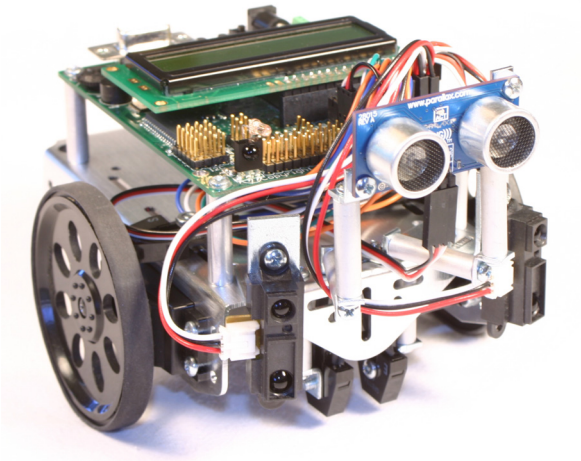
RidgeSoft™

Exploring Robotics with the IntelliBrain-Bot

An Introduction to Robotics and Java™ Programming



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

Second Edition, Revision 1

Table of Contents

CHAPTER 1

Introducing the IntelliBrain-Bot

You will be using the IntelliBrain™-Bot Deluxe educational robot throughout this book to learn about the emerging field of robotics. Your deluxe model IntelliBrain-Bot educational robot is a pre-designed mobile robot that will allow you to focus on robotics programming, using the RoboJDE™ Java™-enabled robotics software development environment. Before we get started programming your robot, we will first take a look at the mechanical and electronic components that make up your IntelliBrain-Bot educational robot.

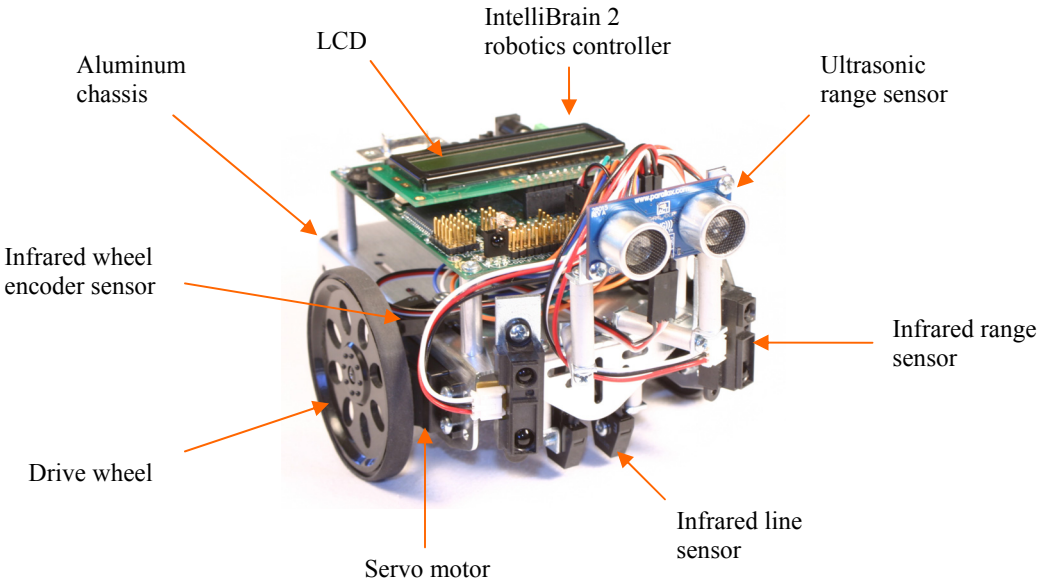


Figure 1-1 - IntelliBrain-Bot Deluxe Educational Robot

IntelliBrain-Bot Educational Robot Design

Figure 1-1 shows your fully assembled IntelliBrain-Bot Deluxe educational robot. As you can see in the figure, your 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

Your 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 your robot. Two motors mounted on the underside of the chassis drive the two large wheels, enabling your robot to move under its own power. A ball tail wheel supports the back end of your robot. The robotics controller, sensors, motors, tail wheel and battery holder mount directly on the chassis.

