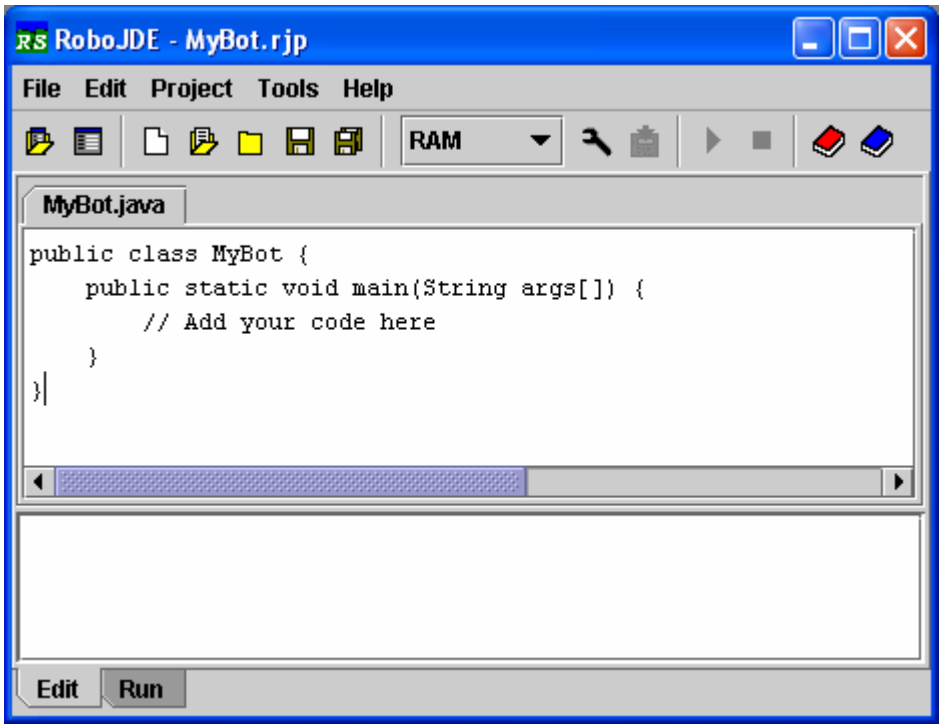


Figure 2-1 - MyBot Project in RoboJDE

7. Using the mouse, select the text "// Add your code here" and replace it with "System.out.println("YourName");" replacing YourName with your name, so your program looks similar to the following:

```
public class MyBot {
    public static void main(String args[]) {
        System.out.println("Mr. Roboto");
    }
}
```

   Note: Java is very particular about details such as upper and lower case letters

and punctuation. Paying careful attention to these details will save you a lot of time and frustration debugging subtle errors in your programs!

8. Click the Save all button (see Figure 2-2) or select the File->Save All menu item.

   The Save dialog will appear with "MyBot" as the proposed file name.

9. Click the Save button to save your project.

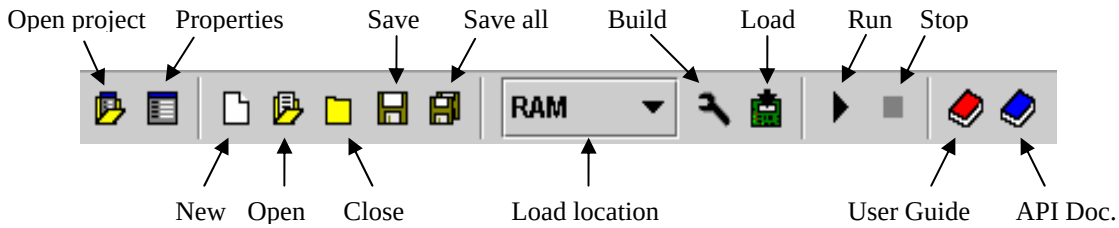


Figure 2-2 - RoboJDE Tool Bar

Your first program is now ready to try out. First you must connect the robot to your computer. This will enable you to download your program into the IntelliBrain 2 robotics controller's memory.

Connecting the Robot to Your Computer

RoboJDE communicates with the IntelliBrain-Bot educational robot via a serial port on your computer. All you need to do to establish a connection is connect a cable between the port labeled "COM1" on the IntelliBrain 2 robotics controller and the serial port you will be using on your computer.

Note: Your lab instructor should have already attached the cable to the serial port on your computer and configured appropriate settings in RoboJDE. However, if this is not the case, please consult the *IntelliBrain 2 User Guide* and *RoboJDE User Guide* for further instructions.

Connect the robot to your computer by doing the following:

1. Locate the load button on the RoboJDE tool bar (see Figure 2-2) and note its state.

   When the RoboJDE GUI is unable to communicate with the robot the load button will appear gray, as it does in Figure 2-1. Since the robot is not currently connected to your computer the button is gray.

2. Gently attach the free end of the serial cable to the port labeled "COM1" on the IntelliBrain 2 robotics controller.

Note: The cable will slide on to the connector more easily if you gently rock the cable left and right as you press it on to the connector. Although the connector on the robot is quite sturdy, be careful not to apply excessive force, which may damage the robot.

3. Switch the power on.

   If the cable is connected properly and the RoboJDE communication settings are correct, the Load button will go from gray to green.

   If the Load button did not turn green, request help from your lab instructor to ensure the settings in RoboJDE (Tools->Settings) and the baud rate setting in the IntelliBrain 2 robotics controller are correct. In most cases the baud rate should be set to 115.2K in both the RoboJDE GUI and on the IntelliBrain 2 robotics controller. Also check to be sure the Serial Port setting in the RoboJDE GUI is the port the cable connects to on your computer. Finally, be sure the Board Type setting in the RoboJDE GUI is set to "IntelliBrain."

4. Switch the power off.

Running Your First Program

Everything is now set to give your program a try. You will need to build it, download it and run it. Fortunately, this is much easier than it sounds – only two mouse clicks!

Do the following to give your program a try:

1. Switch the power on.

2. Click the Load button in the RoboJDE GUI.

   This will compile, link and load your program. You will see messages from the compiler and linker in the output window at the bottom of the RoboJDE GUI window. If you typed everything correctly, there will be no errors and the Load progress window will display briefly. If you made a mistake, you will see error messages in the output window.

   Once the Load progress window disappears, the LCD screen on the robot will display the following message on the second line:

   ```
   Ready – RAM
   ```

   This indicates there is now a program loaded in Random Access Memory (RAM) which is ready to run.

3. Click the Run button on the RoboJDE tool bar to run your program.

Your program will run very quickly. If you watch the LCD screen you will see your name appear momentarily. Click the Run button again if you missed it.

Also notice the RoboJDE GUI switched to its Run window, where your name was also output by your program. By default, any output to "System.out" goes to both the LCD screen and the RoboJDE Run window if the serial cable is connected.

4. Press the START button.

   This also runs your program, but without clearing the output in the RoboJDE Run window. Each time you press the START button another line displaying your name will appear.

5. Switch power off.

Congratulations! You have now created and run your first robotics program!

Programming Concepts

If this is the first program you've ever created or you are new to Java, you are probably a little vague on many of the concepts we've covered so far. If you don't fully understand your program, rest assured, as you work through the hands-on lessons in this book your understanding will become clearer.

What is a Program?

A program is a series of instructions a computer executes in steps. The computer executes one step then proceeds on to the next step, executing it and proceeding on to the next step, and so on, repeating this process until it reaches the end of your program.

Another way to think of a program is as a recipe. With a recipe, the chef is the computer. He or she "executes" the recipe by starting at the beginning and following the steps in order. Likewise, a computer executes your recipe (program) step by step.

The Method Named "main"

In the case of the IntelliBrain-Bot educational robot, the IntelliBrain 2 robotics controller is a small computer. It executes the steps of your program. It begins executing your program in the method named "main."

Look for the word "main" on the second line of the program you just created. This is the start of the main method. Your program begins executing on the line after this – the line that contains your name. Your program is very simple. It has only one step, which prints your name. Once the robotics controller executes this step, it reaches the end of the main method and exits your program, which explains why your name was only displayed on the LCD screen for a split second.

You can change this behavior by adding one more step to your program that will wait for the STOP button to be pressed. This will cause your program to display your name then wait for you to press the STOP button before exiting. Do this as follows:

1. Add the following import statement as the first line of your program

   ```
   import com.ridgesoft.intellibrain.IntelliBrain;
   ```

   This tells he compiler your program will be using the IntelliBrain class from the class library.

2. Add a statement to wait for the STOP button to be pressed after your name has been printed, so your program looks similar to the following:

   ```
   import com.ridgesoft.intellibrain.IntelliBrain;
   public class MyBot {
       public static void main(String args[]) {
           System.out.println("Mr. Roboto");
           IntelliBrain.getStopButton().waitPressed();
       }
   }
   ```

3. Switch power on.

4. Click the load button.

5. Click the run button in the RoboJDE GUI or press the START button on the robot.

   Observe your name does not disappear from the LCD screen. This is because your program is waiting for the STOP button to be pressed.

6. Press the STOP button.

   Observe your program has stopped running and your name is no longer displayed on the LCD screen.

7. Switch power off.

Your program now includes two steps, one which tells the computer to display your name and another which tells the computer to wait for someone to press the STOP button.

Your programming Process

As you develop programs for your robot you will become very familiar with your programming process, which is illustrated in Figure 2-3. You will use this process over and over to program the IntelliBrain-Bot educational robot. Each time you create a new program or make a change to an existing program, you will complete the following steps:

1. **Edit** – add, modify or delete steps in your program (use the RoboJDE edit window).
2. **Build** – compile, link and download your program to the robot (click the load button).
3. **Test** – test your program (click the run button on the RoboJDE tool bar or press the START button on the robot).

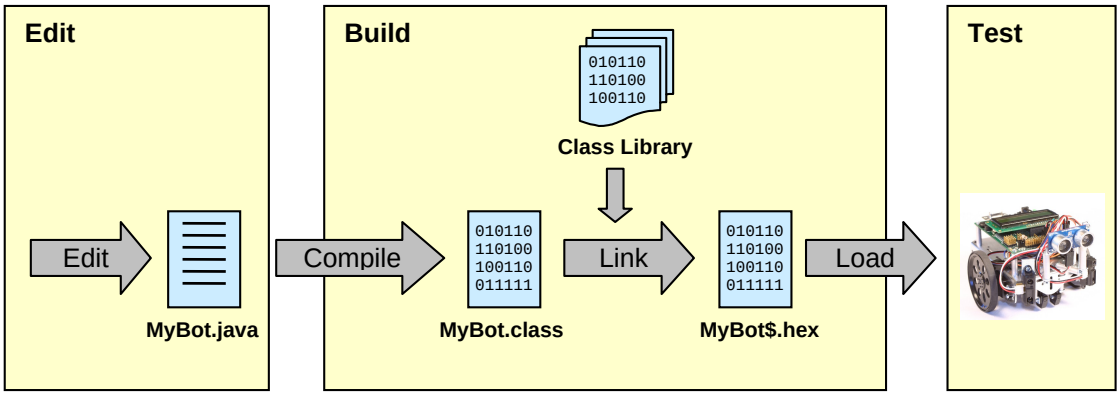


Figure 2-3 – The IntelliBrain-Bot Educational Robot Programming Process

You have now completed this process twice, once when you created your first program, and again when you added a step to it. You will repeat it many more times as you develop larger and more sophisticated programs.

Most seasoned programmers develop programs iteratively, making only one small change at a time, testing it, then moving on to the next small change, and so on, until they have completed their project. Making and testing many small changes has an advantage over making fewer large changes. It is much easier to thoroughly test your changes, as well as find and fix problems, when you haven't made large changes. By keeping changes small, you focus your attention one very small area. If your program doesn't work after you have changed it, it will be easier to resolve the problem if you haven't made a large change. We recommend you follow this approach whenever possible, making and testing small changes, rather than attempting large changes.

Behind the Scenes

Let's take a close look at Figure 2-3 to examine what goes on behind the scenes. If you browse to the folder where you created your program you will see the following files:

1. **MyBot.java** – the Java source file
2. **MyBot.class** – the Java class file generated by the compiler
3. **MyBot$.hex** – the executable file generated by the linker
4. **MyBot.rjp** – a file containing the project's properties

The text you entered into the RoboJDE edit window to define the MyBot Java class was saved in the file named MyBot.java. This is the Java source file for your main Java class, MyBot.

When you clicked the load button, three things took place. The MyBot program was compiled, linked and loaded, as depicted in the block labeled "Build" in Figure 2-3. In the first step, the MyBot class was compiled from its source file, MyBot.java. This generated the Java class file, MyBot.class. In the second step, the MyBot.class file was linked with other classes MyBot references, which are included in the class library. The class named "System," is a class from the class library which is referenced by your program. The linked program was stored in the executable file, MyBot$.hex. Finally, the executable file was transferred via the serial cable and loaded into the memory of the IntelliBrain 2 robotics controller, which allowed it to be run and tested.

The compiler parses and analyzes the text in a Java source file and produces binary "byte codes," which the RoboJDE Java virtual machine can execute. The virtual machine resides on the IntelliBrain 2 robotics controller and enables it to execute Java code rather than native machine code particular to the microcontroller chip at the core of the IntelliBrain 2 robotics controller.

The linker assembles all of the classes that are needed to execute your program into a single file. While doing this, the linker also creates the necessary linkages between the classes. These linkages provide the virtual machine with the information it needs to understand how the classes interact to form your program. You only had to create one very small Java class to create your program, but your program cannot execute without including many more classes from the class library. You can see how many classes are included in the executable by reviewing the output from the linker at the bottom of the RoboJDE edit window. Surprisingly, your simple program references approximately fifty other classes from the class library! The classes in the library provide a rich foundation on which you create your programs. This allows you to focus your effort on the algorithms that control your robot, rather than getting bogged down in low level details.

Debugging Your Programs

If you are like most programmers, your programs will rarely work on the first try. Typically, once you finish making edits, you will find you are faced with one or more compilation errors. You will need to make further edits to your program to correct your errors. Once your program compiles and links successfully, you will be able to download it to the robot and run it, but you will frequently find it doesn't do what you expect. When this happens, you will need to analyze what your program is doing and determine what changes are necessary to make it do what you intend.

The ability to debug problems is an essential programming skill. You will be able to complete your programming projects faster and your programs will often work better if you take the time to develop and apply debugging skills. In contrast, if you try to program without learning how to debug effectively, you will likely find programming a

frustrating experience. Take the time to analyze and understand problems. It will make you a better programmer and you will find programming more enjoyable!

Compilation Errors

In order to convert your programs from the text you enter, which is called "source code," into Java byte codes the RoboJDE Java virtual machine can execute, the compiler needs to read your source code, understand it, and translate it to a form the virtual machine understands. Conceptually, this is similar to translating a document from a foreign language to your native language. In order to translate such a document you would apply your knowledge of the vocabulary and grammar of the foreign language to understand the source document. Once you understood it you could then express it's meaning using the vocabulary and grammar of your native language.

Imagine if you were given the task of translating a document that was full of spelling errors, slang, poor grammar, punctuation errors and ambiguities. This would make your translation job much more difficult. It's likely you wouldn't have a lot of confidence that you could accurately communicate the thoughts and emotions of the author in your translated version.

Similarly, the Java compiler isn't able to reliably convert your Java source code to executable code if it contains misspellings, incorrect grammar, words the compiler doesn't know, poor punctuation and ambiguities. Rather than trying to guess what you intended the Java compiler outputs an error message for each problem it encounters. Each error message indicates where in your source code the compiler encountered a problem followed by a message describing the problem.

Let's run some experiments to see what the compiler does when you introduce errors into your program. Try the following:

1. Edit the fifth line of your program to replace the period between "out" and "println" with a comma so it looks like this:

   ```
   System.out,println("Mr. Roboto");
   ```

2. Click the build button (wrench icon) on the RoboJDE tool bar.

   Observe the compilation error reported in the output pane at the bottom of the edit window. You will see an error message similar to the following:

   ```
   Found 1 syntax error in "MyBot.java":
        5.                 System.out,println("Mr. Roboto");

                                          ^
   *** Syntax: . expected instead of this token
   ```

   The first line of the message indicates there is a syntax error in MyBot.java. The second line shows the problem line from your program and the number of the

line, to the left. The third line indicates the location of the error in the problem line using a carat (^) character. The fourth line tells you what the problem is. In this case, the compiler expected a period instead of the "token" pointed to by the carat, a comma.

3. Select the menu item Edit->Go to Line in the RoboJDE GUI or enter Ctrl-G using the keyboard.

   The Go to Line dialog will appear.

4. Enter the number of the line with the error, "5" and click OK.

   RoboJDE will scroll to the line and highlight it. Since your program is very short, this may not seem necessary. When your programs get larger you will find this feature very useful. For example, if you had an error on line 327 of a 500 line program, you would really appreciate being able to jump right to the line rather than having to scroll around looking for it!

5. Correct the error and click the build button, again.

There are too many possible compilation errors to discuss them all here. The key to debugging them is to carefully read the messages from the compiler and understand what they are telling you. Always scroll up to the first error message and try to fix it first. Subsequent error messages are often due the first problem. When you fix the first problem, it is often best to re-compile immediately because the fix may eliminate subsequent errors. Re-compiling is quick and easy, so don't hesitate to do it often. Just click the build button or load button on the RoboJDE tool bar.

Exceptions and Stack Traces

In addition to encountering errors when you compile your programs, the virtual machine, which executes your program on the robot, is able to catch many errors that can only be detected while your program is running. For example, if your program attempts to use more memory than is available, the virtual machine will detect the problem and "throw an exception." There are many other types of exceptions, such as attempting to divide by zero, or attempting to use a reference to an object when the reference is "null" (not referring to any object).

Without going into all of the details of exceptions, let's take a quick look at what you will see when an exception gets "thrown." We make a small change in your program to switch to using a variable to keep track of your name, but we will introduce a bug while making this change. Do the following:

1. Insert the following line into your program as the second line:

```
private static String myName;
```

This line creates a variable to keep track of you name.

2. Modify the printing statement in your program, replacing the quoted string containing your name with name of the new variable, so your program looks like this:

```
import com.ridgesoft.intellibrain.IntelliBrain;
public class MyBot {
    private static String myName;
    public static void main(String args[]) {
        System.out.println(myName);
        IntelliBrain.getStopButton().waitPressed();
    }
}
```

3. Switch the power on.

4. Click the load button to build and download your program.

   Click the run button.

   You will see the following in the run window:

```
NullPointerException
    at java.io.PrintStream.print(PrintStream.java:44)
    at java.io.PrintStream.println(PrintStream.java:96)
    at MyBot.main(MyBot.java:5)
    at com.ridgesoft.intellibrain.StartupThread.run(StartupThread.java:31)
```

   This is the type of output you will see when your program causes an exception to be thrown. In this case, the exception is a "NullPointerException." The lines that follow indicate exactly which statements in your program resulted in the exception being thrown. This is a "stack trace." This stack trace shows your program was executing the PrintStream class's print method at line 44 of a file named PrintStream.java when an attempt to use a null reference (pointer) occurred. This class happens to be in the RoboJDE class library and is most likely not the source of problem, it's just where the problem showed up. The next line of the stack trace shows the print method was called by the println method, again in the PrintStream class. The third line of the stack trace indicates line 5 of the MyBot class called the println method.

5. Click the Edit tab in the lower left corner of the RoboJDE GUI.

6. Type Ctrl-G at the keyboard.

7. Enter 5, the line indicated in the stack trace, in the Go to Line dialog and click OK.

   This will show you the line in your program that was executing when the

NullPointerException occurred. Examining this line you will see it does indeed cause the println method to execute, as the stack trace indicated. This is the line you just modified to switch to using the new variable you added. The NullPointerException must be due to this change.

When you added the new variable to your program, we neglected to tell you to initialize the variable with your name, therefore, the variable is null. This is a bug.

8. Correct the bug by initializing the variable with a text string containing your name, similar to the following:

```
private static String myName = "Mr. Roboto";
```

9. Click the load button.

10. Click the run button.

    Observe your program once again works correctly!

11. Switch the power off.

Debugging Using Print Statements

Frequently your programs will compile and run just fine but still not work the way you expect. Often, the best way to solve these types of problems is to add print statements to your program. This will enable you to better understand what your program is doing. Being able to peer into your robot's mind is such a valuable debugging and test tool that it is a good idea to start by implementing these features first, knowing they will come in handy as you develop the main features of your program. For example, when working with a new sensor, a great place to start is by displaying the sensor's reading on the LCD screen. This allows you to check that the sensor is connected and functioning properly.

Let's add a print statement to your program to indicate when the STOP button has been pressed.

1. Add the following print statement to your program immediately after the statement to wait for the button to be pressed.

```
System.out.println("STOP pressed!");
```

2. Click the load button to build and download your program.

3. Click the run button.

   Observe your program has printed your name in the RoboJDE Run window.

4. Press the STOP button.

   Notice, unfortunately, the new message you just added did not display. Your program is not working quite the way you may have expected!

5. Add another print statement just before the statement to wait for the button to be pressed.

```
System.out.println("Waiting for STOP");
```

6. Build and run your program again.

   Observe your new message does show up in the RoboJDE run window and on the LCD screen, but the message indicating the STOP button has been pressed still does not display.

   It turns out your program has a minor bug in it. The virtual machine, which executes your program, assumes by default it is responsible for monitoring the STOP button. When the button is pressed the virtual machine immediately stops your program rather than letting it continue. In this case, the new print statement never executed because the virtual machine stopped your program before it got to the print statement.

   This problem is easy to fix.

7. Insert the following statement just before the statement that prints “Waiting for STOP:”

```
IntelliBrain.setTerminateOnStop(false);
```

   This tells the virtual machine it should not terminate your program when the STOP button is pressed.

   Your program should now be similar to the following:

```
import com.ridgesoft.intellibrain.IntelliBrain;
public class MyBot {
    private static String myName = "Mr. Roboto";
    public static void main(String args[]) {
        System.out.println(myName);
        IntelliBrain.setTerminateOnStop(false);
        System.out.println("Waiting for STOP");
        IntelliBrain.getStopButton().waitPressed();
        System.out.println("STOP pressed!");
    }
}
```

8. Load and run your program.

Observe, your program now prints all of the messages, as expected. You have debugged the problem!

Other Methods of Debugging

There are many other ways to debug your programs. In later chapters you will learn about programming the IntelliBrain controller's Light Emitting Diodes (LEDs) and its buzzer. These provide additional means of indicating what your program is doing.

Summary

Now that you have created your first program and made a number of modifications to it, you should be familiar with the process of programming the IntelliBrain-Bot educational robot. You should also be familiar with the types of errors you are likely to encounter as you continue to learn about programming the robot. If you encounter problems you are unable to resolve, refer back to this chapter to remind yourself of the debugging techniques you have learned.

Exercises

1. Create a new project and program to output the message: "Testing 1 2 3."
2. Briefly describe what a computer program is.
3. Describe the differences between a Java source file, a Java class file and an executable file. If you created a program with the main class named "Test" what would be the names of the source file, class file and executable file.
4. List the three main steps that comprise the programming process.
5. Describe what the compiler does.
6. Describe what the linker does.
7. Describe the purpose of the class library.
8. Describe what happens when you download a program to your robot.
9. Modify your program to introduce a compilation error. Write down the error message you receive. Describe how the message relates to the actual error in your program. Fix this error and introduce a different error, repeating this process until you have caused at least three different error messages to be emitted from the compiler.
10. If you were to receive an exception from a program with a stack trace containing the following line of text: "at Test.main(Test.java:173)," what does this tell you? How could you further investigate the source code that relates to this message?

CHAPTER 3

Maneuvering the IntelliBrain-Bot

Now that you are familiar with how to create, build and test programs for the IntelliBrain™-Bot educational robot, you are probably eager to program your robot to do more than just display your name. We will now do just that, by learning how to program the robot to maneuver. In addition, you will also learn how to use the RoboJDE™ API documentation and other documentation to help you accomplish your programming tasks. You will also use a few more features the Java™ programming language.

Differential Drive Robots

The IntelliBrain-Bot educational robot uses a differential-drive system to enable it to move and steer. This may sound complicated, but it is really very simple. There are two wheels, powered by two independently controlled motors. Your program will control the speed and direction of the robot by controlling the power delivered to each of the motors. As illustrated in Figure 3-1, your program can make the robot perform a few basic maneuvers simply by controlling the direction of rotation of each of the motors. The robot will move forward when your program applies the same amount of power to both motors. The robot will rotate in place if your program applies the same amount of power to the motors but in the opposite direction. Applying reverse power to the left wheel and forward power to the right wheel will cause the robot to rotate counter-clockwise. Likewise, applying forward power to the left wheel and reverse power to the right wheel will cause the robot to rotate counter-clockwise.

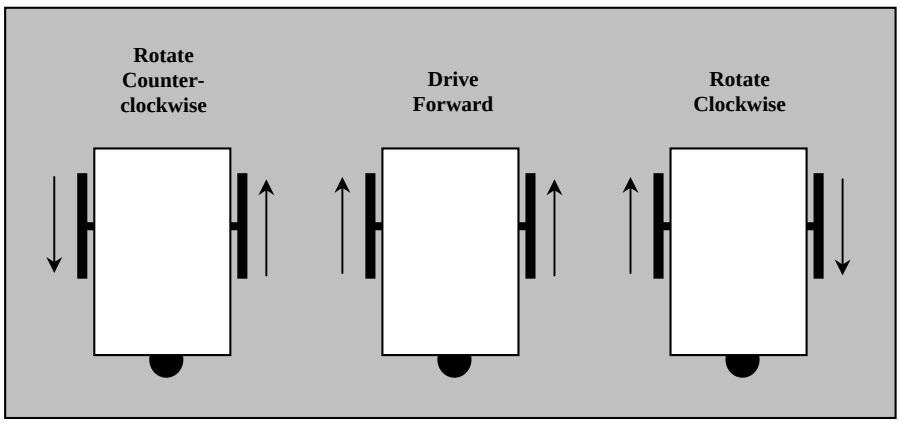


Figure 3-1 - Maneuvering a Differential Drive Robot

Programming Simple Maneuvers

The IntelliBrain-Bot educational robot uses continuous rotation hobby servos for its motors. These motors are based on hobby servos, which were originally designed for use in model airplanes. Standard hobby servos have built-in control circuitry and mechanics designed to rotate the servo's output shaft to a specific position and hold that position. This works well for controlling a model airplane, but it isn't suitable for driving the wheels on a robot.

Robotics researchers discovered hobby servers could be easily modified for continuous rotation, to provide inexpensive motors to drive the wheels of their robots. They did this by removing the mechanical stops and disabling the position sensing circuitry in the servos. Fortunately, this became so common that you can now buy servos manufactured for continuous rotation, eliminating the need to modify them yourself. The servos included in the IntelliBrain-Bot educational robot were manufactured as continuous rotation servos.

While continuous rotation servos can power your robot's wheels just like conventional DC motors, they must be controlled using their built-in control circuitry. This circuitry includes a position input signal intended to communicate the desired position of the servo's output shaft. For continuous rotation servos the position signal actually controls the direction of rotation and amount of power applied to the motor. Rather than working directly with control signals of the servos, the RoboJDE class library provides a class, ContinuousRotationServo, which provides a "wrapper," which enables your program to control the servos as if they were conventional motors.

Before we get started writing a program to control the servos, let's first investigate the classes we will use from the class library.

Using the Programming Documentation

Before you embark on any programming project you first need to understand how to accomplish the task at hand. A good way to do this is to consult the programming documentation. The following documents that will help you understand how to program the IntelliBrain-Bot educational robot: the *IntelliBrain 2 API Quick Reference*, the *IntelliBrain 2 User Guide* and the *RoboJDE Application Programming Interface (API) Documentation*.

The quickest way to learn about programming IntelliBrain 2 features is to consult the *IntelliBrain 2 API Quick Reference*. This can be found on the inside of the back cover of this book, or in the file IntelliBrain2API.pdf in "docs" folder where RoboJDE is installed. It is also available online at www.ridgesoft.com. Take a moment to locate the quick reference document and identify information regarding programming the servo ports.

Further details regarding the many features of the IntelliBrain 2 robotics controller are provided in the *IntelliBrain 2 User Guide*. Take a moment to locate this document and read the section regarding the servo ports. The *IntelliBrain 2 User Guide* is available at www.ridgesoft.com and on the CD-ROM that came with the robot.

The *RoboJDE API Documentation* contains the most detailed programming information. It is essential that you become familiar with using the API documentation. You will refer to it frequently as you program the IntelliBrain-Bot educational robot. The *RoboJDE API Documentation* is in Javadoc format. This is the format used to document most Java APIs. Becoming proficient at using the RoboJDE API documentation will help you become proficient at using similar documentation for other Java programming projects.

Figure 3-2 - API Documentation Button on RoboJDE Tool Bar

Click the API documentation button on the RoboJDE tool bar (shown in Figure 3-2) to display the API documentation in your web browser. This will launch your web browser and display the documentation, as shown in Figure 3-3.

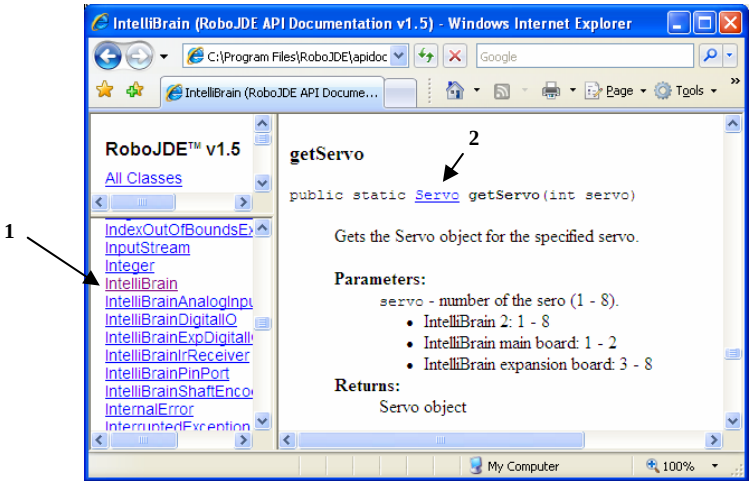


Figure 3-3 - IntelliBrain API Documentation

The documentation for the IntelliBrain class is the best place to start when you are learning about a feature you have not used before. You can display the documentation for the IntelliBrain class by scrolling to and clicking on the class name in the list of classes, as indicated by reference 1 in Figure 3-3. This will display documentation for the class in the pane on the right hand side. By browsing through the class documentation, you can find the methods to access various features of the IntelliBrain or IntelliBrain 2 robotics controller.

Complete the following steps to learn about the programming interface you will be using to control the servo motors:

1. Locate the API documentation for the IntelliBrain class.

2. Locate the documentation for the getServo method of the IntelliBrain class.

3. Locate and review the documentation for the ContinuousRotationServo class.

4. Locate and review the documentation for the Motor interface.

Programming the Robot to Drive Forward

You should now have a general idea of how you can maneuver write a program to maneuver the robot. The robot will perform maneuvers determined by how your program applies power to the motors. Your program can control the motors via the RoboJDE API. Let's put this information to use by programming the robot to drive forward.

The IntelliBrain-Bot educational robot uses servo port 1 for the left servo and servo port 2 for the right servo. In order to control the power applied to the motors your program will need to get the Servo objects for these ports and wrap them in ContinuousRotationServo wrappers, which give them Motor interfaces. This will enable your program to control the motors using the setPower method.

Completing the following steps to program the robot to drive forward:

1. Create a new project named "Maneuver."

2. Add import statements for the classes and interfaces your program will be using: IntelliBrain, Motor and ContinuousRotationServo.

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.robotics.Motor;
import com.ridgesoft.robotics.ContinuousRotationServo;
```

   Note: Import statements refer to pre-built classes your program "imports" from the class library, which is where all of the classes referred to in the API documentation exist. The import statements refer to the full name of each class, which includes the name of the package to which the class belongs. The package name appears immediately above the class name at the top of the API documentation for each class.

3. Add line at the beginning of the main method to output a message identifying your program.

```
System.out.println("Maneuver");
```

4. Create left and right motor objects by retrieving the objects for servo ports 1 and 2 from the IntelliBrain class and using them to create a ContinuousRotationServo object for each servo. The sense of direction of the right servo is opposite that of the left servo, so it must be reversed by specifying "true" for the reverse parameter. A range value of 14 works well for the servos used on the IntelliBrain-Bot educational robot.

```
Motor leftMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(1), false, 14);
Motor rightMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(2), true, 14);
```

5. Add two steps to use the setPower method to set both motors to maximum forward power.

```
leftMotor.setPower(Motor.MAX_FORWARD);
rightMotor.setPower(Motor.MAX_FORWARD);
```

6. Place a step at the end of the main method to wait for the STOP button to be pressed so your program will not terminate until the STOP button is pressed.

```
IntelliBrain.getStopButton().waitPressed();
```

7. Connect the serial cable to the robot and switch power on.

8. Build and download your program to the robot.

9. Disconnect the serial cable.

10. Place the robot on the floor in an open area.

11. Press the START button.

12. Follow the robot and pick it up or press the STOP button before it crashes into anything.

Programming the Robot to Rotate in Place

Referring back to Figure 3-1, all that you need to do to make the robot rotate in place instead of driving straight ahead is reverse the direction your program applies power to one of the wheels.

Complete the following steps to make the robot rotate in place:

1. Modify the setPower step for the left motor to apply reverse power.

```
leftMotor.setPower(Motor.MAX_REVERSE);
```

2. Connect the serial cable to the robot and switch power on.

3. Build and download your program to the robot.

4. Disconnect the serial cable.

5. Place the robot on the floor in an open area.

6. Press the START button.

   Observe the robot will rotate in place counter-clockwise.

7. Press the STOP button and switch power off.

8. Change your program such that the left motor is powered forward and the right motor is powered in reverse.

```
leftMotor.setPower(Motor.MAX_FORWARD);
rightMotor.setPower(Motor.MAX_REVERSE);
```

9. Download and run your program.

   Observe the robot will rotate in place clockwise.

Programming the Robot to Drive in a Circle

Your first guess might be that it has got to be very difficult to program the robot to drive in a circle. However, it is surprisingly easy! All that you have to do is apply more power to one motor than to the other. This will make one wheel rotate faster than the other and the robot with follow an arced path and drive in circles.

Program the robot to drive in a circle by completing the following steps:

1. Change your program to provide full power to the left motor and half power to the right motor. Consulting the API documentation for the Motor interface you will see that the value for full power forward is 16, which is the value of the constant Motor.MAX_FORWARD you've been using thus far.

```
leftMotor.setPower(16);
rightMotor.setPower(8);
```

2. Download and run your program.

   Observe the robot will drive in a circle.

Combining Simple Maneuvers

You can program the robot to perform more advanced maneuvers by executing combinations of simple maneuvers. For example, you can program it to drive in a square by programming it to drive forward briefly then rotate 90 degrees then drive forward again, repeating the sequence four times in a row.

You've programmed the robot to drive forward forever and rotate in place forever. In order to drive in a square you'll have to limit how long it does each of these simple maneuvers. You can do this by having your program issue commands to the motor then

sleep while the motors run. When your program wakes up it will issue the next command and sleep until it is done.
Modify the Maneuver program to program the robot to drive forward for 2 seconds, as follows:

1. Change the setPower commands back to powering both motors forward at maximum power.

2. Browse to and review the documentation for the sleep method of the Thread class in the API documentation. Note this method allows your program to sleep for the number of milliseconds it specifies. The method also throws an InterruptedException. We don't need to be concerned with this exception for this exercise, but the Java compiler will insist that we provide code to catch it.

3. Add a call to the sleep method of the Thread class and an associated try-catch block immediately following the set power commands to instruct your program to sleep for 2000 milliseconds (2 seconds).

   ```
   try {
       Thread.sleep(2000);
   }
   catch (InterruptedException e) {}
   ```

   The sleep method of the Thread class causes the computer to stop executing (sleep) for the specified number of milliseconds. We have placed this call in a try-catch block because the sleep method may throw an InterruptedException. We are not concerned about the possible exception, so we have left the catch block empty. The loop will simply continue executing if the sleep call is ever interrupted.

4. Delete the line which causes your program to wait for the STOP button to be pressed.

   This line is no longer needed because we want the robot to stop once your program wakes up from its sleep.

5. Download and run your program.

   Observe the robot will drive forward for 2 seconds and stop.

In a similar fashion, you can program the robot to rotate 90 degrees by reversing the power to one of the motors and adjusting the time such that it stops rotating after approximately 90 degrees. You could then combine the go forward and the rotate 90 degrees steps to program the robot to complete one side of a square, turning at the end. Then by cutting and pasting, you could duplicate the code four times, programming the robot to complete a square. However, there's a better way to do this.

Creating Methods

Methods provide a way to take a set of program steps and combine them into a larger step, so you can reuse subsets of your program without copying and pasting. To see how this works, let's create "goForward" and "rotate90" methods:

1. Convert the variables leftMotor and rightMotor into member variables so they can be used by methods other than main. As they are now, they are local variables, which can only be used in the method they are defined in. Do this by adding two lines declaring these variables between the declaration of the class and the main method.