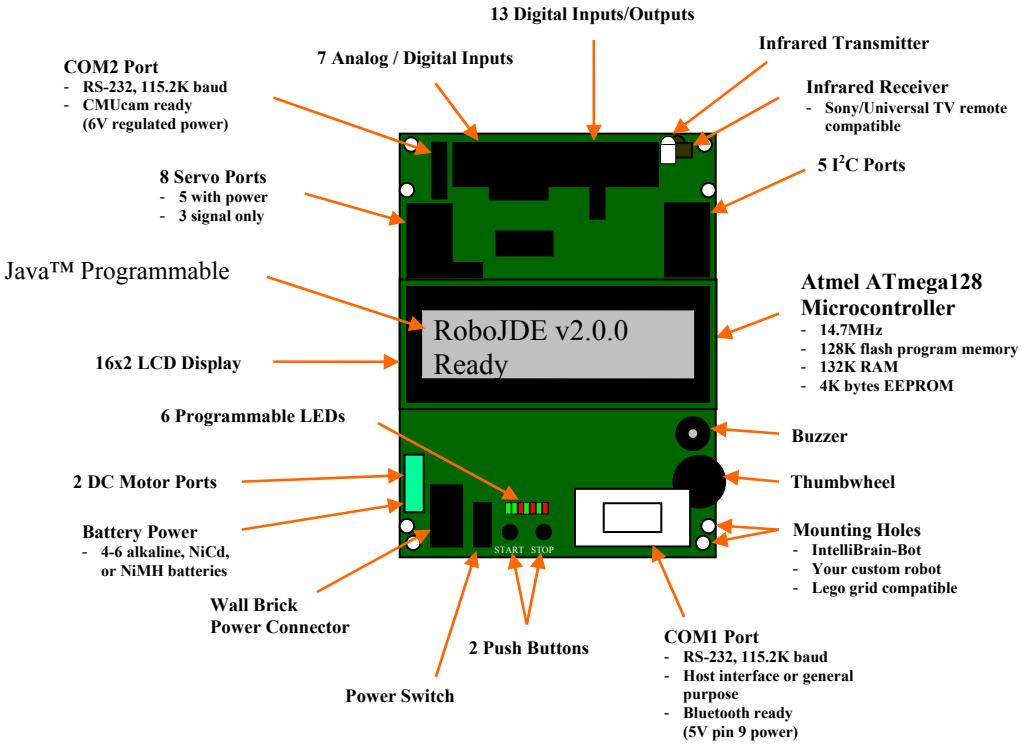


Figure 1-2 - IntelliBrain 2 Robotics Controller

IntelliBrain 2 Robotics Controller

The IntelliBrain 2 robotics controller, depicted in Figure 1-2, is your IntelliBrain-Bot educational robot's brain. A Java™ program executing on the IntelliBrain 2 robotics controller enables your robot to function independently, analyzing input from your robot's sensors, and controlling your robot's motors. By creating and loading different programs you can program your robot to perform a wide range of functions. Your programs may be as trivial as displaying your name on the LCD screen, or as complex as implementing algorithms that might be used in 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(s). The ports are arranged side-by-side across the forward edge of the controller board. Each port is marked by a label on the controller board that indicates the type and 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 are commonly used in small robots.
Motor	M1, M2	Motor ports directly power conventional DC motors using pulse width modulation (PWM) to vary the amount of power supplied to 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, your robot can communicate wirelessly with a host computer or other robots. The COM1 port also serves as the connection to your PC when you are programming your robot.

Port Type	Labels	Description
I^2C	I2C	I^2C (pronounced eye-two-see) 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 by software, not by the physical connection.
Infrared transmitter	none	The infrared transmitter transmits 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 that make it easy to program your 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. This provides 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 your program; however, your program can use them for other purposes.
Thumbwheel	The thumbwheel works like a conventional volume control knob on a radio. It provides a means for you to manually control a variable setting, such as the speed of a motor. You can also use it to scroll through screens of output displayed on the LCD screen.
Buzzer	You can program the buzzer to beep, click or play a tune.
Universal remote control receiver	The universal remote control receiver allows your program to take input from a universal (television) remote control, enabling you to remotely control your robot.
Light Emitting Diodes (LEDs)	Seven LEDs provide visual status indicators. Six of these can be turned on, off or blinked by your programs. 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 your programs. The LEDs are numbered 1 through 6, starting with the LED to the right of the power LED.

Sensors

Your IntelliBrain-Bot Deluxe educational robot has 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 your robot's wheels. These sensors use reflected infrared light to sense motion of the wheels.
Line Sensor (2)	Two line sensors mounted on the underside of your robot sense the presence of a non-reflective line. These sensors use reflected infrared light to enable your robot to sense and follow a line on the floor.
Infrared Range Sensor (2)	Two infrared range sensors sense the distance to objects. Their effective range is between 4 and 30 inches ahead of your robot.
Ultrasonic Range Sensor	One ultrasonic range sensor senses the distance to the nearest object ahead of your robot. Its effective range is between 1.2 and 118 inches ahead of your 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 stored in the four AA batteries installed in the battery holder, mounted on the underside of the chassis, powers your robot's motors and electronics.

A Hands-on Demonstration

Now for some fun, let's take a few minutes to see what your 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 is included with the example programs installed with the RoboJDE software development environment.

The demo program provides examples of your robot using various sensors to perform different behaviors. It also provides a simple user interface that demonstrates use of the IntelliBrain 2 robotics controller's human interface features, as well as using a 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 scroll through a list of pre-programmed functions to select a function you want your robot to demonstrate.

Use the following procedure to become familiar with scrolling through the functions provided by the demo program:

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 your robot.

2. Press the button labeled "START."

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

   ```
   Select Function
   Do Nothing
   ```

   The first line of text is telling you to select a function for your robot to perform. The second line displays the name of the current 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 available functions.

   You should see the following text displayed on the LCD screen:

   ```
   Select Function
   Play Tune
   ```

4. Press the STOP button repeatedly, scrolling through the available functions.

   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.

   Your robot will play Beethoven's *Ode to Joy* on the buzzer.