```
public class Maneuver {
    private static Motor leftMotor;
    private static Motor rightMotor;

    public static void main(String args[]) {
```

2. Delete the word "Motor" from in front of the lines that initialize these variables with ContinuousRotationServo objects.

```
leftMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(1), false, 14);
rightMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(2), true, 14);
```

   This was used to declare these variables local to the main method, which means they could only be used in the main method. Now they are member variables which can be used in any method in the Maneuver class.

3. Click the build button (wrench icon) to check that you've done this correctly and your program still compiles and links.

4. Split the main method into two methods by inserting a closing brace after the statement which initializes the right motor and declaring the goForward method such that it contains the remainder of the statements.

```
    rightMotor =
        new ContinuousRotationServo(IntelliBrain.getServo(2), true, 14);
}

public static void goForward() {
    leftMotor.setPower(Motor.MAX_FORWARD);
```

5. Add a step to make goForward the final step of the main method.

```
    rightMotor =
        new ContinuousRotationServo(IntelliBrain.getServo(2), true, 14);

    goForward();
}
```

6. Download your program to the robot and run it.

   Observe the robot will behave just as it did before you added the goForward method.

7. Using the mouse, select the entire goForward method from its declaration through the closing brace, but do not select the final closing brace which signifies the end of the class.

8. Enter Ctrl-C or use Edit->Copy to copy the method.

9. Move the cursor down to the last line of the class, just to the left of the closing brace and enter Ctrl-V or use Edit->Paste to paste a second copy of the goForward method.

10. Change the name of the copy of the "goForward" method to "rotate90."

```
public static void rotate90() {
```

11. Change the power settings to 8 and -8 for the left and right motors, respectively.

```
leftMotor.setPower(8);
rightMotor.setPower(-8);
```

    Using a power level less than the maximum will cause the robot to rotate more slowly, making it easier to tune the angle of rotation.

12. Change the sleep time in the rotate90 method to 600.

13. Create a stop method following the rotate90 method, which will stop the motors.

```
public static void stop() {
    leftMotor.stop();
    rightMotor.stop();
}
```

14. Add two steps to make rotate90 and stop the final steps of the main method, right after goForward.

```
    goForward();
    rotate90();
    stop();
}
```

15. Download and run your program.

    The robot will drive forward, rotate approximately 90 degrees and stop. If it

rotates too much, reduce the sleep time, then download and run your program again. If it rotates too little, increase the sleep time, then download and run your program again. Once you have it rotating about 90 degrees proceed to the next step.

16. Copy the goForward and rotate90 steps at the end of the main method and duplicate them three more times.

```
goForward();
rotate90();
goForward();
rotate90();
goForward();
rotate90();
goForward();
rotate90();
stop();
```

17. Download and run your program.

    The robot will drive approximately in a square. Adjust the sleep time in the rotate90 method to fine tune the robot to drive in as close to a perfect square as possible.

By creating methods to go forward, rotate 90 degrees and stop, you avoided creating a lot of duplicate code. This makes your program smaller and easier to maintain. If you were to make slight adjustments to any of these methods, you would only need to make changes in one copy of the procedure, rather than in four. This is a big improvement, but there's still more room to improve your program by adding a loop to eliminate more duplication.

Looping

Most programming languages include a looping construct. This allows your program to repeat a set of steps a number of times without requiring the steps to be duplicated, as we did previously. Java provides three types of loops, while loops, do-while loops and for loops.

Suppose you would like to program the robot to drive in a square until you tell it to stop. Without the concept of looping – repeating the same set of steps over and over – you would be forced to cut and paste the same set of steps, repeating enough times such that your program would not terminate before you lost patience watching the robot drive in a square. With looping, this is easy to accomplish with very little Java code.

Do the following to program your robot to drive in a square indefinitely:

1. Delete three of the four repetitions of the goForward and rotate90 steps.

2. Delete the stop step.

3. Add a while loop around the remaining goForward and rotate90 steps.

```
while (true) {
    goForward();
    rotate90();
}
```

4. Download and run your program.

   The robot will now drive in a square until you stop it or its batteries drain.

   You will most likely notice the corners of the square drift as the robot continues to drive around the square. This occurs because the robot does not rotate exactly 90 degrees on each turn. If it overshoots or undershoots by even a small amount on each turn, the error will accumulate and become very noticeable. You can reduce this by fine tuning the sleep time in the rotate90 method, but you will not be able to eliminate it.

   Coping with imperfections and uncertainty in the real world is one of the greatest challenges of robotics. Fortunately, robotics researchers are making great progress developing techniques for robots to cope with uncertainty.

Conditionals and Variables

You have now programmed two extremes, maneuvering once around a square and driving indefinitely around the square. What if you want the robot to drive just the four legs of the square and then stop?

In the previous exercise the while statement included the word “true” in parenthesis. This clause tells the computer under what condition it should continue executing the loop. The computer will check the condition prior to each execution of the step of the loop. If the condition is true, the computer will continue iterating the loop. If it is false, it will stop iterating the loop and start executing the instructions following the loop. In the previous exercise the condition was always true.

We can program the loop to use an integer variable to count the number of iterations and continue iterating until the count reaches a certain value. You can think of a variable like you do the memory function on a calculator. A variable remembers the last value assigned to it, just like a value you store in the memory register on your calculator. Your program can recall the last stored value at a later time, just like you can recall the last value you stored in your calculator’s memory whenever you need to.

Complete the following steps to add an integer variable to count the loop iterations and limit the loop to four iterations:

1. Declare the integer variable “i” and initialize it to zero on the line prior to the while statement.

   ```
   int i = 0;
   ```

   The int keyword tells the Java compiler to create an integer variable. Integer variables hold whole numbers with positive, zero or negative values.

2. Add a statement at the end of the loop to increment the variable i at the end of each iteration.

   ```
   i++;
   ```

   Note: The ++ operator increments the associated variable. A similar operator, --, decrements the associated variable.

3. Modify the condition clause of the loop such that it will only continue if the count is less than four.

   ```
   while (i < 4) {
   ```

4. Add a statement to call the stop method after the loop.

   ```
   int i = 0;
   while (i < 4) {
       goForward();
       rotate90();
       i++;
   }
   stop();
   ```

5. Download and run your program.

   The robot will now drive once around the square.

do-while and for Loops

The Java language provides do-while loops and for loops. The do-while loop checks the condition at the end of loop rather than the beginning, which is useful if statements in the loop always need to execute at least one time. The while loop above can be expressed as the following do-while loop:

```
int i = 0;
do {
    goForward();
    rotate90();
    i++;
} while (i < 4);
```

```
stop();
```

A for loop combines the initialization, conditional clause and post-iteration operation into a single statement as follows:

```
for (int i = 0; i < 4; i++) {
    goForward();
    rotate90();
}
stop();
```

Relational Operators

In the examples thus far we have used only the less than (<) relational operator. The Java language provides other relational operators. These are listed in Table 3-1.

Table 3-1 - Relational Operators

Operator	Description	Example
<	less than	`i < 4`
<=	less than or equal to	`i <= 4`
>	greater than	`i > 4`
>=	greater than or equal to	`i >= 4`
==	equal to	`i == 4`
!=	not equal to	`i != 4`

Summary

The key to maneuvering a differential-drive robot is individually controlling the direction and speed of the two wheels. By programming the robot to turn both wheels in the same direction, it will go forward. By programming the robot to turn its wheels in opposite directions, it will rotate in place. You can program the robot to perform other maneuvers, such as driving in an arc, by applying different amounts of power to each motor.

You can program the robot to perform more sophisticated maneuvers by performing timed sequences of basic maneuvers. By programming your robot to perform four repetitions of moving straight and turning 90 degrees it will drive in a square pattern.

As you experiment with the robot, you will observe that using time as the basis for navigation has its limitations. As the batteries drain, the behavior of the robot will change. Also, the robot's performance will vary depending on the surface it operates on. The difference in friction of different surfaces will have a significant effect on how the robot performs.

Unfortunately, as your program is currently implemented, it provides no mechanism for the robot to account for variations in conditions such as battery charge and friction. The robot's control system is "open loop" because it lacks "feedback" to account for variations in conditions that affect its performance.

You can improve maneuvering consistency and accuracy by adding sensors to provide feedback to your program. One way you can do this is by adding wheel encoders to sense the positions of the wheels.

This chapter also demonstrated how to use the class library API documentation and the *IntelliBrain 2 User Guide* to learn how to program the robot. By using the documentation you can learn how to use features you have not previously used.

Finally, you learned about a few more features of the Java language, methods, variables and loops. Methods allow you to combine a sequence of program steps into a single high-level step. Instead of copying and pasting the steps to go forward and rotate 90 degrees, you created methods to execute the detailed steps to do the higher level operations of going forward, rotating and stopping. By using a loop, you were able to construct your program to repeat a sequence of steps a number of times without duplicating statements. Loops included conditional statements which control when the computer should continue iterating the loop. When the condition is false the loop will terminate and the remainder of your program will execute.

Exercises

1. Browse to the API documentation for the IntelliBrain class and review it. List three methods of this class and describe what they do.
2. Locate the discussion of using servo ports in the *IntelliBrain 2 User Guide*. How many servo ports does the IntelliBrain 2 robotics controller provide? Where on the circuit board are they located? Which servo ports do the left and right motors on the IntelliBrain-Bot connect to?
3. Browse to the documentation on the Motor interface in the class library API documentation. What methods does this interface offer? What is the range of the power parameter used by the setPower method?
4. Identify each class and method your Maneuver program references. Browse to the API documentation for each of these classes and methods and write a short description of each based on the information from the documentation.
5. Describe how a differential drive robot can accomplish the following maneuvers: drive forward, rotate counter-clockwise in place and drive in an arc to the right.
6. Program the robot to drive straight ahead for exactly 12 inches. Using the same program, run the robot on different surfaces such as tile or carpet. Record how far it goes on each surface. Why does it not go the same distance on all surfaces?
7. Program the robot to rotate exactly 180 degrees. Using the same program, run the robot on different surfaces. Record how far it rotates on each surface. Why does it not rotate the same amount on all surfaces?
8. Program the robot to navigate in a triangle or some other shape you choose. Modify your program to use while, do-while and for loops to accomplish the same maneuver.

9. Program the robot to arc to the left or right as it moves forward by applying a different amount of power to each wheel. What happens when you increase or decrease the difference in power?
10. Program the robot to navigate a particular shape. Change the batteries to a set of batteries that has more or less charge remaining. How does this affect the navigation?

CHAPTER 4

Interacting with the IntelliBrain-Bot

"Danger, Will Robinson!"

Whether it's the nameless robot from the 1960s television show, Lost in Space, who shouted warnings and flailed his arms when his human companion, Will Robinson, was in danger, or one of the many other science fiction robots we have fantasized about, we often imagine robots as machines that interact freely with humans.

In this chapter you will learn how to program the features of the IntelliBrain™-Bot educational robot that enable it to interact with people.

Using Text Output

On way for the IntelliBrain-Bot educational robot to interact with humans is by displaying textual messages on its Liquid Crystal Display (LCD) screen. The LCD screen provides space to display two sixteen character lines of text.

Before jumping into writing a program to use the display, first review the documentation for the classes you will need to be familiar with to output text to the LCD screen.

1. Start RoboJDE™, click the API documentation button.
2. Navigate to the documentation for the IntelliBrain class.
3. Navigate to and review documentation for the getLcdDisplay method.
4. Click on the link to documentation for the Display class.
5. Review the documentation for the Display class, taking note of the methods which are available to display text.

Now, let's use what you have learned from the documentation by creating a simple program to display the name of your program on the first line of the LCD screen. You will begin by creating a new project for your program. You will build on this program as you work your way through this chapter.

1. Using the File->New Project menu item, create a new project named “Interact.”

2. Add import statements at the beginning of your program for the two library classes your program will use, IntelliBrain and the Display.

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.io.Display;
```

3. Declare a variable to refer to the display object for the LCD display.

```
private static Display display;
```

4. Insert a line into the main method to retrieve the Display object from the IntelliBrain class and save a reference to it in a variable named display.

```
display = IntelliBrain.getLcdDisplay();
```

5. Add a line to display the name of your program on the first line of the LCD screen.

```
display.print(0, "Interact");
```

   The LCD screen has two lines. The first line is numbered 0. The second line is numbered 1. (It is very common in computer programs to start with zero when numbering elements in a list. This stems from the fact that computer hardware accesses elements in a list by their offset from the beginning of the list. The first element is offset by 0, the second element is offset by 1, and so on.)

   The entire program should appear as follows:

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.io.Display;

public class Interact {
      private static Display display;

    public static void main(String args[]) {
        display = IntelliBrain.getLcdDisplay();
        display.print(0, "Interact");
    }
}
```

6. Build, load and test your program.

   Your program name will quickly display its name on the first line of the LCD screen and then exit.

Using LEDs

One very simple way for robots to communicate with humans is by using Light Emitting Diodes (LEDs). You are undoubtedly familiar with these, as they are used on almost every computer system and electronic gadget you will ever encounter.

One of the most common ways to use an LED is to indicate when power is on. When the power is on, the LED is illuminated. When the power is off, the LED is not illuminated. The IntelliBrain 2 robotics controller has a power LED. It is the green LED to the right of the power switch. Switch the power on and off and observe this LED goes on and off.

The power LED is hardwired into the circuit of the IntelliBrain controller. Your programs cannot control this LED. However, the six other LEDs to the right of it can be controlled by your programs.

The first two LEDs to the right of the power LED are the status LED and the fault LED, respectively. These LEDs are only partially controllable by your programs. They are wired to the START and STOP buttons. The status LED, which is green, will always illuminate when you press the START button. The fault LED, which is red, will always illuminate when you press the STOP button. Your program can control the state of these LEDs when the buttons are not pressed. However, if the RoboJDE virtual machine encounters an unexpected problem, it will illuminate the fault LED.

The next four LEDs are user LEDs, which are fully under the control of your programs. We will extend your program to use these LEDs so you will become familiar with how LEDs can be used to enable the robot to interact with humans.

In the previous chapter you wrote a program that enabled the robot to perform various maneuvers. Although you could see the robot performing particular maneuvers, there was no indication of how your program was commanding the individual motors. We will extend your program to use the LEDs to provide more visibility into the commands your program gives the motors as it maneuvers the robot. Before we discuss programming the LEDs, let's first extend your program by adding the ability to maneuver in a square pattern. Once you have this working, we will discuss programming the LEDs.

1. Add import statements to your program for Motor and ContinuousRotationServo.

```
import com.ridgesoft.robotics.Motor;
import com.ridgesoft.robotics.ContinuousRotationServo;
```

2. Define two variables to reference the left and right motor objects.

```
private static Motor leftMotor;
private static Motor rightMotor;
```

3. Instantiate the motor objects in the main method.

```
leftMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(1), false, 14);
rightMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(2), true, 14);
```

4. Create a method that allows the caller to specify the power to apply to each motor and the length of time to hold that power.

```
public static void go(int leftPower, int rightPower, int milliseconds) {
    leftMotor.setPower(leftPower);
    rightMotor.setPower(rightPower);

    try {
        Thread.sleep(milliseconds);
    }
    catch (InterruptedException e) {}
}
```

   This method defines three variables – also referred to as the methods arguments – which must be passed to the method when it is called. They are leftPower, rightPower and milliseconds.

5. Create a stop method to turn off both motors.

```
public static void stop() {
    leftMotor.stop();
    rightMotor.stop();
}
```

6. Create a method that uses the go and stop methods to maneuver in a square pattern of a specified size.

```
public static void maneuverSquare(int size) {
    for (int i = 0; i < 4; i++) {
        go(16, 16, size);
        go(8, -8, 600);
    }
    stop();
}
```

   For convenience, the size is in units of milliseconds, rather than units of distance.

7. Add a statement to call the maneuverSquare method at the end of the main method.

```
maneuverSquare(3000);
```

8. Build, load and test your program.

   The robot will maneuver in a square pattern.

Now that you have the base program working, let's extend it to make use of several LEDs. We'll use four LEDs to indicate how the motors are being powered while the robot maneuvers. We'll use the four user LEDs on the IntelliBrain 2 robotics controller. These are the four right most LEDs, above the STOP button. The odd numbered LEDs are green. The even numbered LEDs are red. We'll use the left two LEDs to indicate the status of the left motor and the right two LEDs to indicate the statues of the right motor. We will indicate the direction of the motor by illuminating the green LED when your program applies forward power and illuminating the red LED when your program applies reverse power. We will turn the LEDs off when no power is applied to the motor.

1. Review the API documentation for the getUserLed method of the IntelliBrain class. Also review the API documentation for the LED interface.

2. Add an import statement for the LED interface.

```
import com.ridgesoft.io.LED;
```

3. Declare variables to refer to the LED objects.

```
private static LED leftFwdLED;
private static LED leftRevLED;
private static LED rightFwdLED;
private static LED rightRevLED;
```

4. Add statements to the main method to obtain the LED objects from the IntelliBrain class.

```
leftFwdLED = IntelliBrain.getUserLed(1);
leftRevLED = IntelliBrain.getUserLed(2);
rightFwdLED = IntelliBrain.getUserLed(3);
rightRevLED = IntelliBrain.getUserLed(4);
```

We must now extend your program to turn the LEDs on and off appropriately whenever it adjusts the power applied to either motor. Whenever your program sets the motor power to a value greater than 0, it will illuminate the green LED and turn off the red LED. Whenever your program sets the motor power to a value less than 0, it will illuminate the red LED and turn off the green LED. Whenever your program sets the motor power to 0 or stops the motor, it will turn off both LEDs. We will use if statements to enable your program to accomplish this.

if Statements

The Java language enables your program to conditionally execute statements using an if statement. This statement has the form:

```
if (condition) {
    conditionally executed statements
        :
}
```

The statements within the parenthesis are a block of code that is only executed if the condition is true. Using the following statements, your program will turn the green LED on and the red LED off when the power to the left motor is greater than 0:

```
if (leftPower > 0) {
    leftFwdLED.on();
    leftRevLED.off();
}
```

This takes care of the case when forward power is applied to the motor. It doesn't handle the reverse and stop cases. The "else" clause allows us to handle these cases.

The Java language provides for chaining conditional clauses together using the else keyword, as follows:

```
if (condition1) {
    conditionally executed statements
        :
}
else if (condition2) {
    conditionally executed statements
        :
}
else {
    conditionally executed statements
        :
}
```

When a program runs, the if conditions will be checked in order. The block of code within the first clause whose condition evaluates to true will be executed. The other code blocks will be skipped. If none of the conditions are true, the else code block, at the end, will be executed. You can chain together as many else if clauses as you need. You can also leave out the final else clause if it isn't necessary.

By using else clauses, we can handle all of the statements needed to control the left motor's LEDs, as follows:

```
if (leftPower > 0) {
    leftFwdLED.on();
    leftRevLED.off();
}
else if (leftPower < 0) {
    leftFwdLED.off();
    leftRevLED.on();
}
else {
    leftFwdLED.off();
    leftRevLED.off();
}
```

The right motor's LEDs can be handled similarly.

Let's add if statements to your program to control the LEDs.

5. Add the code above to the go method, right after the statements that set the motor power.

6. Copy the statements you entered in the previous step and paste them right below the first copy. Change all instances of "left" in the pasted copy to "right."

7. Add statements to the stop method to turn the LEDs off.

```
leftFwdLED.off();
leftRevLED.off();
rightFwdLED.off();
rightRevLED.off();
```

8. Build, load and test your program.

   Observe the LEDs as the robot drives. As the robot goes straight ahead the two green LEDs are illuminated and the two red LEDs are off. When the robot turns, you observe the right motor's green LED turns off and the red LED illuminates, because your program has reversed power to the right motor.

Using the Thumbwheel

The IntelliBrain 2 robotics controller's thumbwheel works similar to the volume control knob on a car radio. In this section, we will extend your program to use the thumbwheel to control the size of the square pattern the robot maneuvers. We will also use the thumbwheel to control other functions we add to your program later in this chapter.

The thumbwheel functions as a variable analog input. When your program samples (reads) the thumbwheel input it will obtain an integer value between 0 and 1023. The reading will vary depending on the position of the wheel. When the wheel is turned all the way counterclockwise, the reading will be 0. When the wheel is turned all the way clockwise, the reading will be 1023. When the wheel is in an intermediate position, the reading will be a value between 0 and 1023.

Complete the follow steps to extend your program such that you can use the thumbwheel to vary the size of the square the robot maneuvers.

1. Review the API documentation for the getThumbWheel method of the IntelliBrain class and the API documentation for the AnalogInput interface.

2. Add an import statement for the AnalogInput interface.

```
import com.ridgesoft.robotics.AnalogInput;
```

3. Define a variable to refer to the thumbwheel object.

```
private static AnalogInput thumbwheel;
```

4. Add a statement to the main method to obtain a reference to the thumbwheel input object from the IntelliBrain class.

```
thumbwheel = IntelliBrain.getThumbWheel();
```

5. Add a statement to the main method to sample and display the thumbwheel reading on the second line of the LCD screen. Add this statement just prior to the point where you call the maneuverSquare method.

```
display.print(1, "Thumbwheel: " + thumbwheel.sample());
```

6. Modify the input parameter of the maneuverSquare method to use the thumbwheel to control the size of the square.

```
maneuverSquare(thumbwheel.sample() * 5);
```

   Recall that the range of the thumbwheel is 0 to 1023. This statement multiplies the thumbwheel reading by five and passes the result to the maneuverSquare method. The size of the square is specified as the number of milliseconds your program powers the robot straight ahead along each side of the square.

7. Build, load and test your program.

   Run your program several times, setting the thumbwheel to a different position prior to each run. Observe that the thumbwheel reading is displayed on the second line of the LCD screen. The size of the square the robot maneuvers depends on the position of the thumbwheel.

Arithmetic Operations

In the previous section, we used the multiplication operator (*) to scale the thumbwheel reading to a value suitable for passing to the maneuverSquare method. This is one of many arithmetic operations the Java language supports. Table 4-1 lists all of the arithmetic operations that are available to Java programs.

Assignment Operator

The assignment operator (=) assigns the variable on the left hand side of the operator, the value of the expression on the right hand side of the operator. For example, the statement:

```
a = 2;
```

sets the value of the left hand side variable, a, to 2. The expression on the right hand side may be more complex, such as:

```
a = b + c;
```

In this case, a will be assigned the sum of the values of the variables b and c. For example, if values of variables b and c are 9 and 7, the value 16 will be assigned to the variable a.

Table 4-1 - Arithmetic Operators

Operator	Description	Example
=	assignment	`b = 3;`
+	add (integer)	`a = b + 3;`
	concatenate (String)	`s = "Value: " + i;`
-	subtract	`a = b – 3;`
*	multiply	`a = b * 5;`
/	divide	`a = b / 5;`
%	remainder	`a = b % 5;`
++	pre-increment (increment before use)	`a = ++b + 2;`
	post-increment (increment after use)	`a = b++ + 2;`
--	pre-decrement (decrement prior to use)	`a = --b + 2;`
	post-decrement (decrement after use)	`a = b-- + 2;`
+=	add and assign	`a += 2;`
-=	subtract and assign	`a -= 2;`
*=	multiply and assign	`a *= 5;`
/=	divide and assign	`a /= 2;`
%=	remainder and assign	`a %= 5;`

Arithmetic Operators

The Java language supports the four common arithmetic operations you are familiar with from using a calculator: add (+), subtract (-), multiply (*) and divide (/). These operators function as you would expect when solving algebraic equations on your calculator. As with most programming languages, multiply and divide are denoted using the symbols * and / instead of the standard math symbols. The standard symbols for multiplication and division don't exist on a computer keyboard, therefore, * and / are used in their place. The following code snippet provides examples of the arithmetic operations. The comments to the right list the values of each variable will have after that statement is executed. Two backslashes (//) denote the remainder of the line is a comment intended for the reader. The Java compiler ignores the slashes and the remainder of the line, which allows you to place comments (notes) in your programs.

```
                    //    a    b    c
                    //---------------
int a = 1;          //    1    -    -
int b = 2;          //    1    2    -
int c = 3;          //    1    2    3
a = a + c;          //    4    2    3
```

```
a = a – 2;        //    2    2    3
a = a – c;        //   -1    2    3
a = c * 10;       //   30    2    3
a = a + b;        //   32    2    3
b = a / b;        //   32   16    3
b = a % c;        //   32    2    3
```

In addition to summing two numeric values, the + operator can also be used to concatenate strings. For example:

```
int a = 32;
System.out.println("The value of a is:  " + a);
```

will print the following output:

```
The value of a is:  32
```

Whenever the operand to the left of the + operator is a string, the operand to the right of the + sign is converted to its string equivalent. The two strings are then concatenated to form a single string. In the case above, the value of the variable a is converted from an integer to a string ("32") and appended to the string "The value of a is: "

Precedence

As with standard algebraic notation, arithmetic operations have precedence in the Java. For example, given the expression:

```
a = 3 + b * c;
```

b and c will be multiplied prior to adding 3. Multiplication takes precedence over addition. Therefore, the Java compiler ensures the multiplication operation will be done first. The precedence of arithmetic operations in Java is given in Table 4-2. Operations higher in the table have higher precedence. The Java compiler will ensure operations higher in the table are computed prior to operations lower in the table. Operations in the same row in the table have the same precedence. Equal precedence operations are evaluated in the left to right order they appear in your program.

Table 4-2 - Arithmetic Operator Precedence (greatest to least)

Operators	Description
++, --	pre-increment/decrement
++, --	post-increment/decrement
*, /, %	multiplication, division, remainder
+, -	addition, subtraction
=, *=, /=, =+, -=	assignment

Similar to most calculators, you can supply parentheses to specify the order of operations, overriding the default precedence. For example,

```
a = (3 + b) * c;
```

will result in the sum of 3 and b being multiplied by c. If b is 2 and c is 3, the value 15 will be assigned to a.

Frequently, you will find you programs are easier to understand if you include parentheses even when they are not needed. Parentheses make it easy to understand the order of operations without having to recall the precedence rules.

Using Push Buttons

The thumbwheel enhancement we added in the previous section has a significant shortcoming. It doesn't provide a way for you to dial in the setting prior to the start of the maneuver. We can address this by enhancing your program to delay the beginning of the maneuver until a second press of the START button. During this period your program can loop displaying the thumbwheel reading. This will give you a chance to dial in the exact setting you desire.

Complete the following steps to add this feature:

1. Review the API documentation for the getStartButton method of the IntelliBrain class and the API documentation for the PushButton interface.

2. Add an import statement for the PushButton interface.

   ```
   import com.ridgesoft.robotics.PushButton;
   ```

3. Declare a variable to refer to the START button object.

   ```
   private static PushButton startButton;
   ```

4. Add a statement to the main method to obtain the START button object from the IntelliBrain class.

   ```
   startButton = IntelliBrain.getStartButton();
   ```

5. Add a while loop around the thumbwheel print statement. The loop must continue iterating until the second press of the START button.

   ```
   startButton.waitReleased();
   while (!startButton.isPressed()) {
       display.print(1, "Thumbwheel: " + thumbwheel.sample());
   }
   ```

   Preceding the loop with a statement to wait for the button to be released ensures your program will waits for the button to be released after the initial press. By including this line, your program will wait until the second START button press

before starting the maneuver.

The exclamation point (!) in the while statement is the logical NOT operator. It causes the true or false value returned by the call to the isPressed method to be inverted. If you mentally insert the word, not, whenever you see this operator, you will understand what it does. In this case, read the while loop as follows, "While the start button is not pressed, print the reading of the thumbwheel."

6. Build, load and test your program.

   Your program will now allow you to adjust the thumbwheel setting. Once you have dialed in the setting you want, press the START button. The robot will maneuver a square size you specified. Run your program several times using a different thumbwheel setting each time.

Logical Operators and Boolean Variables

In addition to integers, Java supports Boolean variables, which have only two possible values, true or false. The isPressed method defined by the PushButton interface returns a Boolean value. The value will be true if the button is pressed and false if it is not pressed. The "boolean" keyword allows you to declare Boolean variables in your programs. For example,

```
boolean pressed = false;
```

Table 4-3 - Logical Operators

Operator	Description	Example
!	NOT	!b;
&&	AND	a = b && c;
||	OR	a = b || c;

Table 4-3 lists the logical operators that operate on Boolean values. The following code snippet illustrates the use of these Boolean operators.

```
                          //  a      b     c
                          //------------------
boolean a = true;         // true    -     -
boolean b = false;        // true  false   -
boolean c = true;         // true  false true
a = !c;                   // false false true
a = b && c;               // false false true
a = b || c;               // true  false true
b = a && c;               // true  true  true
a = !b && !c;             // false true  true
```

The relational operators listed in Table 3-1 operate on numeric values, but generate a Boolean result. For example,

```
boolean moving = power != 0;
```

```
boolean goingForward = power > 0;
boolean goingBackward = power < 0;
```

Teaching the Robot New Tricks

As with a pet, your robot will be more fun if it can do more than just one trick. In addition to its first trick, driving in a square, let's teach the robot a second trick, dancing.

Rather than programming the robot to perform a highly choreographed dance, we'll keep it simple. We'll teach the robot to "dance" by moving around randomly on the floor. We can do this by adding a new method to your program that uses the Random class to generate random numbers. Your program will use to randomly vary the power it applies to the motors.

Extend your program as follows:

1. Review the API documentation for the Random class.

   The Random class has a method, nextInt, which returns a random number from the full range of the int data type (-4,294,967,295 to 4,294,967,296). Alternatively, if you call nextInt with an integer argument it will return a random value between 0 and the value of the argument. Neither of these options provides exactly what we need, which is a random value between -16 and 16, the range of motor power settings. By using the remainder operator (%), your program can convert a random integer to the correct range. This operator computes the remainder of a division operation. Your program can obtain a random number between -16 and 16 by using the remainder operator to compute the remainder of a division by 17.

   ```
   int leftPower = random.nextInt() % 17;
   int rightPower = random.nextInt() % 17;
   ```

   You can further randomize the dance by programming the robot to do each step of the dance for a random period of time. The following statement chooses a random number of milliseconds between 100 and 499.

   ```
   int time = random.nextInt(400) + 100;
   ```

2. Add the following method to your program.

   ```
   public static void dance(int seed) {
       Random random = new Random(seed);

       while (true) {
           int leftPower = random.nextInt() % 17;
           int rightPower = random.nextInt() % 17;
           int time = random.nextInt(400) + 100;
           go(leftPower, rightPower, time);
       }
   ```

```
}
```

This method loops forever performing dance steps by randomly varying the power applied to each motor and the time spent doing each step. It requires a seed which randomizes the random number generator. Otherwise, the dance would consist of the same set of pseudo random steps each time.

3. Add an import statement for the Random class.

```
import java.util.Random;
```

4. Replace the call to maneuverSquare method with a call to dance.

```
dance(thumbwheel.sample());
```

5. Build, load and test your program.

   Observe the robot will move around randomly, but will not wander far from the spot where it started.

Now that we've taught the robot a second trick, let's further extend your program to allow you to select the trick you would like the robot to perform.

We can do this by using the STOP button to cycle through the list of tricks and the START button to perform the trick you select.

1. Declare a variable to refer to the STOP button object.

```
private static PushButton stopButton;
```

2. Initialize the stopButton variable by adding a call to the getStopButton method. Use the setTerminateOnStop method to configure the STOP button so it does not terminate your program.

```
stopButton = IntelliBrain.getStopButton();
IntelliBrain.setTerminateOnStop(false);
```

3. Declare a variable to keep track of which trick is currently proposed. Initialize it to 1. Place this statement just prior to the startButton loop in the main method.

```
int trick = 1;
```

4. Add statements in the startButton loop to cycle to the next trick each time the STOP button is pressed.

```
if (stopButton.isPressed()) {
    stopButton.waitReleased();
    trick++;
    if (trick > 2)
        trick = 1;
}
```

There are only two tricks. If the value of trick is 2 when the STOP button is pressed, the if statement causes it to cycle back to trick number 1.

5. Insert statements in the startButton loop to display the proposed trick.

```
if (trick == 1) {
    display.print(0, "Maneuver Square");
}
else {
    display.print(0, "Dance");
}
```

6. Reconfigure the STOP button such that it will terminate your program once the trick has been selected. Insert the following statement after the end of the loop.

```
IntelliBrain.setTerminateOnStop(true);
```

This will allow you to stop the robot when it is performing a trick. Without doing this, you would have to switch the power off.

7. Replace the call to the dance method with statements that will call the method for the selected trick.

```
if (trick == 1) {
    maneuverSquare(thumbwheel.sample() * 5);
}
else {
    dance(thumbwheel.sample());
}
```

8. Build, load and test your program.

You will now be able to use the STOP button to select whether the robot should maneuver in square pattern or dance randomly.

Switch Statements

Up to this point we have used if statements to control which trick to execute. As we add more tricks, we will need to add another else-if clause for each new trick. This works adequately, but the Java language provides another type of statement, the switch statement, which is intended to handle this type of situation. Using a switch statement instead of an if statement results in a more efficient program. The if statement:

```
if (trick == 1) {
    // trick one statements
}
else {
    // trick two statements
}
```

can be replaced by the equivalent, but more efficient, switch statement:

```
switch (trick) {
case 1:
    // trick one statements
    break;
case 2:
    // trick two statements
    break;
}
```

Switch statements are more efficient than equivalent if statements. With the switch statements the computer doesn't need to check multiple conditions each time the statement is executed. Instead, the Java compiler builds a table that enables the computer to look up the appropriate case based on the value of the switch variable. In our case the integer variable trick is the switch variable. When there are a lot of cases, looking up the correct case is much more efficient than checking each case to find the code block to execute.

Modify your program to use switch statements instead of if statements, as follows:

1. Replace the if statement in the startButton loop with an equivalent switch statement.

```
switch (trick) {
case 1:
    display.print(0, "Maneuver Square");
    break;
case 2:
    display.print(0, "Dance");
    break;
}
```

2. Replace the if statement after the startButton loop with an equivalent switch statement.

```
switch (trick) {
case 1:
    maneuverSquare(thumbwheel.sample() * 5);
    break;
case 2:
    dance(thumbwheel.sample());
    break;
```

```
        }
```

3. Build, load and test your program.

   Your program will work the same as it did previously.

One other feature of the switch statement is the case named "default." If a default case is present, that case will be used when the value of the variable in the switch statement doesn't match any of the cases listed. You can eliminate the statement:

```
if (trick > 2)
    trick = 1;
```

by adding a default case to the first switch statement. Do this as follows:

1. Delete the if statement in the startButton loop.

2. Add a default case prior to case 1, but leave out the break statement at the end of the case.

   ```
   switch (trick) {
   default:
       trick = 1;
   case 1:
       // case one statements
   ```

   Whenever the value of trick is not 1, the computer will select the default case, which will reset the trick value to 1.

   The break statement causes the computer to break out of the switch statement and continue executing the statements following the switch statement. By leaving the break statement out of default case, execution will fall through into case 1. This is exactly what we want to happen when the trick variable exceeds the number of tricks. The trick number will cycle back to 1 and display the name of trick 1.

3. Build, load and test your program.

   Your program will function as it did previously.

Using the default case instead of an if statement makes your program slightly easier to maintain. As we add new tricks, you won't need to be concerned with modifying the limit check to cycle the trick number back to one. The default statement will adjust for this automatically each time you add a new case to the switch statement.

Using the Buzzer

The IntelliBrain robotics controller includes a buzzer, which provides audio interaction with users.

Let's extend your program to give audio feedback when we push buttons.

1. Review the API documentation for the getBuzzer method of the IntelliBrain class and the documentation for the Speaker class.

2. Add an import statement for the Speaker class.

```
import com.ridgesoft.io.Speaker;
```

3. Declare a variable to refer to the Speaker object for the buzzer.

```
private static Speaker buzzer;
```

4. Add a statement to the main method to obtain the buzzer object from the IntelliBrain class.

```
buzzer = IntelliBrain.getBuzzer();
```

5. Add calls to the buzzer object's beep method each time the START or STOP button is pressed. There are two places where these calls need to be added: 1) as the first statement within the stopButton.isPressed if statement, and 2) immediately after the startButton loop.

```
buzzer.beep();
```

6. Build, load and test your program.

   Now, the buzzer will beep when you press the START and STOP buttons.

Playing a Tune

For its next trick, we will teach the robot how to play a tune, Mary Had a Little Lamb, using the buzzer.

A musical tune is simply a sequence of notes played one after the other. Each note is a sound wave generated at a particular frequency and played for a certain period of time. The tune, Mary Had a Little Lamb, consists of whole notes, half notes and quarter notes. A half note is one half the duration of a whole note and a quarter note is one quarter the duration of a whole note. The duration of a whole note, determines the tempo, or speed, at which the tune plays. We will use the thumbwheel to control the tempo.

1. Define constants at the beginning of the Interact class to define the frequency of various musical notes.