7. Switch the power switch to the off position.

Table 1-4 – IntelliBrain-Bot Demo Program Functions

Function	Description
Do Nothing	Your robot does not attempt to move. This allows you to test the sensors while your robot remains stationary. You will find this is extremely useful for debugging sensor problems.
Play Tune	Plays Beethoven's *Ode to Joy* on the buzzer.
Remote Control	Allows you to remotely control your robot using a Sony compatible infrared remote control. Use the channel up button to move your 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 your robot straight ahead 24 inches.
Rotate 180	Uses wheel encoder sensors and navigation classes to rotate your robot in place 180 degrees.
Navigate Square	Uses wheel encoder sensors and navigation classes to navigate your robot around a 16 inch square.
Random Dance	Uses software generated random numbers to perform a "dance" consisting of random movements.
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 then 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 that give you a glimpse into your robot's view of the world. This will allow you to verify that all of the sensors are functioning properly.

Use the following procedure to peer into your robot's brain and verify each sensor functions as expected:

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 does.

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. Manually 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 counter values maintained by the shaft encoders. Turn a wheel and observe the change in the counter value. The encoder will not sense changes in direction when you turn the wheel by hand. The counters will always count up (forward) when you turn the wheel by hand.
Pose	Displays the x and y coordinates and the heading of your robot's current position. Distances are in inches from your robot's starting point. The heading is in radians, with zero being the direction your 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 your 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 how the IntelliBrain 2 robotics 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 your robot down with the sensors over a bright white surface. For example, the white area of the line following poster (available from www.ridgesoft.com).

   Observe that both line sensors report a reading below 300.

10. Set your 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 your 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 over 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 your robot up such that there are no objects within four feet of your 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 reads 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 your robot and an object ahead of it.

1. Set your 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 your robot.

4. Slowly move your hand away from your robot.

   Observe your robot follows your hand forward.

5. Slowly move your hand toward your robot.

   Observe your robot backs away from your hand.

6. Switch the power off.

Navigation Demonstration

The demo program includes three functions which demonstrate your 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 your 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 your 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 your robot drives straight ahead for 2 feet, then stops.

3. Press STOP.

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

   Observe your robot turns in place approximately 180 degrees.

5. Press STOP.

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

7. Select the “Navigate Square” function.

   Observe your robot drives in a 16 inch square pattern.

8. Switch the power off.

Random Dance Demonstration

The “Random Dance” function of the demo program moves your robot in an endless sequence of small random moves. Because your robot has equal probability to move in any direction, it will not drift far from where it started as it performs this unusual dance.

1. Set your 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 your robot moves about randomly, but it 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 sensors is to avoid collisions with objects in your robot’s path. The “Avoid Obstacles” function of the demo program demonstrates how your robot can steer around obstacles in its way.

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

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

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

   Observe your robot will drive to a point 2 feet from its starting point, steering around the obstacle in its path, and will then return back to where it started.

4. Switch the power off.

Line Following Demonstration

The line sensors enable your 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 your robot on the floor over the black line.

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

Observe your robot follows the line.

3. Switch the power off.

Remote Control Demonstration

Your 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 your 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 your robot moves forward while you hold the button down.

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

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

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

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

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

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

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

8. Switch power off.

Summary

You should now be familiar with the features of your 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 your 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, recording the minimum and maximum reading you observe 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 your robot's brain by executing Java programs and interfacing to sensors, motors and humans.
1	Aluminum chassis	
2	Servo motor	
2		Covert torque and rotation of the motor shafts to force and linear motion.
1	Ball tail wheel	
	Tires	Provide traction (friction) so the wheels don't slip.
2		Uses reflected infrared light to enable your robot to sense and follow a line on the floor.
2		Uses reflected infrared light to enable your robot to sense the distance to an object between 4 and 30 inches away.
1		Measures the time between issuing a sound pulse and sensing its echo to measure the distance to an object between 1.8 and 118 inches away.
2		Uses reflected infrared light to sense rotation of a wheel, enabling your 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 Your IntelliBrain-Bot

In the previous chapter you became familiar with the hardware features of your IntelliBrain™-Bot Deluxe educational robot. You also observed your robot in action by working with the demo program. In this chapter you will begin to learn about the software features of your 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 your project's properties. The RoboJDE development environment uses project files to make it easy for you to switch between different projects.

Use the following procedure to create a project:

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.

   You can create a new folder by browsing to the location where you want to create a new folder and 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.

   Your MyBot project and Java class file will be created, as shown in Figure 2-1.

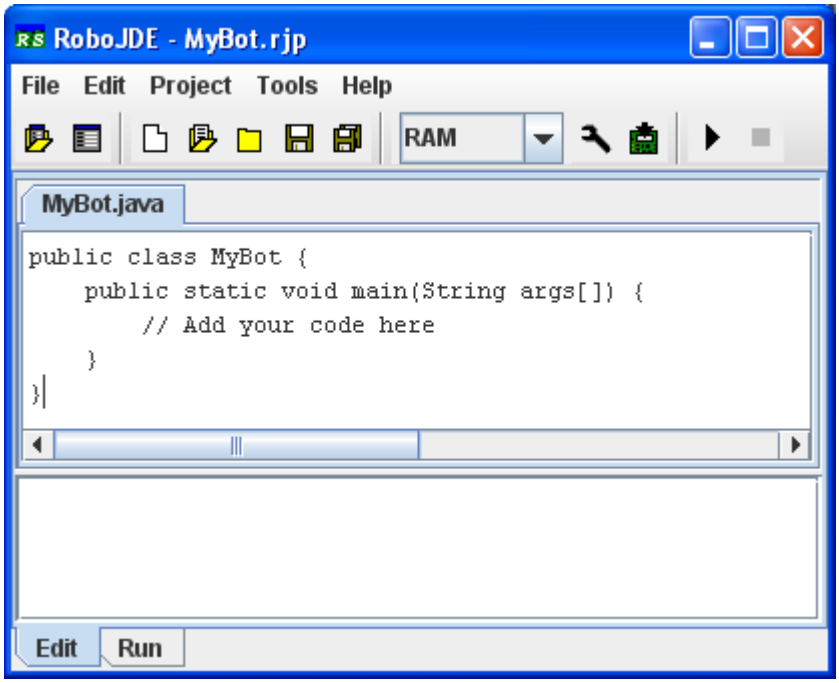


Figure 2-1 – New MyBot Project Viewed in the RoboJDE GUI

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

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

   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.