```
public static final int C4 = 262;
public static final int C4_SHARP = 277;
public static final int D4 = 294;
public static final int D4_SHARP = 311;
public static final int E4 = 330;
public static final int F4 = 349;
public static final int F4_SHARP = 370;
public static final int G4 = 392;
public static final int G4_SHARP = 415;
public static final int A4 = 440;
public static final int A4_SHARP = 466;
public static final int B4 = 494;
public static final int C5 = 523;
public static final int C5_SHARP = 554;
public static final int D5 = 587;
public static final int D5_SHARP = 622;
public static final int E5 = 659;
public static final int F5 = 698;
public static final int F5_SHARP = 740;
public static final int G5 = 784;
public static final int G5_SHARP = 831;
public static final int A5 = 880;
public static final int A5_SHARP = 932;
public static final int B5 = 988;
```

   The frequencies of notes are available on many sites on the Internet. You find them using a search engine, such as Google. The notes listed above include the range of notes your program will use to play Mary Had a Little Lamb.

   The “final” keyword is used to define a constant value that cannot be changed while your program is running.

2. Create a method to play the tune, Mary Had a Little Lamb.

```
public static void playTune(int wholeNote) {
    int quarterNote = wholeNote / 4;
    int halfNote = wholeNote / 2;

    // Mary Had a Little Lamb
    buzzer.play(B4, quarterNote);
    buzzer.play(A4, quarterNote);
    buzzer.play(G4, quarterNote);
    buzzer.play(A4, quarterNote);

    buzzer.play(B4, quarterNote);
    buzzer.play(B4, quarterNote);
    buzzer.play(B4, halfNote);

    buzzer.play(A4, quarterNote);
    buzzer.play(A4, quarterNote);
```

```
    buzzer.play(A4, halfNote);

    buzzer.play(B4, quarterNote);
    buzzer.play(D5, quarterNote);
    buzzer.play(D5, halfNote);

    buzzer.play(B4, quarterNote);
    buzzer.play(A4, quarterNote);
    buzzer.play(G4, quarterNote);
    buzzer.play(A4, quarterNote);

    buzzer.play(B4, quarterNote);
    buzzer.play(B4, quarterNote);
    buzzer.play(B4, quarterNote);
    buzzer.play(B4, quarterNote);

    buzzer.play(A4, quarterNote);
    buzzer.play(A4, quarterNote);
    buzzer.play(B4, quarterNote);
    buzzer.play(A4, quarterNote);

    buzzer.play(G4, wholeNote);
}
```

You can find the sheet music for Mary Had a Little Lamb and many other tunes by searching on the Internet.

The argument, wholeNote, is the duration in milliseconds of a whole note.

3. Add a case to the first switch statement to display the name of this new trick.

```
case 3:
    display.print(0, "Play Tune");
    break;
```

4. Add a case to the second switch statement to call the playTune method, using the thumbwheel to allow the duration of a whole note to be adjusted between 1000 and 2023 milliseconds.

```
case 3:
    playTune(thumbwheel.sample() + 1000);
    break;
```

5. Build, load and test your program.

   Play the tune several times, varying the tempo using the thumbwheel.

Using a Universal Remote Control

Another way for the IntelliBrain-Bot educational robot to interact with humans is via a universal remote control. As with a television, the remote control allows human

interaction with the robot from a distance. The remote control does this by transmitting pulses of infrared light, which the IntelliBrain-Bot can sense using its infrared remote control receiver.

Understanding How an Infrared Remote Control Works

Universal remote controls use pulses of infrared light modulated at 38 kHz to transmit signals to the receiving device. Different vendors have different ways of converting keypad input into a stream of infrared light pulses. These are referred to as "protocols" in computer communications terminology. We will focus on the protocol used by Sony infrared remote controls.

Note: In order to complete the exercises in this section, you will need a Sony compatible universal remote control.

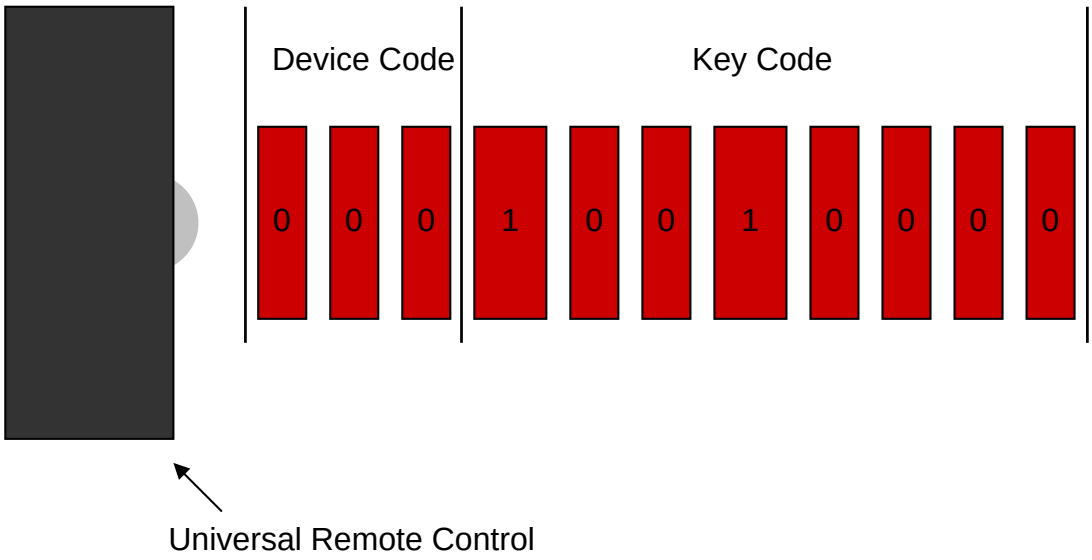


Figure 4-1 - Sony Infrared Remote Control Data Transmission

If you are familiar with how Morse Code is used to transmit letters of the alphabet over a telegraph link, then you already understand the basic concept of how key codes are transmitted by a Sony universal remote control. As depicted in Figure 4-1, each time you press a key on the remote control it transmits a series of short and long infrared light pulses. Each key on the remote control is represented by a unique sequence of short and long pulses of infrared light. This is similar to how each letter in the alphabet is represented by a unique sequence of long and short pulses in Morse Code. As indicated in Figure 4-1, a Sony remote control transmits the key code as eight infrared light pulses. It follows the key code with a device code, which consists of three pulses. The device code indicates which device the transmission is addressing, for example, the TV or VCR

Each infrared light pulse contains a single bit of data. A bit can have one of two values, 0 or 1. A short pulse represents a bit with value 0 and a long pulse represents a bit with

value 1. As indicated in Figure 4-1, the key code consists of eight bits of data and the device code consists of three bits of data. Figure 4-1 depicts the transmission of the channel-up key code to the TV. The channel-up key code is 10010000 in binary, or 144 decimal. Table 4-4 lists the 8-bit binary code and the equivalent decimal value for many of the keys typically found on a Sony television remote control. The television device code is 000 binary, or 0 decimal.

Table 4-4 – Sony Remote Control Key Codes

Key	Binary	Decimal	Key	Binary	Decimal
1	10000000	128	TV/Video	10100101	165
2	10000001	129	Right Arrow	10110011	179
3	10000010	130	Left Arrow	10110100	180
4	10000011	131	Display	10111010	186
5	10000100	132	Recall	10111011	187
6	10000101	133	Fast Forward	11011000	216
7	10000110	134	Rewind	11011001	217
8	10000111	135	Record	11011010	218
9	10001000	136	PIP	11011011	219
0	10001001	137	Pause	11011100	220
Enter	10001011	139	Stop	11011110	222
Channel Up	10010000	144	Play	11011111	223
Channel Down	10010001	145	Menu	11100000	224
Volume Up	10010010	146	OK	11100101	229
Volume Down	10010011	147	Up Arrow	11110100	244
Mute	10010100	148	Down Arrow	11110101	245

Receiving Input from the Remote Control

We will now extend your program to enable the remote control to be used as an alternative to the START and STOP buttons when selecting the trick to perform. Let's use the channel-down button on the remote control as an alternative to using the STOP button to select a trick. We will use the channel-up button to scroll backward through the list of tricks. We'll use the play button as an alternative to the START button to start the trick.

1. Review the API documentation for the getIrReceiver method of the IntelliBrain class, the documentation for the IrRemote interface and the documentation for the SonyIrRemote class.

2. Add import statements for the IrRemote interface and the SonyIrRemote class.

```
import com.ridgesoft.robotics.IrRemote;
import com.ridgesoft.robotics.sensors.SonyIrRemote;
```

3. Consult Table 4-4 and define constants in your program for the three buttons your program will use.

```
private static final int CHANNEL_UP = 144;
private static final int CHANNEL_DOWN = 145;
private static final int PLAY = 223;
```

4. Declare a variable to refer to the infrared remote control object.

```
private static IrRemote irRemote;
```

5. Add a statement to the main method to create an object to receive input from a Sony compatible infrared remote control via the IntelliBrain 2 robotics controller's remote control receiver.

```
irRemote = new SonyIrRemote(IntelliBrain.getIrReceiver());
```

   The SonyIrRemote class configures the remote control receiver such that it can receive infrared transmissions from a Sony compatible infrared remote control.

6. Declare a variable in the main method to hold the most recently received key code from the remote control. Initialize it to the value -1, to indicate no new key code data has been received from the remote control.

```
int keyCode = -1;
```

7. Modify the condition check of the while loop such that the it will continue iterating as long as the START button is not pressed and the play button key code is not received.

```
while (!startButton.isPressed() && (keyCode != PLAY)) {
```

   We have used the && operator (logical AND) listed in Table 4-3 to add a second condition. Both conditions must be true for iteration of the loop to continue.

8. Insert a statement at the beginning of the while loop to read the most recently received key code from the remote control.

```
keyCode = irRemote.read();
```

9. Modify the if statement such that a press of the STOP button or reception of the channel-down key code will cause the next trick to be proposed.

```
if (stopButton.isPressed() || (keyCode == CHANNEL_DOWN)) {
```

   We have used the || operator (logical OR) listed in Table 4-3 to add a second condition to the if statement. If either of these conditions is true, the statements within the block will be executed.

10. Add an else-if clause such that reception of the channel-up key code will cause the previous trick to be proposed.

```
else if (keyCode == CHANNEL_UP) {
    buzzer.beep();
    trick--;
}
```

11. Add a case to the switch statement to cycle the proposed trick to the last available trick if the channel-up key code is received when the first available trick is proposed.

```
case 0:
    trick = 3;
case 3:
    display.print(0, "Play Tune");
    break;
```

The value of trick will become zero if the channel-up key code is received while the first available trick is proposed. In this case, we want the proposed trick to cycle back to the last of the available tricks, Play Tune. By adding a case to the switch statement to set the trick value to the number of the last available trick, 3, the list will cycle. Inserting this case just prior to the case for the last available trick and leaving out the break statement achieves exactly the behavior we want.

12. Add statements are the end of the while loop to handle the fact that the remote control sends the key code repeatedly whenever a key is held down. Even a brief tap of the key will result in the key code being received multiple times.

```
if (keyCode != -1) {
    do {
        try {
            Thread.sleep(100);
        }
        catch (InterruptedException e) {}
    } while (irRemote.read() != -1);
}
```

This code first checks that a key code has been received. If one has been received, it enters a loop whereby it sleeps for 100 milliseconds and then reads from the receiver again. It continues in the loop until no more key codes are received from the remote control.

13. Ensure your universal remote control is programmed for operation with a Sony television set.

14. Build, load and test your program.

You can now use the channel-up and channel-down buttons on the remote control to scroll through the list of available tricks. You can also use the play button on the remote control instead of the START button on the robot to begin a trick.

Maneuvering by Remote Control

As a final trick, we will program the robot so it can be maneuvered using the remote control. We will use the channel-up button to drive forward, the channel-down button to drive backward, the volume-up button to rotate right and the volume-down button to rotate left. Holding the button down will cause the robot to continue the operation. Releasing the button will cause the robot to stop.

1. Define constants for the volume-up and volume-down key codes.

```
private static final int VOLUME_UP = 146;
private static final int VOLUME_DOWN = 147;
```

2. Add a new case to the first switch statement to handle the new trick. Also, move and adjust case 0 to account for the longer list of tricks.

```
case 0:
    trick = 4;
case 4:
    display.print(0, "Remote Control");
    break;
```

3. Add a case to the second switch statement to call the method for the new trick, remoteControl.

```
case 4:
    remoteControl();
    break;
```

4. Create the remoteControl method.

```
public static void remoteControl() {
}
```

5. Insert a while loop to read key codes received from the remote control.

```
while (true) {
    int keyCode = irRemote.read();
}
```

6. Add a switch statement within the while loop to handle the four maneuvering keys and to stop the robot whenever no key codes are being received from the robot.

```
switch (keyCode) {
case CHANNEL_UP:
    go(Motor.MAX_FORWARD, Motor.MAX_FORWARD, 100);
    break;
case CHANNEL_DOWN:
```

```
        go(Motor.MAX_REVERSE, Motor.MAX_REVERSE, 100);
        break;
    case VOLUME_DOWN:
        go(Motor.MAX_REVERSE, Motor.MAX_FORWARD, 100);
        break;
    case VOLUME_UP:
        go(Motor.MAX_FORWARD, Motor.MAX_REVERSE, 100);
        break;
    default:
        stop();
        break;
    }
```

7. Build, load and test your program.

   You will now be able to remotely control the robot using the channel-up, channel-down, volume-up and volume-down buttons on the remote control.

Summary

One of the most important skills for a robot to possess is the ability to interact with its human users. In this chapter you programmed the IntelliBrain-Bot education robot to allow you to select one of four tricks for the robot to perform. You also learned how to program the robot to receive input via the push buttons, thumbwheel and universal remote control receiver, and to provide output using the LCD screen, LEDs and the buzzer.

In addition to programming the robot to interact with you, you learned how to use if statements, switch statements, arithmetic operators and Boolean operators.

Lastly, you learned how a universal remote control transmits key code values as unique sequences of long and short infrared light pulses that indicate which key has been pressed.

Exercises

1. Write a program to count and display the number of times each push button on the IntelliBrain 2 robotics controller is pressed. Display the number of presses of the START button on the first line of the LCD screen and the number of times the STOP button is pressed on the second line.

2. Write a program to create a heartbeat by blinking the status LED once per second.

3. Write a program that uses the four user LEDs to create a VU meter (segmented volume meter commonly used in stereo equipment) indicating the position of the thumbwheel.

   Hint: Turn all of the LEDs off when the thumbwheel reading is 0. Turn LED 1 on when the thumbwheel reading is between 1 and 255. Turn LED 1 and 2 on when the thumbwheel reading is between 256 and 511, and so on.

4. Enhance your program to blink the LEDs while it plays a tune.

   Hint: Create a method that is a "wrapper" around the buzzer's play method. In this method, prior to calling the buzzer's play method, turn LEDs on and off based on the frequency of the note to be played. Replace the calls to the play method in your playTune method with calls to the new method.

5. Extend your program to play a second tune of your choosing.

6. Enhance your program by changing the thumbwheel value display to be in units appropriate for the proposed trick. For example, display the size of the square in inches or centimeters for the maneuver square trick.

7. Write a program to receive key codes from a universal remote control and display their value on the LCD screen. Create a key code table, similar Table 4-4.

8. Write a program that will enable you to enter an integer value using the key pad on a universal remote control. Display the value of the number as you type it in.

CHAPTER 5

Introduction to Sensing

In the first two DARPA Grand Challenges, the United States Defense Advanced Research Projects Agency, the agency behind initial development of the Internet, issued a challenge to robot builders to build a robotic vehicle that could navigate an unknown course through more than 120 miles (200 km) of desert on its own. "Stanley," a modified Volkswagen Touareg developed by Stanford University researchers, won the $1 million prize by finishing the race in just under 7 hours.

In order to win, Stanley had to find his way through a series of checkpoints he was given immediately prior to the race. He had to rely on his own ability to identify and follow dirt roads without crashing or getting stuck. Stanley had to use his built-in senses and intelligence to find his way without the benefit of any outside help.

Robots like Stanley are intelligent devices capable of accomplishing tasks in an unknown or changing environment. They do this by using sensors to collect information that allows them to perform effectively in a changing environment.

This chapter will introduce you to sensing. You will become familiar with the Ping))) ultrasonic range sensor and then use it to create a "tractor beam." The IntelliBrain™-Bot educational robot will be able to sense and follow your hand, as if there were an invisible beam attaching the robot to your hand.

Sonar Range Sensing

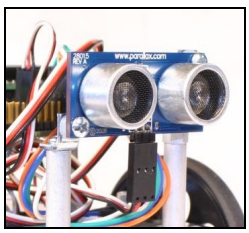


Figure 5-1 – Ping))) Sonar Range Sensor

The IntelliBrain-Bot deluxe educational robot includes a Parallax Ping)))™ sonar range sensor, which is shown in Figure 5-1. This sensor is able to measure the distance to an object in front of the robot by "pinging." This sensor detects objects by generating a

short high frequency sound, then listening for an echo. The sensor will hear an echo if there is an object in front of the robot, as shown in Figure 5-2. If there is no object, the sound will not be reflected and the sensor will not detect an echo.

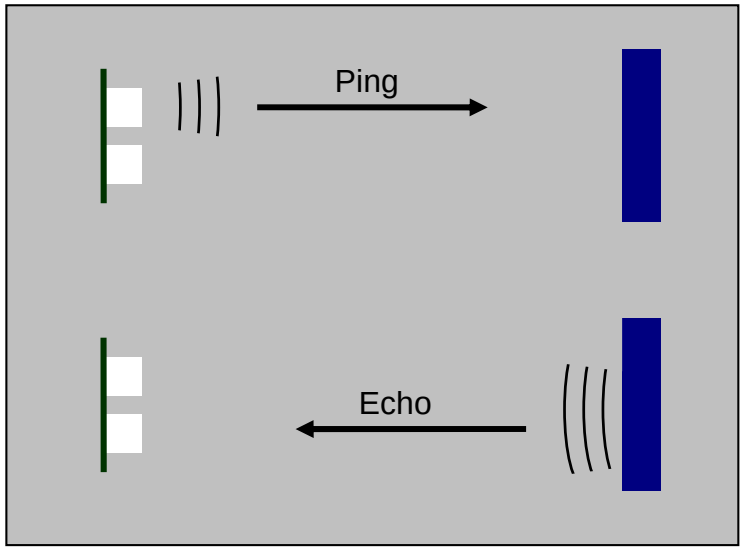


Figure 5-2 – Sonar Ping and Echo

Your program can determine the distance to the object by measuring the time between issuing the ping and hearing the echo. The further away the object is, the longer it will take for the echo to return to the sensor.

Programming the Ping))) Sensor

The RoboJDE Class Library includes a class that provides support for the Ping))) Sensor. This class is named ParallaxPing. However, rather than using the ParallaxPing class we will program the sensor directly. This will enable you to better understand how the sensor functions.

The Ping))) sensor interfaces to the IntelliBrain 2 robotics controller via three wires. The red and black wires provide the sensor with power (red) and the ground (black). The white wire is the signal wire. Through clever design, one signal wire is used to both trigger the sensor to issue a sound pulse and to communicate the echo delay back to the robotics controller. Figure 5-3 illustrates this.

The Ping))) sensor must be connected to one of four digital input/output ports on the IntelliBrain 2 robotics controller that provide pulse input and output features. These are the ports labeled IO3, IO4, IO5 and IO6. The Ping))) sensor should be attached to port IO3 on the IntelliBrain 2 robotics controller.

Your program tells the Ping))) sensor to send a sound pulse by quickly switching the signal on the IO3 port on and off. You can imagine this as if you were flipping a light switch on and off very quickly, only your program must do it faster than humanly possible. In fact, the pulse is so short that it is measured in millionths of a second, or microseconds. As indicated in Figure 5-3, the trigger pulse must be greater than 2

microseconds (usec) long. This timing is so short that it can't be done in software, it must be done by the microcontroller chip on the IntelliBrain 2 robotics controller.

Once the Ping))) sensor receives the trigger pulse, it issues a sound pulse after first holding off for 750 microseconds. The hold off period gives the software on the robotics controller a chance to switch the signal from an output to an input. On the IntelliBrain 2 robotics controller, the virtual machine software takes care of briefly switching the port to an output when your program calls the method to output a pulse, so your program doesn't need to be concerned with this.

The Ping))) sensor sets the signal high (+5 volts) when it issues the sound pulse. It leaves the signal high until it hears the first echo, at which time it sets the signal low (0 volts). Therefore, the time for the sound burst to travel to the closest object and bounce back to the sensor is the amount of time the signal level is high. Your program can determine the echo delay by measuring the time the signal level is high. If the sensor does not hear an echo within 18,500 microseconds, it sets the signal low. There is also a minimum time the Ping))) sensor will leave the signal on. This is 115 microseconds. It is important to take note of the minimum and maximum echo delay. These limit the effective range of the sensor.

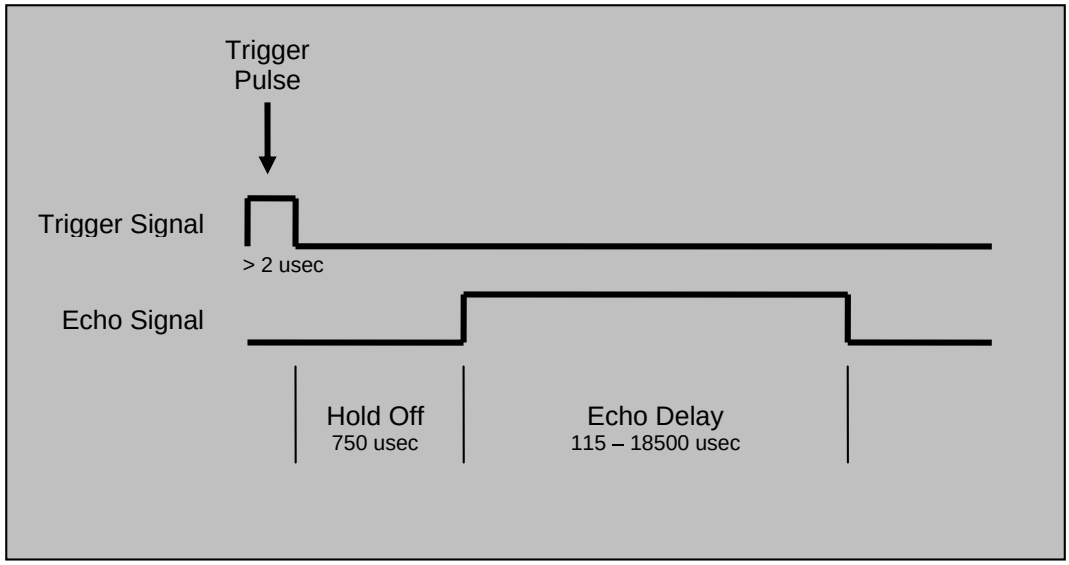


Figure 5-3 –Ping))) Sensor Signals

Create a program to measure and display the time for sound to travel from the Ping))) sensor to the nearest object by doing the following:

1. Review the API documentation for IntelliBrain.getDigitalIO and IntelliBrainDigitalIO.

2. Create a new project named PingTest.

3. Add the following import statements:

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.io.Display;
import com.ridgesoft.intellibrain.IntelliBrainDigitalIO;
```

4. Replace the comment in the main method with a try-catch statement, as follows:

```
try {
}
catch (Throwable t) {
    t.printStackTrace();
}
```

5. Add statements within the try statement to get the display object and print the name of the program.

```
Display display = IntelliBrain.getLcdDisplay();
display.print(0, "Ping Test");
```

6. Add statements to get the port object for IO3 and enable it to be used for pulse measurement.

```
IntelliBrainDigitalIO pingPort = IntelliBrain.getDigitalIO(3);
pingPort.enablePulseMeasurement(true);
```

7. Add a loop that loops forever.

```
while (true) {
}
```

8. Within the loop, use the pulse method of the port object to issue a trigger pulse 20 microseconds in duration.

```
pingPort.pulse(20);
```

9. Add a statement to put the program to sleep for 50 milliseconds, so it will not continue until after the longest possible echo delay pulse will be able to complete.

```
Thread.sleep(50);
```

10. Add a statement to read and display the duration of the echo delay pulse.

```
display.print(1, "Time: " + pingPort.readPulseDuration());
```

    The readPulseDuration returns the pulse duration measurement made by the microcontroller chip.

11. Add another sleep method call such that the program takes two readings per second.

```
Thread.sleep(450);
```

12. Build, load and test the program.

    Hold your hand steady in front of the sensor. Note the reading, then move your hand closer or further away and note the new reading. The reading will increase as you move your hand away and decrease as you move it closer. The approximate range of the readings will be between 115 and 18,500 microseconds.

Measuring the Speed of Sound

You can use your PingTest program to measure the speed of sound. With the program running, hold the robot such that the Ping))) sensor is 1 foot from a wall. The sound pulse will travel 2 feet, as it goes from the sensor to the wall and back to the sensor. You can calculate the speed of sound by dividing the distance traveled by the time of travel. In this case the distance is 2 feet and the time is the echo delay measured by your program.

speed of sound = distance traveled / time of travel

When you do this experiment, you will measure an echo delay of approximately 1800 microseconds. Performing the calculation above, you will find the speed of sound is approximately 1100 feet/second.

Calculating Distance

Knowing the speed of sound, you can now modify your program to display distance rather than time. To do this, you will need to use the equation:

distance = rate * time

The rate is the speed of sound. The distance we are interested in is the distance to the nearest object. This is one half the distance the sound pulse travels, so we need to divide the result by two. Substituting yeilds:

distance = speed of sound * round trip time / 2

Substituting the value for the speed of sound and accounting for unit conversions yields:

distance = 1100 ft/sec * 12 in/ft * 1/1,000,00 sec/usec * round trip time / 2

or

distance = round trip time / 150

where round trip time is in microseconds and the result is in inches.

Update your program to display distance, as follows:

1. Replace the existing statement to display the sensor reading with a statement to read the round trip time and assign it to a new variable.

```
int roundTripTime = pingPort.readPulseDuration();
```

2. Add statements to only display the distance if the sensor reading is within its valid range (115 – 18500), otherwise display "--" to indicate no object is in range.

```
if (roundTripTime < 115 || roundTripTime > 18500)
    display.print(1, "Distance: --");
else
    display.print(1, "Distance: " + roundTripTime / 150 + '"');
```

3. Build, load and test your program.

   Hold your hand at various distances. Observe the distance values displayed are correct.

Sensor Performance

The Ping))) sensor can sense objects in a cone shaped region directly ahead of the sensor, as shown in Figure 5-4. The dimensions of the region vary based on the properties of the object being sensed as well as the properties of the surrounding surfaces.

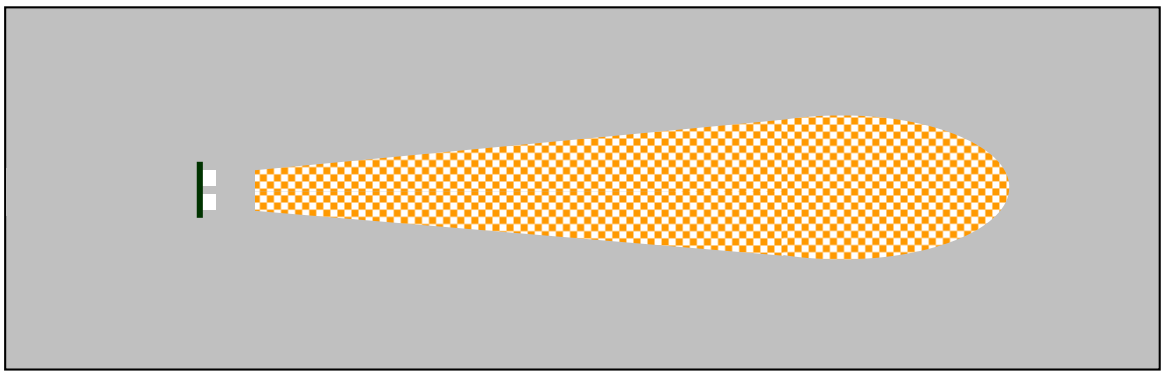


Figure 5-4 – Effective Sensing Region of Ping))) Sensor

Using the Ping))) Sensor

An interesting and fun application of the Ping))) sonar range sensor is to use it to create a "tractor beam" effect. A tractor beam is a science fiction device that forms an invisible connection between two objects. These devices are frequently used in fiction writing to enable one spaceship to tow another, keeping the towed ship at a fixed distance from the towing ship.

Let's write a program using the Ping))) sensor to create a tractor beam. Once you have completed the program, the IntelliBrain-Bot robot will be able to form an invisible connection to an object. The robot will move forward or back to maintain a fixed spacing from to the object. When you place your hand in front of the robot, it will follow it forward and back, creating the illusion of an invisible beam between your hand and the robot.

Surprisingly, creating this program is not nearly as challenging as it might seem. All the program needs to do is use the Ping))) sensor to repeatedly measure the distance to the nearest object and adjust the power to the motors on each repetition. If the distance is too large, the program needs to power the motors forward. If the distance is too small, the program needs to power the motors in reverse. If the distance is just right, the motors need to be turned off.

Let's go ahead and create the program, as follows:

1. Review the API documentation for the SonarRangeFinder and ParallaxPing classes. Pay particular attention to the getDistanceInches method.

2. Create a new project named "TractorBeam."

3. Add import statements to the TractorBeam.java source file for the library classes the program will refer to.

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.io.Display;
import com.ridgesoft.robotics.PushButton;
import com.ridgesoft.robotics.Motor;
import com.ridgesoft.robotics.ContinuousRotationServo;
import com.ridgesoft.robotics.SonarRangeFinder;
import com.ridgesoft.robotics.sensors.ParallaxPing;
```

4. Place a try-catch statement in the main method.

```
try {
}
catch (Throwable t) {
    t.printStackTrace();
}
```

5. Add statements within the try clause to obtain the Display for the LCD device and print the name of the program.

```
Display display = IntelliBrain.getLcdDisplay();
display.print(0, "Tractor Beam");
```

6. Create motor objects for the two motors, again within the try clause.

```
Motor leftMotor = new ContinuousRotationServo(
    IntelliBrain.getServo(1), false, 14);
Motor rightMotor = new ContinuousRotationServo(
    IntelliBrain.getServo(2), true, 14);
```

7. Create a SonarRangeFinder object for the Ping))) sensor.

```
SonarRangeFinder pingSensor =
    new ParallaxPing(IntelliBrain.getDigitalIO(3));
```

8. Create a loop that runs forever.

```
while (true) {
}
```

9. Add statements within the loop to issue a sonar ping, wait long enough for the echo to be heard and then read the range in inches.

```
pingSensor.ping();
Thread.sleep(100);
float range = pingSensor.getDistanceInches();
```

10. Add statements to display the range or "--" if there is no object in range.

```
if (range > 0.0f)
    display.print(1, Integer.toString((int)(range + 0.5f)) + '"');
else
    display.print(1, "--");
```

11. Build, load and test the program.

    The program will display the number of inches to the nearest object or "--" if there is no object in range.

We have used a new data type in this program we haven't used before, float. Let's discuss data types before completing the TractorBeam program.

Numeric Data Types

Up until the previous exercise, the only data type we had used was "int," to declare integer variables. An int type variable can represent whole numbers (integers) between -2,147,483,648 and 2,147,483,647, inclusive. While this is a fairly large range of numbers, int variables can't represent numbers that are not whole. For example, if you used an int variable to hold the value of Pi (3.14159…) it would have to be approximated by the nearest whole number, 3. Additionally, if your program performs calculations using integers, you have to be careful to avoid intermediate results that can't be

represented with adequate precision. If you aren't careful to avoid precision problems truncation and overflow errors may cause wildly inaccurate results. For example, if you divide 3 by 2, the integer result will be 1. The fractional portion of the result, .5, will be truncated because integers can only be whole numbers. If you multiply by 2 again, the result will be two, so the result of (3 / 2) * 2 will be 2, not 3. However, the result of (3 * 2) / 2 is 3, as you would expect. Doing the division after the multiplication avoids the truncation error.

In order to overcome the limitations of integers, Java also supports the "float" data type. The major difference between float and int is float variables can represent real numbers, while int variables can only represent whole numbers. Therefore, a float variable can more precisely represent numbers that are not whole, such as Pi. Float values can be very small – as small as $\pm 1.4 \times 10^{-45}$ – and very large – as large as $\pm 3.4 \times 10^{38}$. The primary disadvantage of the float data type is that it takes significantly more computation to do arithmetic and comparison operations than for the int data type. In other words, the same program will run slower if it uses float variables instead of int variables. However, improved precision may be worth the extra computation cost.

Java supports other data types besides int and float. These are listed in Table 5-1. Typically, you can use int and float for most of your variables. The smaller integer data types, byte, short and char, are useful if your program stores a large amount of data and you need to conserve space in the robot's memory. The double floating point data type is not fully supported by the RoboJDE virtual machine. The double data type has the same range and precision as float the float data type, but takes up 64 bits of memory instead of 32 bits for a float. Normally this will not present a problem, but you should use float instead of double whenever possible.

Table 5-1 – Numeric Data Types

Type	Size	Range
Integer		
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
char	16 bits	0 to 65768
int	32 bits	-2,147,483,648 to 2,147,483,647
long	64 bits	-2^{63} to $2^{63}-1$
Floating Point		
float	32 bits	-3.4×10^{38} to 3.4×10^{38}
double	64 bits	-3.4×10^{38} to 3.4×10^{38}

Selecting a Data Type

With so many data types to choose from, you might be asking yourself, how do I choose the best one?

There are three things to consider when selecting a data type:

1. The range and type of values the variable needs to store.
2. The relative amount of computation time to perform arithmetic operations using various data types.
3. The amount of memory required by the data type.

Integer data types can only represent whole numbers and have a more limited range than floating point numbers. However, it takes significantly less time to perform arithmetic calculations using integer data than it does using floating point data. Normally, it is preferable to select an integer data type whenever an integer will suffice. Only use float when an integer data type is not sufficient.

The amount of memory required to store variables is only a concern when dealing with large amounts of data. If this is the case, choose the smallest data type that meets your needs, otherwise, use int or float.

The amount of time it takes for the microcontroller to perform arithmetic operations depends on the data type your program uses. Floating point computations take significantly longer than similar operations using integers. Therefore, you can make your programs more efficient by preferring to use integers instead of floating point numbers. Use integers whenever your program is working with whole numbers within the integer range. Use floating point numbers only when whole numbers are not sufficient or the range of integer numbers is too small.

Java converts byte, short and char values to int values whenever they are used in arithmetic expressions. There is no computation time advantage to using integer data types smaller than int. However, there is a significant computation time advantage to using int instead of long.

The RoboJDE virtual machine treats double arithmetic the same float, therefore, there is not a significant difference in computation time.

Converting Between Data Types

Java allows you to convert values of one data type to another. This is called "casting." The cast operator is denoted by the target data type enclosed in parenthesis. For example, "(int)" is the operator to cast from any other data type to the int data type. The following code illustrates casting.

```
int i = 10;
byte b = (byte)i;
short s = (short)i;
char c = (char)i;
float f = (float)i;

f += 0.6f;
int j = (int)f;
int k = (int)(f + 0.5f);
```

When you use casting you need to be sure the range of possible values of the source variable will always be within the range of the destination.

Casting a floating point number to an integer causes the number to truncated, not rounded. In the example above, the value of f, 10.6, will be truncated to 10 when calculating the value to assign to j. You can round floating point numbers by adding 0.5 before casting to an integer. In the example above, the value 11 will be assigned to k.

Java automatically converts byte, short and char data types to int whenever they are used in arithmetic or logical operations. In the example below, the value of b will automatically be cast to an int prior to multiplying it by 100. The value assigned to i will be 12,300.

```
byte b = 123;
int i = 100 * b;
```

Implementing the Tractor Beam Effect

In order to implement the tractor beam effect we need to use distance reading measurements to control the power applied to the motors. If the distance to the object is further than desired, the motors need to be powered forward. If the distance to the object is less than desired, the motors need to be powered backward. Otherwise, the motors need to be turned off. We will use a second press of the START button to activate the tractor beam.

1. Add the following statements immediately prior to the while loop.

```
PushButton startButton = IntelliBrain.getStartButton();
startButton.waitReleased();
boolean go = false;
```

2. Add the following statements immediately after the statement to read the range.

```
if (go) {
}
else if (startButton.isPressed()) {
    go = true;
}
```

   These statements cause the program to wait for the second press of the START button before executing the code in the go clause of the if statement.

3. Within the go clause of the if statement above, add another if statement which checks to see if there is an object within 20 inches of the robot. If there is not, stop the motors.