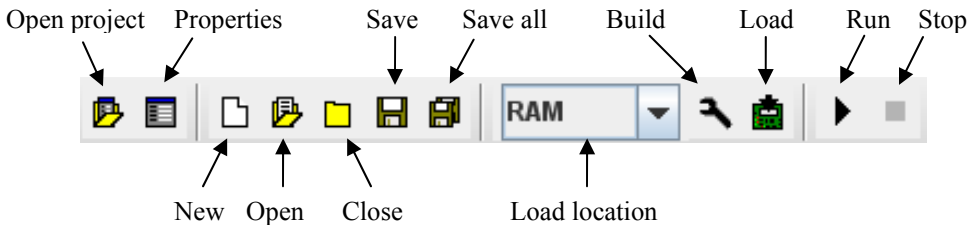


Figure 2-2 - RoboJDE Tool Bar

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

Connecting Your Robot to Your Computer

The RoboJDE development environment communicates with your 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 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 the RoboJDE development environment. However, if this is not the case, please consult the *IntelliBrain 2 User Guide* and *RoboJDE User Guide* for further instructions.

Use the following procedure to connect your robot to your computer:

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 your robot the load button will appear gray. Since your 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.

   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 your

robot is quite sturdy, be careful not to apply excessive force because you may damage your robot.

3. Switch the power on.

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

   If the Load button did not turn green, request help from your lab instructor to ensure the settings (Tools->Settings) in RoboJDE development environment 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 both in 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 on your computer you are using. 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 your robot will display the following message on the second line:

   ```
   Ready - RAM
   ```

   This indicates there is now a program loaded into Random Access Memory (RAM) that 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, one after the other. The computer executes one step and then proceeds on to the next step, executing each step before proceeding to the next step, and so on, until computer reaches the end of your program.

You can think of a program similar to how you think of a recipe. A computer executes a program similar to how you execute the steps of a recipe – starting at the beginning, executing steps, one after another, until you have completed the recipe.

The Method Named "main"

In the case of your 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 your program. 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. This 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 so it will wait for the STOP button to be pressed. This will cause your program to display your name and then wait for you to press the STOP button before it exits.

Use the following procedure to extend your program:

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

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

   This tells the 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 your robot.

   Observe that 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 you to press the STOP button.

The Programming Process

As you develop programs for your robot you will become very familiar with the programming process, which is illustrated in Figure 2-3. You will use this process over

and over to program your 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 your 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 your robot).

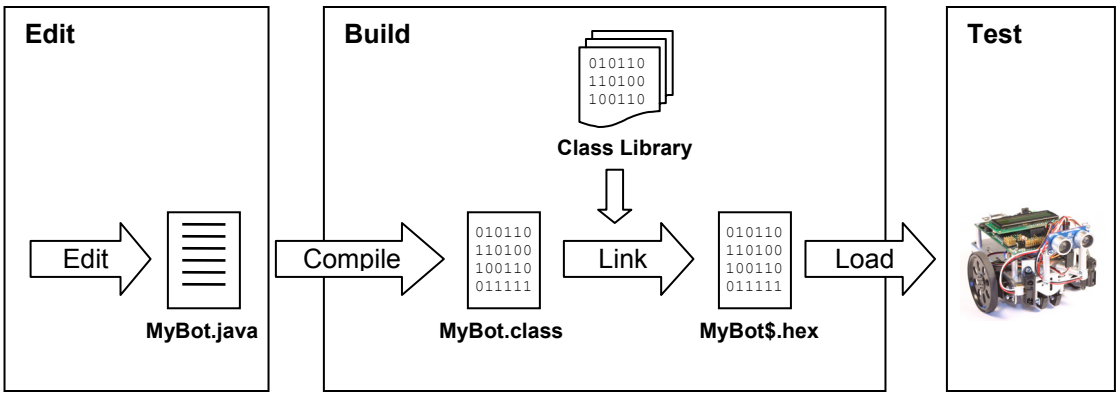


Figure 2-3 – The IntelliBrain-Bot Educational Robot Programming Process

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 his or her project is completed. Making and testing changes in small increments 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** – your 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 your project's properties

The text you entered into the RoboJDE edit window to define your 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. Your MyBot program was compiled, linked and loaded, as depicted in the block labeled "Build" in Figure 2-3. In the first step, your 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 that 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, producing binary "byte codes," which the RoboJDE virtual machine can execute. The virtual machine resides on the IntelliBrain 2 robotics controller, enabling it to execute Java byte codes.

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 your 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 your 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 the problems you encounter. It will make you a better programmer and you will find programming more enjoyable!