```
if (range > 0.0f && range < 20.0f) {
}
else {
```

```
        leftMotor.stop();
        rightMotor.stop();
    }
```

4. Add statements within the range check clause of the above if statement to power the motors forward if there is an object more than 6.5 inches from the robot, power the motors in reverse if there is an object within 5.5 inches of the robot, and stop the motors if there is an object between 5.5 and 6.5 inches of the robot.

```
if (range > 6.5f) {
    leftMotor.setPower(Motor.MAX_FORWARD);
    rightMotor.setPower(Motor.MAX_FORWARD);
}
else if (range < 5.5f) {
    leftMotor.setPower(Motor.MAX_REVERSE);
    rightMotor.setPower(Motor.MAX_REVERSE);
}
else {
    leftMotor.stop();
    rightMotor.stop();
}
```

5. Build, load and test your program.

   Place your hand in front of the sensor. If your hand is closer than 5.5 inches the robot will backup. If your hand is further than 6.5 inches the robot will move forward. Notice the robot is a bit jerky when it starts and stops. This is because the way the programs is controlling the motors. In the next exercise we will make the robot operate more smoothly.

Proportional Control

In the previous exercise you were able to create the tractor beam effect by powering the motors to correct for an "error" in the desired distance to the nearest object. There is error if the nearest object is not between 5.5 and 6.5 inches away from the robot. The program attempts to eliminate the error by powering the motors in the direction that will reduce the error. The program does this in a simple minded way, by going forward or back at full power until there is no error. This results in a jerky response and overshooting the desired position because the motors are either on at full power or completely turned off. This approach is often called "bang-bang control" because the program bangs the power from one extreme – off – to the other extreme – full power. You can image if you were in a car where the driver used this technique to control the speed, it would result in a very uncomfortable ride!

We can correct the jerky behavior by modifying the program to use "proportional control." With proportional control the amount of power applied to correct for the error is varied in proportion to the error. If the error is small, a small amount of power is applied to the wheels. If the error is large, a large amount of power is applied to the wheels. As the error decreases the power is decreased. If there is no error, no power is

applied to the wheels. This is easily accomplished by making the power proportional to the amount of error, as follows:

power = error * gain

The gain is a constant multiplier which makes the power level proportional to the error. You can see the power will be zero if there is no error, and the greater the error, the greater the power. The value of the gain has to be carefully determined such that the robot will not overreact or under react to error. A reasonable gain value can be determined by experimentation.

With the current tractor beam exercise, there is no error when the object in front of the robot is exactly six inches away. Therefore, the error can be calculated by the difference between the actual range to the object and the desired range:

error = range – 6.0

Convert your program to use proportional control as follows:

1. Replace the power setting statements with proportional control calculations.

```
if (range > 0.0f && range < 20.0f) {
    float gain = 5.0f;
    float error = range - 6.0f;
    int power = (int)(gain * error);
    if (power != 0) {
        leftMotor.setPower(power);
        rightMotor.setPower(power);
    else {
        leftMotor.stop();
        rightMotor.stop();
    }
}
else {
    leftMotor.stop();
    rightMotor.stop();
}
```

   Based on experimentation, a gain value of 5.0 has been determined to work well.

   Note the calculated power value may exceed the limits defined by the Motor.setPower method. However, the Motor.setPower method handles this by constraining out-of-range values to the maximum or minimum, whichever is appropriate.

2. Build, load and test your program.

   The robot will now track your hand more smoothly.

Summary

In this chapter, you learned how robots can use sensors to respond to their environment. You used an ultrasonic range sensor to measure the distance to an object in front of the robot. By measuring the time it takes for a high frequency sound pulse to travel to an object and return back to the sensor and knowing the speed of sound, your program was able to determine the distance to the nearest object.

Using distance measurements, you implemented a "tractor beam" by programming the robot to strive to maintain a fixed distance, six inches, from your hand. You used a proportional control algorithm to vary the power to the robot's wheels in proportion to the error between the desired distance and the actual distance to your hand. This provided for smoother motion than the simpler bang-bang control technique, which you also implemented.

Finally, you learned about the various numeric data types Java supports. You learned that integers can only represent whole numbers. This can lead to inaccurate results if you are not careful to consider the effect of truncation errors. You can use floating point numbers for greater precision when working with real numbers. Doing so greatly reduces the concern you need to have for truncation errors. However, computations using floating point numbers take much longer than similar computations using integers.

Exercises

1. Using your PingTest program, hold the robot in your hands and face a wall. Walk toward the wall and away from the wall. What are the nearest and furthest distances at which the sensor can accurately measure the distance to the wall?

2. Using your PingTest program, measure the dimensions of your classroom. How high is the ceiling?

3. Using your PingTest program, make several plots similar to Figure 5-4 showing the dimensions of the effective sensing region of the Ping))) sensor. Use a different object, such as a book, cup, jacket or another robot for each plot. How do the properties of the object affect the ability to sense it?

4. Try various gain settings in your TractorBeam program and observe the robot's response to movements of your hand. How does the robot's behavior change if you set the gain constant to 1.0? How does the robot's behavior change if you set the gain constant of 20.0?

5. Modify your TractorBeam program to enable the thumbwheel to be used to adjust the gain constant. Run the program and use the thumbwheel to find the gain value which you think is best.

CHAPTER 6

Line Following

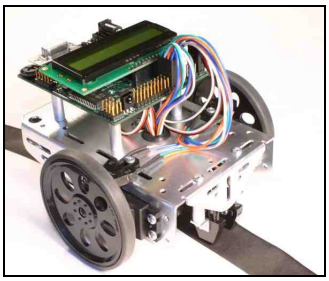


Figure 6-1 - IntelliBrain-Bot Educational Robot Following a Line

The IntelliBrain™-Bot Deluxe education robot incorporates two Fairchild QRB1134 photo-reflective sensors on the underside of the robot, as shown in Figure 6-1. In this chapter, you will learn how to program the robot to use these sensors to follow a line.

Line Sensing

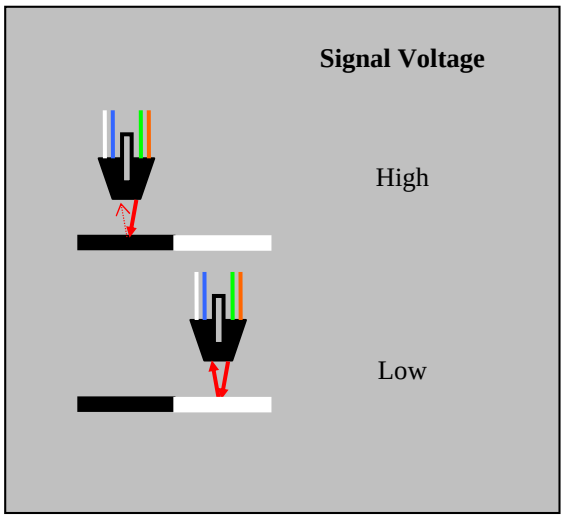


Figure 6-2 - Sensing a Line

Each line sensor consists of an infrared light emitting diode (LED) and a phototransistor mounted side by side. The LED outputs infrared light and the phototransistor receives infrared light. When the sensor is close to a surface, light emitted by the LED is reflected on to the phototransistor, as shown in Figure 6-2. The voltage on the signal lead of the phototransistor varies depending on the amount of light that is reflected on to its receiver. When a lot of infrared light is reflected on to the receiver, the voltage is low. When a small amount of infrared light is reflected on to the receiver, the voltage is high. Hence, the signal voltage is low when the sensor is over a white (highly reflective) surface and the signal voltage is high when the sensor is over a black (less reflective) surface. The signal voltage is highest when the sensor is not near any surface and, consequently, there is no reflection at all.

Working with Analog Sensors

In order to make use of each line sensor, your program will need to determine the voltage level on the sensor's signal lead. Your program can do this by using one of the IntelliBrain 2 robotics controller's built-in Analog-to-Digital converters to measure the voltage at the signal pin. The IntelliBrain 2 controller has seven analog input ports. The left line sensor is attached to analog port 6 and the right line sensor is attached to port 7. Initially, we will work with just the left line sensor, which connects to analog port 6 on the IntelliBrain 2 controller. You can obtain the object for analog port 6 as follows:

```
AnalogInput lineSensor = IntelliBrain.getAnalogInput(6);
```

You can sample the voltage at the port's signal pin using the port object's sample method, as follows:

```
int sample = lineSensor.sample();
```

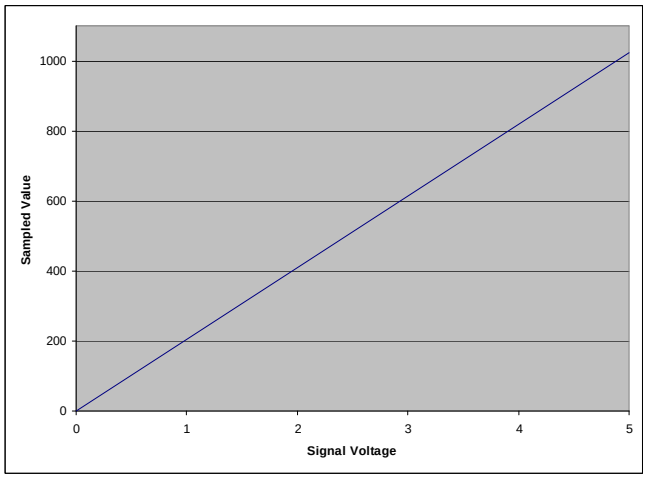


Figure 6-3 - Sampled Value Verses Signal Voltage

The sample method uses the IntelliBrain 2 controller's A-to-D converter to measure the analog voltage at the port 6 signal pin and convert it to an integer value. The value is proportional to the voltage at the port's signal pin, as shown in Figure 6-3. A sampled value of 0 corresponds to 0 volts and a sampled value of 1023 corresponds to 5 volts.

Testing the Line Sensor

Whenever you begin working with a new sensor, it is always a good idea to write a small program to sample and display its reading. This will allow you to experiment and verify the sensor functions as you expect. It will also enable you to test the sensor to make sure it is functioning properly.

Let's experiment with a line sensor by writing a simple program to sample and display its current reading. Later, we will expand upon this program to make the robot follow a line.

1. Create a new project named LineFollower1.

2. Add the following import statements:

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.io.Display;
import com.ridgesoft.robotics.AnalogInput;
```

3. Define the LineFollower1 class and the main method.

```
public class LineFollower1 {
    public static void main(String args[]) {
    }
}
```

4. Within the main method, get the display object and clear both lines of the display.

```
Display display = IntelliBrain.getLcdDisplay();
display.print(0, "");
display.print(1, "");
```

5. Get the AnalogInput object for analog input port 6, the port to which the left line sensor connects.

```
AnalogInput lineSensor = IntelliBrain.getAnalogInput(6);
```

6. Create a while loop to sample the sensor and display its value.

```
while (true) {
    int sample = lineSensor.sample();
    display.print(0, Integer.toString(sample));
}
```

7. Build, load and test your program.

   Set the robot down with the left line sensor over surfaces with various colors and textures. Note how the line sensor reading varies depending on the reflectivity of each surface. A flat black surface will reflect the least infrared light. In this case, the sensor reading will be high – greater than 500. A glossy white surface will

reflect the most infrared light. In this case, the sensor reading will be low – less than 100.

Identifying the Line

In order to follow the line, your program will need to be able to identify the line. It will be easiest to sense the line if there is a large contrast between the line and the background surface. A flat black line on an opaque bright white background is ideal. If you do not have a line following poster, you can use a strip of one inch wide black electrical tape on a white or light colored surface to create your own line following course. Determine the sensor reading ranges for your line following course as follows:

1. Using your LineFollower1 program, place the robot on the floor with the left line sensor over the middle of the line and record the sensor reading in Table 6-1.

2. Repeat with the sensor over the background and over the edge of the line.

Table 6-1 – Line Sensor Readings Relative to Line

Sensor Position	Sensor Reading
Over middle of line	
Over background	
Over edge of line	

Following a Line Using One Sensor

In order to follow the line, your program will need to use the line sensor to provide feedback indicating the position of the robot relative to the line. You could program the robot to drive straight ahead when the sensor is over the middle of the line. However, when the robot drifts off the line, your program will have no way to know if it is left of the line or right of the line when using just one line sensor. Therefore, your program would have no way of knowing whether to steer left or right to get back on course. This is a problem!

You can solve this problem by programming the robot to follow the edge of the line, rather than the middle of the line. By following the edge of the line, the program will know the robot has drifted left or right of the edge. The sensor will read low when the robot has drifted away from the edge (over the white background). It will read high when the robot has drifted over the center of the line (over the black line). You can program the robot to follow either the left or right edge of the line. You must choose one edge or the other.

Since we are using the left sensor, we will choose to follow the left edge of the line. When the robot drifts off course to the left, the sensor will move over the white background and will read low. Your program will need to steer the robot to the right. It can do this by applying more power to the left wheel than to the right wheel. Vice versa, when the robot drifts right, the sensor will move over the black line. The sensor will read high. Your program will need to steer the robot left by applying more power to the right wheel than the left wheel.

We can use the proportional control technique you learned about in the previous chapter to correct for an error in the robot's position relative to the line. In this case, your program will need to steer the robot left or right depending on the error it senses in the robot's position relative to the line. Your program will need to apply more power to one motor and less power to the other motor to steer the robot left or right. It can do this by calculating an offset proportional to the error between the desired sensor reading – the set point – and the actual sensor reading, then applying the offset to the power applied to each wheel. By adding the offset to the power applied to one motor and subtracting it from the power applied to the other motor, the robot will arc left or right, back toward the edge of the line. The greater there error, the more aggressively the robot will turn.

The power offset can be calculated as follows:

```
offset = (setPoint - reading) * gain
```

Your program will steer the robot by applying the offset in opposite directions when applying power to the motors, as follows:

```
leftPower = power + offset
rightPower = power - offset
```

In this way the average power applied to the motors will be positive, ensuring the robot moves forward. The offset will cause the robot to arc left or right as it moves forward.

Extend your LineFollower1 program to program the robot to follow the line, as follows:

1. Add import statements for PushButton, Motor and ContinuousRotationServo.

```
import com.ridgesoft.robotics.PushButton;
import com.ridgesoft.robotics.Motor;
import com.ridgesoft.robotics.ContinuousRotationServo;
```

2. Add statements to create two Motor objects.

```
Motor leftMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(1), false, 14);
Motor rightMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(2), true, 14);
```

3. Add statements to retrieve the start button object and wait for the start button to be released.

```
PushButton startButton = IntelliBrain.getStartButton();
startButton.waitReleased();
```

4. Modify the existing while loop condition to loop until the start button is pressed a second time.

```
while (!startButton.isPressed()) {
```

5. Add a second loop after the first loop to execute the control algorithm once the start button has been pressed a second time.

```
display.print(0, "Following line");
while (true) {
    int sample = lineSensor.sample();
}
```

6. Determine the set point value to use in the proportional control equation by averaging the over line and over background sensor readings in Table 6-1.

   By choosing the midpoint between the extreme readings as the set point, the control algorithm will strive to maintain the sensor's position near the edge of the line.

7. Estimate a value for the gain constant and add a statement to your program to declare and initialize a variable named gain. Insert this statement prior to the first loop.

   One way to estimate the gain constant is to choose a value such that, at the extremes, one wheel will receive full power and the other wheel will receive no power. Assuming a base motor power setting of 8, the maximum desired power offset is 8. This will result in one motor receiving full power (16) and the other motor receiving no power (0) in the most extreme cases. The maximum variation of any sensor reading from the set point will be less than the half the range of possible readings from the sensor. The full range of an analog port is 0 to 1023. Therefore, half of the range is 512. Dividing the maximum desired offset, 8, by half the sensor range, 512, yields an estimate for the gain constant of 0.016.

```
float gain = 0.016f;
```

8. Within the second loop, after the statement to sample the sensor, add the proportional control equation to calculate the power offset in the go branch of the if statement.

```
float offset = (360.0f - (float)sample) * gain;
```

9. Following the offset calculation, set the power to each motor, adding the offset to the base setting for the left motor and subtracting it from the base setting for the right motor. Use 8 as the base power setting.

```
leftMotor.setPower((int)(8.0f + offset));
rightMotor.setPower((int)(8.0f - offset));
```

10. Build, load and test your program.

    Set the robot down with the left line sensor over the left edge of the line. Press the start button once to start the program and a second time to start the robot following the line. Observe the robot's ability to follow the line. If the robot is sluggish in responding and drifts away from the line or over the line, try increasing the gain constant about 10%. If the robot is jittery, try decreasing the gain constant about 10%.

Following a Line Using Two Sensors

Using a second line sensor will allow you to use an entirely different approach to programming the robot to follow a line. With two line sensors your program can detect if the robot is left, right or centered over the line. In contrast, with only one sensor your program could only determine if the sensor was over the line or not. It had no way to know if it was left or right of the line without following one edge of the line. With two sensors, your program can sense which side of the line the robot is on. Therefore, it can correct for left or right drifts.

Using a Finite State Machine

Considering the position of the two sensors relative to the line, at any point in time the robot will be in one of the six states listed in Table 6-2. These states are defined by the position of the sensors relative to the line. The actions listed in the table define the action your program will need to take in order to cause the robot follow the line. For example, when both sensors are over the line, your program will need to steer the robot straight ahead. If the robot drifts slightly left, such that the left sensor is not over the line, but the right sensor is over the line, your program will need to steer slightly right. If the robot drifts so far left that both sensors are left of the line, your program will need to steer the robot hard to the right to get back to the line. Similarly, your program will need to steer left when the robot drifts right.

The Lost state in Table 6-2 is necessary to account for the possibility that your program won't know if the robot is left or right of the line when neither sensor senses the line. This can occur when the program is started when both sensors are off the line. It can also occur if the robot drifts off the line so fast that your program misses whether the robot was drifting left or drifting right. Finally, it can occur if the robot drives off the end of the line. In any of these cases, the robot has gotten lost. The program should, therefore, stop the robot rather than risk wandering into a hazard.

Table 6-2 - Line Sensing States

State Name	State Description	Action
Lost	Neither sensor over line, position unknown	Stop
Both Left	Both sensors left of line	Steer hard right
One Left	One sensor left of line	Steer slightly right
Centered	Both sensors over line	Steer straight ahead
One Right	One sensor right of line	Steer slightly left
Both Right	Both sensors right of line	Steer hard left

Another way to represent these states is using a state diagram, as shown in Figure 6-4. This diagram may look like a bewildering set of bubbles and connecting arrows, but once you understand the notation it is fairly easy to comprehend.