Compilation Errors

The compiler converts the text you enter, which is called "source code," into Java byte codes, which the RoboJDE virtual machine can execute. 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.

Use the following procedure to experiment and see what the compiler does when you introduce errors into your program:

1. Edit the fifth line of your program to replace the period between "out" and "println" with a comma as follows:

```
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, with the number of the

line displayed on the left. The third line indicates the location of the error in the problem line by 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, which is 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.

   The RoboJDE editor 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 find it convenient to jump right to the line rather than having to scroll down looking for it.

5. Correct the error and click the build button.

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 to 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, 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." Use the following procedure to make a small change in your program that will introduce a bug:

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

```
private static String myName;
```

   This line creates a field to keep track of your name.

2. Modify the print statement in your program, replacing the quoted string containing your name with "myName," the new field you just declared.

```
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. It shows that 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 (null 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 println method called the print method. This is also in the PrintStream class. The third line of the stack trace indicates line 5 of your 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 then 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 call the println method, as the stack trace indicated. This is a line you just

modified. The NullPointerException must be due to this change.

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

8. Correct the bug by initializing the field with a text string containing your name.

```
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 brain 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 correctly and functioning properly.

Use the following procedure to add a print statement to your program indicating 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 that you expect. 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 2 robotics controller's Light Emitting Diodes (LEDs) and its buzzer. These provide additional means of indicating what your program is doing.

Summary

You have created your first program and made a number of modifications to it. You are now familiar with the process of programming your 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 your 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 Your IntelliBrain-Bot

Now that you are familiar with how to create, build and test programs for your IntelliBrain™-Bot Deluxe educational robot, you are probably eager to program your robot to do something more than just display your name. In this chapter you will learn how to program your robot to maneuver. In addition, you will also learn how to use the RoboJDE™ Java™-enabled robotics software development environment API documentation and other documentation to help you accomplish your programming tasks. Furthermore, you will learn about a few more features of the Java™ programming language.

Differential Drive Robots

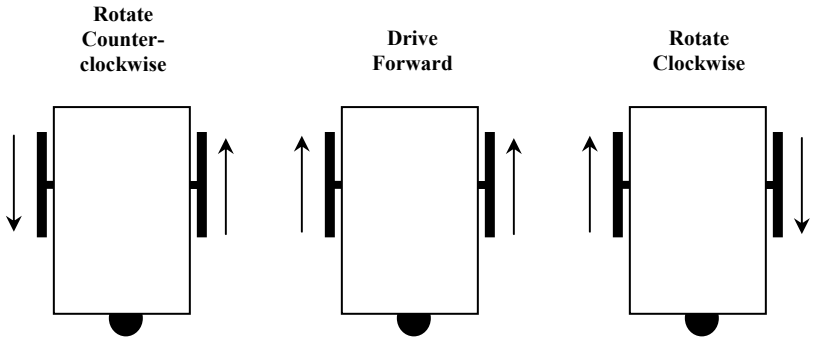


Figure 3-1 - Maneuvering a Differential Drive Robot

Your IntelliBrain-Bot educational robot uses a differential-drive system to move and steer. Its name may sound complicated, but differential-drive is easy to understand; there are two wheels, powered by two independently controlled motors. Your programs will control the speed and direction of your robot by controlling the power applied to each motor. As illustrated in Figure 3-1, you can program your robot to perform a few basic maneuvers simply by controlling the direction of rotation of each motor. Your robot will move forward when your program applies the same amount of power to both motors. Your 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 your robot to rotate counter-clockwise.

Likewise, applying forward power to the left wheel and reverse power to the right wheel will cause your robot to rotate clockwise.

Programming Simple Maneuvers

Your 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. The motors must be able to rotate continuously to drive your robot.

Robotics researchers discovered hobby servers could be easily modified for continuous rotation to provide inexpensive motors for their robots. They modified the servos by removing the mechanical stops and disabling the position sensing circuitry in the servos. Fortunately, the use of servos has become so common for robots that you can now buy servos manufactured for continuous rotation, eliminating the need to modify them yourself. The servos on your 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 servo 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 the control signals of the servos, the RoboJDE class library provides the ContinuousRotationServo class. This class provides a wrapper that allows your program to control a servo as if it were a conventional motor.

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 will help you understand how to program your 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 the "docs" folder in the RoboJDE installation folder. It is also available online at www.ridgesoft.com. Take a moment to locate the quick reference document. Study the 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 your 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 your IntelliBrain-Bot educational robot. The *RoboJDE API Documentation* is in Javadoc format. This is the format used to document most Java APIs. Becoming proficient with using the RoboJDE API documentation will also make you more proficient when using similar documentation for other Java programming projects.

Display the API documentation by clicking on the RoboJDE API Documentation item in the RoboJDE program group on the start menu, or by clicking on the index.html file in the apidoc folder in the RoboJDE installation folder. This will launch your web browser, which will display the documentation, as shown in Figure 3-2.

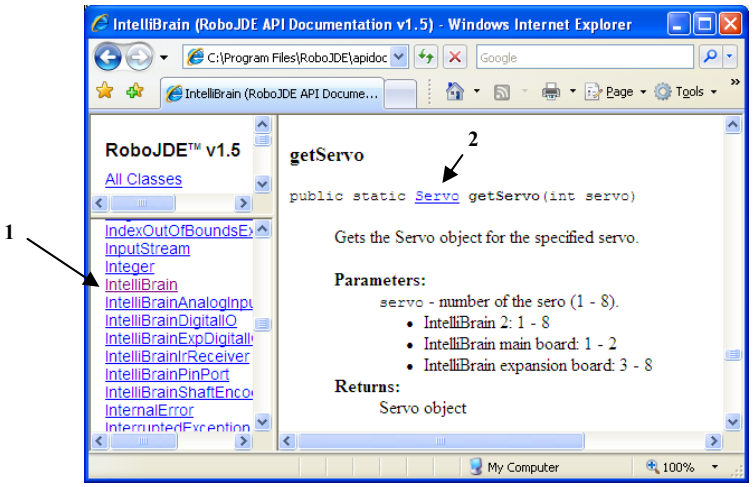


Figure 3-2 - IntelliBrain API Documentation

The documentation for the IntelliBrain class is a good 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 3-2. This will display documentation for the selected 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 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 Your Robot to Drive Forward

You should now have a general idea of how you can write a program to maneuver your robot. Your 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 your robot to drive forward.

Your 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 object wrappers, which will give them Motor interfaces. This will enable your program to control the motors using the setPower method.

Completing the following steps to program your 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;
```

   Import statements refer to pre-built classes your program "imports" from the class library. All of the classes described in the API documentation are stored in the RoboJDE class library. 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 a 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 by your 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 your robot and switch power on.

8. Build and download your program to your robot.

9. Carefully disconnect the serial cable without turning your robot's power off.

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

11. Press the START button.

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

Programming Your Robot to Rotate in Place

Referring back to Figure 3-1, your program can make your robot rotate in place by applying power such that one wheel rotates forward and the other wheel rotates backwards.

Complete the following steps to make your 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 your robot and switch power on.

3. Build and download your program to your robot.

4. Disconnect the serial cable.

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

6. Press the START button.

   Observe your robot will rotate 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 your robot will rotate clockwise.

Programming Your Robot to Drive in a Circle

Your first guess might be that it has got to be very difficult to program your 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. Your robot will follow a curved path, which will, more or less, form a circle.

Program your 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 your robot will drive in a circle.

Combining Simple Maneuvers

You can program your 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 your 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 again.

Modify your Maneuver program to program your 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 API documentation for the sleep method of the Thread class. This method allows your program to sleep for the number of milliseconds you specify. 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 you 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. Specify your program should sleep for 2000 milliseconds (2 seconds).

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

   The sleep method of the Thread class causes the program to stop execution (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 your robot to stop once your program wakes up from its sleep.

5. Download and run your program.

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

In a similar fashion, you can program your 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 your 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 your 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 local variables leftMotor and rightMotor into fields 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 to be fields 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 fields 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 fields 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 that initializes the right motor. Follow this by declaring the goForward

method, including the remainder of the statements in your program in the new method.

```
        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 your robot and run it.

   Observe your 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 that 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 your robot to rotate more slowly, making it easier to calibrate the angle of rotation.

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

13. Create a stop method following the rotate90 method that will stop both motors when called.

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

14. Add two steps to make rotate90 and stop the final steps of the main method.

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

15. Download and run your program.

    Your 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 calibrated the sleep time such that your robot rotates about 90 degrees, proceed to the next step.

16. Copy the goForward and rotate90 steps at the end of the main method and paste another three copies of these two lines, so there are a total of four copies when you are done.

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

17. Download and run your program.

    Your robot will drive in an approximate square. Adjust the sleep time in the rotate90 method to fine tune your 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,

making your program easier to maintain, 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. Java provides three types of loops: while loops, do-while loops and for loops.

Suppose you would like to program your 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, ensuring there were enough copies that your program would not terminate before you lost patience watching your robot drive in a square. Clearly this approach is awkward. Using a loop is a better approach.

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.

   Your robot will now drive in a square until you stop it.

   You will most likely notice the corners of the square drift as your robot continues to drive around the square. This occurs because your 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 created a loop so your robot will drive indefinitely around a square, but what if you want your robot to drive just once around 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 iteration of the loop. If the condition is true, the computer will continue iterating the loop. If it is false, it will “fall” out of the loop and begin executing the instructions that follow 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++;
   ```

   The ++ operator increments the associated variable. The -- 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.

   Your robot will now drive once around the square.

Do-While and For Loops

The Java language also supports do-while loops and for loops. The do-while loop checks the condition at the end of loop rather than the beginning. This is useful if the 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 your robot to turn both wheels in the same direction, it will go forward. By programming your robot to turn its wheels in

opposite directions, it will rotate in place. You can program your robot to perform other maneuvers, such as driving in a circle, by applying different amounts of power to each motor.

You can program your 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 your robot, you will observe that using time as the basis for navigation has its limitations. As the batteries drain, the behavior of your robot will change. Your robot's performance will also vary depending on the surface it operates on. The difference in friction of different surfaces will have a significant effect on how your robot performs. Unfortunately, as your program is currently implemented, it provides no mechanism for your robot to account for variations in conditions such as battery charge and friction. Your 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. In later chapters, you will 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*. 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, fields, local 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 your IntelliBrain-Bot connect to?