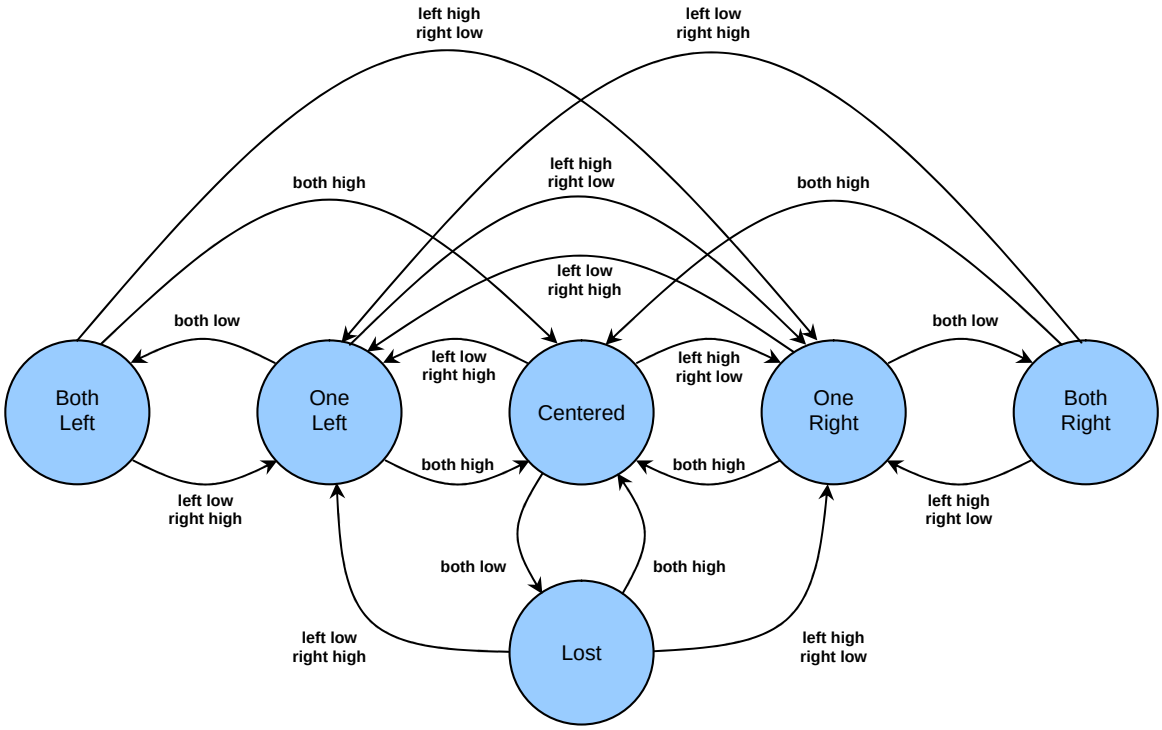


Figure 6-4 - Line Sensor State Diagram

Notice that the six states we have already discussed are each represented by a bubble in the diagram. The arrows represent the conditions that cause the state machine to transition from one state to another. The specific conditions which cause each transition are listed next to the arrow for that transition. In this case, the conditions are changes in the line sensor readings. For example, the state machine will transition to the Centered state anytime both sensors read high. Observe that the state machine can transition into this state from any of the other five states. Each of the five transitions leading to the Centered state is due to the same condition, both sensors reading high. That is, each sensor is over the line and, therefore, the robot is approximately centered over the line.

Further examining the diagram, notice for each state there are three conditions that cause a transition out of the state. These occur when the sensor readings aren't the readings expected for that state. This is due to the fact that there are four permutations of sensor readings: (low, low), (low, high), (high, low) and (high, high). For each state, one permutation corresponds to that state and the other three indicate a condition that causes a transition to a different state. Hence, there are three conditions for each state that will result in a transition to a different state.

Lastly, note that three of the permutations of sensor readings correspond to exactly one state: low, low – One Left; high, high – Centered; and high, low – One Right. All of the transitions into these states are due to the same conditions. However, both sensors reading may indicate any one of three states: Both Left, Both Right and Lost. In this case, the previous state determines the new state. If the robot was drifting left (One Left) when both sensors began reading low, it indicates the robot has drifted further left of the line and entered the Both Left state. Likewise, if the robot was drifting right (One Right) when both sensors began reading low, it indicates the robot has drifted further right of the line and entered the Both Right state. If both sensors were over the line (Centered) and suddenly they both begin reading low, there isn't any way to know if the robot drifted left, right or ran off the end of the line. Therefore, the robot has entered the Lost.

Table 6-3 - State Transition Table

	Conditions (left sensor, right sensor)			
Current State	**(low, low)**	**(low, hi)**	**(hi, low)**	**(hi, high)**
Lost	Lost	One Left	One Right	Centered
Both Left	Both Left	One Left	One Right	Centered
One Left	Both Left	One Left	One Right	Centered
Centered	Lost	One Left	One Right	Centered
One Right	Both Right	One Left	One Right	Centered
Both Right	Both Right	One Left	One Right	Centered

Defining a State Machine

You can define a state machine in software by representing it as tables of data. The tables specify the operation of the state machine. Using tables reduces the complexity of the logic you will need to program. The statements to implement the state machine reduce down to the following pseudo code:

```
while (true) {
    sample sensors
    evaluate conditions
    lookup the new state
    perform actions for the new state
}
```

Let's begin developing a program to use this method of following a line.

1. Create a new project named LineFollower2.

2. Add import statements for IntelliBrain, Display and AnalogInput.

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.io.Display;
import com.ridgesoft.robotics.AnalogInput;
```

3. Add statements to sample and display the readings of both line sensors.

```
Display display = IntelliBrain.getLcdDisplay();

AnalogInput leftLineSensor =
                          IntelliBrain.getAnalogInput(6);
AnalogInput rightLineSensor =
                          IntelliBrain.getAnalogInput(7);

while (true) {
    int leftSample = leftLineSensor.sample();
    int rightSample = rightLineSensor.sample();
    display.print(0, Integer.toString(leftSample) + ' '
                        + Integer.toString(rightSample));
}
```

4. Build, load and test your program.

   Place the robot over the line and observe the sensor readings. Change the robot's position to correspond to each of the four possible combinations of sensor conditions: 1) both high (centered over line), 2) both low (entirely off line), 3) left low and right high (slightly left of center) and 4) left high, right low (slightly right of center). Verify that both sensors read as expected. Also, note the highest and lowest readings of each sensor.

Next you will need to create the state tables in your program. We will use another feature of Java we have not used before as we create the tables, constants.

Constants

Constants are variables you declare in you program that have a value that does not change. You initialize the variable with its constant value when you create it. You can do this as follows:

```
private static final byte LOST = 0;
private static final byte BOTH_LEFT = 1;
private static final byte ONE_LEFT = 2;
private static final byte CENTERED = 3;
private static final byte ONE_RIGHT = 4;
private static final byte BOTH_RIGHT = 5;
```

Notice these constants are byte variables declared with the "final" keyword. This keyword specifies the values are final and cannot be changed when the program is running. If you put a statement in your program to assign a new value to a final variable, the Java compiler will report an error.

Arrays

Arrays provide a way to organize a collection of related data of the same type. For example, you can use an array to keep track of the possible state transitions from one particular state to the next state. For the Lost state, the list of next states may be organized in an array as follows:

LOST	ONE_LEFT	ONE_RIGHT	CENTERED

You can define this as a byte array in your program using following statement:

```
byte[] nextState =
      new byte[] { LOST, ONE_LEFT, ONE_RIGHT, CENTERED };
```

Note, the values between the braces ({}) initialize the elements of the array.

Each element of an array is identified by an integer index, starting with zero. The elements of the array are indexed as follows:

0	1	2	3
LOST	ONE_LEFT	ONE_RIGHT	CENTERED

Your program can access an element of an array using its index. For example:

```
state = nextState[2];
```

After executing this statement, the value of state would be the value of the constant ONE_RIGHT, which is 4.

We can use a two dimensional array to represent the entire state transition table as follows:

	0	1	2	3
0	LOST	ONE_LEFT	ONE_RIGHT	CENTERED
1	BOTH_LEFT	ONE_LEFT	ONE_RIGHT	CENTERED
2	BOTH_LEFT	ONE_LEFT	ONE_RIGHT	CENTERED
3	LOST	ONE_LEFT	ONE_RIGHT	CENTERED
4	BOTH_RIGHT	ONE_LEFT	ONE_RIGHT	CENTERED

5	BOTH_RIGHT	ONE_LEFT	ONE_RIGHT	CENTERED

You can define this as a two dimensional array in your program as follows:

```
private static final byte[][] NEXT_STATE = new byte[][] {
    new byte[] { LOST,       ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_LEFT,  ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_LEFT,  ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { LOST,       ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_RIGHT, ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_RIGHT, ONE_LEFT, ONE_RIGHT, CENTERED },
};
```

Your program can access an element in this two dimensional array as follows:

```
state = NEXT_STATE[3][0];
```

The first index value selects the byte array corresponding to a particular state and the second index selects the element within that array. Therefore, in the statement above, the variable state will be assigned the value of LOST, which is 0.

The first index of the state transition array is the current state and the second index corresponds to the condition that causes the transition to the new state. Replacing the numeric value of the first index with the corresponding state name and listing the corresponding sensor conditions above the second index, the state transition table is as follows:

	(low, low) 0	(low, high) 1	(high, low) 2	(high, high) 3
LOST	LOST	ONE_LEFT	ONE_RIGHT	CENTERED
BOTH_LEFT	BOTH_LEFT	ONE_LEFT	ONE_RIGHT	CENTERED
ONE_LEFT	BOTH_LEFT	ONE_LEFT	ONE_RIGHT	CENTERED
CENTERED	LOST	ONE_LEFT	ONE_RIGHT	CENTERED
ONE_RIGHT	BOTH_RIGHT	ONE_LEFT	ONE_RIGHT	CENTERED
BOTH_RIGHT	BOTH_RIGHT	ONE_LEFT	ONE_RIGHT	CENTERED

We have conveniently chosen the index values corresponding to the sensor conditions such that each of the two least significant binary digits in the index corresponds to one sensor, as indicated in Table 6-4. The least significant bit corresponds to the right sensor.

The next most significant bit corresponds to the left sensor. A bit value of 0 corresponds to a low reading and the bit value 1 corresponds a high reading.

Table 6-4 - Condition Index Value

Sensor Conditions		Index	
Left	**Right**	**Binary**	**Decimal**
low	low	00	0
low	high	01	1
high	low	10	2
high	high	11	3

The condition index can be calculated as follows:

```
int conditions = 0;
if (leftSample > THRESHOLD)
    conditions |= 0x2;
if (rightSample > THRESHOLD)
    conditions |= 0x1;
```

THRESHOLD is a constant defining the threshold value that distinguishes between high and low sensor readings.

Given the state transition table, the current state and the conditions index, transitioning to the new state requires only the following line of code:

```
state = NEXT_STATE[state][conditions];
```

Implementing Two Sensor Line Following

Extend your LineFollower2 program to track and display the current state by completing the following steps:

1. Right after the class definition, declare a constant to define the threshold between high and low readings of the line sensors.

   Choose a value halfway between the low value the sensor reads when it is over the background and the high value it reads when over the center of the line. For a bright white background and black line, a value of 300 usually works well.

   ```
   private static final int THRESHOLD = 300;
   ```

2. Declare constants to identify the six states.

   ```
   private static final byte LOST = 0;
   private static final byte BOTH_LEFT = 1;
   private static final byte ONE_LEFT = 2;
   private static final byte CENTERED = 3;
   private static final byte ONE_RIGHT = 4;
   private static final byte BOTH_RIGHT = 5;
   ```

3. Declare the state transition table.

```
private static final byte[][] NEXT_STATE = new byte[][] {
    new byte[] { LOST,       ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_LEFT,  ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_LEFT,  ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { LOST,       ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_RIGHT, ONE_LEFT, ONE_RIGHT, CENTERED },
    new byte[] { BOTH_RIGHT, ONE_LEFT, ONE_RIGHT, CENTERED },
};
```

4. Just prior to the start of the while loop, declare variables for the current state and the name of the current state.

```
int state = LOST;
String stateName = "Starting";
```

5. Just after the statements that sample the line sensors, add code to determine the condition index corresponding to the sampled readings.

```
int conditions = 0;
if (leftSample > THRESHOLD)
    conditions |= 0x2;
if (rightSample > THRESHOLD)
    conditions |= 0x1;
```

6. Following this, add a statement to determine the new state from the current state.

```
state = NEXT_STATE[state][conditions];
```

7. Finally, add a switch statement to update the stateName variable to match the new state.

```
switch (state) {
case BOTH_LEFT:
    stateName = "Both Left";
    break;
case ONE_LEFT:
    stateName = "One Left";
    break;
case CENTERED:
    stateName = "Centered";
    break;
case ONE_RIGHT:
    stateName = "One Right";
    break;
case BOTH_RIGHT:
    stateName = "Both Right";
    break;
case LOST:
    stateName = "Lost";
    break;
}
```

8. Finally, output the name of the new state on the LCD display.

```
display.print(1, stateName);
```

9. Build, load and test your program.

   Set the robot down with both sensors over the line. Start the program and verify that the second line of the LCD displays "Centered." Without lifting the robot up, gently slide it left so the left most line sensor is over the background while the right most line sensor is still over the line. Verify the LCD displays, "One Left." Slide the robot further left, so both sensors are over the background. Verify the LCD displays, "Both Left." Repeating the same procedure, slide the robot right, verifying the correct state names display. Finally, stop the program, place the robot with both sensors over the background, restart the program and verify the displayed state name is "Lost."

The final step in implementing your LineFollower2 program is to program the robot to perform the actions defined in Table 6-2. We will program the robot to steer slightly left or right by reducing the power to the left or right wheel, respectively. We will program it to steer hard left or hard right by turning the power off for the left or right wheel, respectively. Similar to the state table, the an action table can be created as another two dimensional array.

We will also modify your program such that it will begin performing actions after a second press of the START button. This will allow you to test the sensors and position the robot over the line before starting it moving.

Update your LineFollower2 program as follows:

1. Add import statements for PushButton, Motor and ContinuousRotationServo.

```
import com.ridgesoft.robotics.PushButton;
import com.ridgesoft.robotics.Motor;
import com.ridgesoft.robotics.ContinuousRotationServo;
```

2. Define constants for normal and low motor power.

```
private static final byte NORMAL = 10;
private static final byte LOW = 5;
```

3. Define constants to index the left and right elements of the action table, which you will create next.

```
private static final byte LEFT = 0;
private static final byte RIGHT = 1;
```

4. Use a two dimensional array to create an action table that contains the power settings for each motor to achieve the desired action for each state.

```
private static final byte[][] POWER = new byte[][] {
    new byte[] { 0,      0      }, // LOST
    new byte[] { NORMAL, 0      }, // BOTH_LEFT
```

```
        new byte[] { NORMAL, LOW    }, // ONE_LEFT
        new byte[] { NORMAL, NORMAL }, // CENTERED
        new byte[] { LOW,    NORMAL }, // ONE_RIGHT
        new byte[] { 0,      NORMAL }, // BOTH_RIGHT
};
```

5. Add statements to create motor objects.

```
Motor leftMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(1), false, 14);
Motor rightMotor =
    new ContinuousRotationServo(IntelliBrain.getServo(2), true, 14);
```

6. Add statements to obtain the START button object and wait for it to be released.

```
PushButton startButton = IntelliBrain.getStartButton();
startButton.waitReleased();
```

7. Just prior to the while loop, define a variable to keep track of whether the START button has been pressed a second time.

```
boolean go = false;
```

8. Following the statement that updates the state, add an if statement that provides branches to either update the motor power settings or to check if the START button has been pressed. Within the branch, update the power applied to each motor according to the action table.

```
if (go) {
    leftMotor.setPower(POWER[state][LEFT]);
    rightMotor.setPower(POWER[state][RIGHT]);
}
else if (startButton.isPressed()) {
    go = true;
}
```

9. Build, load and test your program.

   Place the robot on the line, start the program, press the START button a second time and verify the robot follows the line.

Summary

In this chapter, you learned how to work with analog sensors. You experimented with two photo reflective infrared sensors, which you used as line sensors. You used the IntelliBrain 2 controller's analog-to-digital converter to sample the sensors. This allowed your program to obtain an integer value proportional to the voltage on sensor's signal line.

You learned that the voltage of the signal output of a line sensor varies with the amount of infrared light reflected from the sensor's transmitter LED on to the sensor's receiving phototransistor. When the sensor is over a white background the voltage is low. When the sensor is over a black line the voltage is high.

Using the difference in sensor reading, you created a program that used one line sensor to provide feedback on the robot's position relative to the line. By implementing a proportional control algorithm you were able to program the robot to follow the edge of a line using just one sensor. Subsequent to this, you wrote another program to use both sensors to implement a state machine which also enabled the robot to follow the line using an entirely different control algorithm.

In the process of writing these programs you learned how to use constants and arrays. Constants give names to numeric values that you use in your programs, making your programs easier to read and maintain. Arrays provided a means for you to store and access tabular data in your program.

Exercises

1. Using your LineFollower1 program, set the robot down with the left line sensor over surfaces with various colors and textures. Record the surface characteristics and the associated sensor readings for several surfaces in Table 6-5. Using the chart in Figure 6-3, estimate the sensor's signal voltage for each reading and record it in the table.

2. Using your LineFollower1 program, take a number of readings varying the left line sensor's distance above a bright white surface. Record the distances and associated readings in Table 6-6. Using the chart in Figure 6-3, estimate the sensor's signal voltage for each reading and record it in the table. Plot the sensor reading verses distance from the surface.

3. Change your LineFollower1 program so the robot follows the right edge of the line rather than the left edge.

4. Modify your LineFollower1 program to allow the thumbwheel to be used to adjust the gain value without having to modify the program. Observe the robot's ability to follow the line using various gain values. What happens if you choose a gain value that is too small? What happens if you choose a gain value that is too high? What is the ideal gain value?

   Hint: Add statements to sample the thumbwheel, calculate the gain, and display its value within the first loop. Multiplying the sampled thumbwheel value by 0.00003 to obtain the gain value will allow for fine tuning the gain.

5. Using your LineFollower2 program, complete Table 6-7 by placing the robot on the line following course with the sensor positions indicated in the left two columns. Record the sensor readings and the state.

Table 6-5 – Line Sensor Readings for Various Surfaces

Surface Color	Surface Texture	Sensor Reading	Sensor Voltage

Table 6-6 – Line Sensor Readings Verses Distance

Distance	Sensor Reading	Sensor Voltage

Table 6-7 – Robot States

Sensor Position		Sensor Reading		State
Left	**Right**	**Left**	**Right**	
Over line	Over line			
Over line	Off line			
Off line	Over line			
Off line	Off line			