3. Browse to the documentation on the Motor interface in the class library API documentation. What methods does this interface provide? 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. Write a short description of each method 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 your robot to drive straight ahead for exactly 12 inches. Using the same program, run your 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 your robot to rotate exactly 180 degrees. Using the same program, run your 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 your 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 your 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 your 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 Your IntelliBrain-Bot

In this chapter you we will investigate the human interface features of your IntelliBrain™-Bot Deluxe educational robot. You will learn how your programs can use these features to enable your robot to interact with people.

Using Text Output

One way for your 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 be using to output text to the LCD screen:

1. Display the RoboJDE™ Java™-enabled robotics software development environment class library API documentation.

2. Navigate to the documentation for the IntelliBrain class.

3. Navigate to and review documentation for the getLcdDisplay method.

4. Click on the link for the Display class.

5. Review the documentation for the Display class, taking note of the methods that are available to display text.

Now, use what you have learned from the documentation to create a simple program to display the name of your program on the first line of the LCD screen. You will need to begin by creating a new project for your program:

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 field 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 field 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.

   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 will quickly display its name on the first line of the LCD screen before exiting.

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; 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 as an indicator of whether power is on or off. 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: the green LED

to the right of the power switch. Switch the power on and off. Observe the LED turns on and off, as well.

The power LED is hardwired into the circuit of the IntelliBrain 2 robotics controller. Your programs cannot control this LED; however, the six other LEDs to the right of the power LED can be controlled by your programs.

The first two LEDs to the right of the power LED are the status and the fault LEDs, respectively. These LEDs are 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, to the right, are user LEDs. These LEDs are fully under the control of your programs. You will become familiar with how these LEDs can be used to enable your robot to interact with humans by extending your program to use them.

In the previous chapter, you wrote a program that enabled your robot to perform various maneuvers. You could see your robot performing particular maneuvers, but 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 your 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 fields 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, as well as 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 – that 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.

   Your 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 as your robot

maneuvers. We'll use the four "user" LEDs on the IntelliBrain 2 robotics controller. These are the four right most LEDs, just 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 status 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 fields 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);
```

You must now extend your program to turn the LEDs on and off as it adjusts the power it applies to the motors. Whenever your program sets the motor power to a value greater than 0, it will need to 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 need to 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 need to 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. If-statements have the following form:

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

The statements within the parenthesis consist of a block of statements 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 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. If you do include an else clause with no condition, it must be at the end, and you can only include one such clause.

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 your robot drives. As your robot goes straight ahead the two green LEDs should be illuminated and the two red LEDs should be off. When your robot turns, the right motor's green LED should be off and the red LED should be on.

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 your 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 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 to use the thumbwheel to vary the size of the square your 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 field 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 your program calls 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 robot drives 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 your 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 value of the expression on the right hand side of the operator to the variable on the left hand side of the operator. For example, the statement:

```
a = 2;
```

sets the value of 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. Multiply and divide are denoted using the symbols * and / instead of the standard math symbols. The following code snippet provides examples of the arithmetic operations. The comments to the right list the values 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 language. 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 your 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 field 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 wait 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 inverts the value returned by the isPressed method. 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. Your robot will maneuver a square of the 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. Boolean variables have only two possible values: true or false. The isPressed method 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 Your 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 your robot a second trick, dancing.

Rather than programming your robot to perform a highly choreographed dance, we'll keep it simple. We'll teach your 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 random numbers 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 your 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 value, which randomizes the random number generator. Otherwise, the dance would consist of the same sequence of pseudo random moves each time you run your program.

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

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

```
private static PushButton stopButton;
```

2. Initialize the stopButton field 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 local 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 the trick local variable 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 your 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 your robot should maneuver in a 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 local 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 correct case 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 the trick local variable 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 should 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 2 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 field 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 your 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 can 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 *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 your 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 your robot from a distance. The remote control does this by transmitting pulses of infrared light, which your 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.

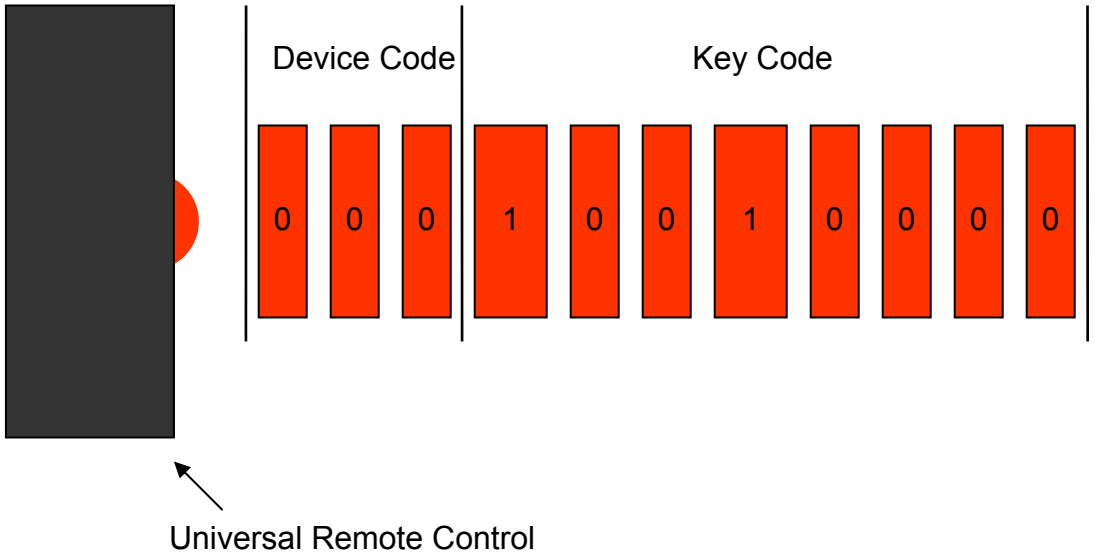


Figure 4-1 - Sony Infrared Remote Control Data Transmission

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 field 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 local 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 the trick local variable 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 at 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 your robot to begin a trick.

Maneuvering by Remote Control

As a final trick, we will program your 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 your robot to continue the operation. Releasing the button will cause your 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 your robot whenever no key codes are being received from your 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 your 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 your IntelliBrain-Bot educational robot to allow you to select one of four tricks for your robot to perform. You also learned how to program your 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 your 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 creates 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 to 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 Parallax Ping))) ™ ultrasonic range sensor and then use it to create a "tractor beam." Your IntelliBrain™-Bot Deluxe educational robot will be able to sense and follow your hand, as if there were an invisible beam attaching your robot to your hand.

Sonar Range Sensing

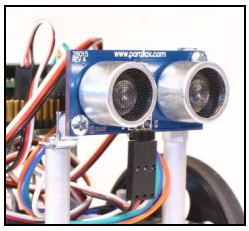


Figure 5-1 – Ping))) Sonar Range Sensor

Your IntelliBrain-Bot 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 your robot by "pinging." This sensor detects objects by generating a short high

frequency sound, and then listening for an echo. The sensor will hear an echo if there is an object in front of your 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.

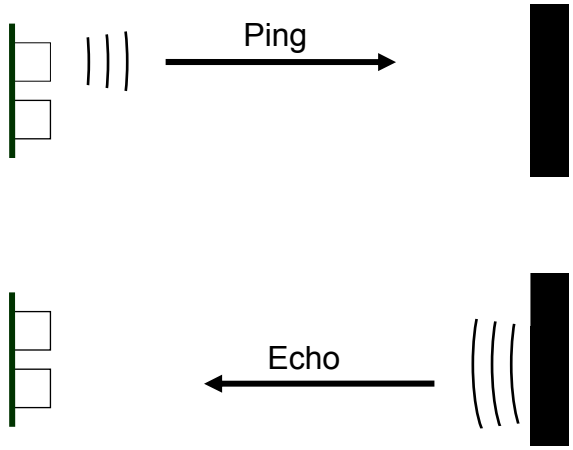


Figure 5-2 – Sonar Ping and Echo

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™ Java™-enabled robotics software development environment 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 IntelliBrain 2 robotics controller ports that provides 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 can trigger the Ping))) sensor to send a sound pulse by quickly switching the IO3 port signal on and off. You can imagine this as if you were flipping a light switch on and off very quickly; however, 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 the pulse can't be generated by software; the microcontroller chip on the IntelliBrain 2 robotics controller must generate it.

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 to its 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 Ping))) sensor.

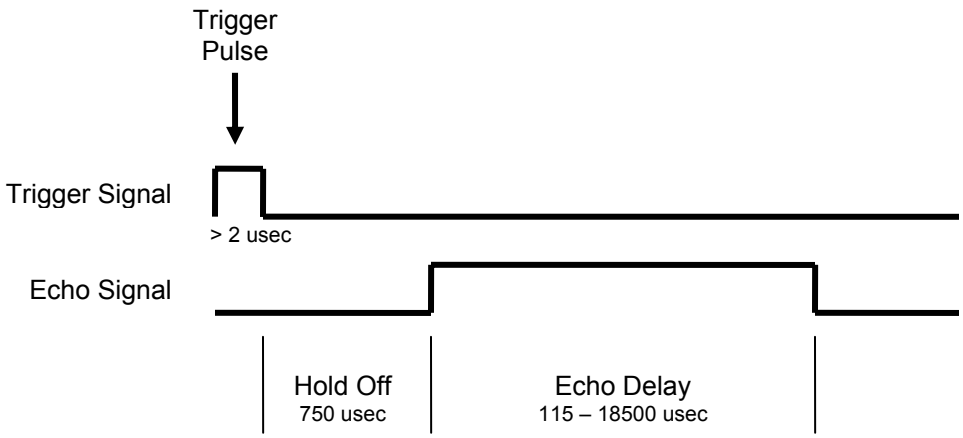


Figure 5-3 –Ping))) Sensor Signals

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.

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

5. Add statements within the try statement to get the display object and print the name of your 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 your 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 your program takes two readings per second.

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

12. Build, load and test your program.

    Hold your hand steady in front of the sensor. Note the reading, then move your hand closer or further away, noting 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 your program running, hold your 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 yields:

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 local 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. Verify 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.

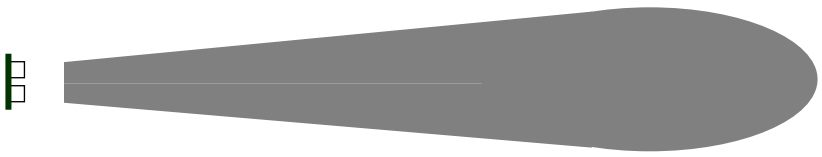


Figure 5-4 – Effective Sensing Region of Ping))) Sensor

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.

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

Surprisingly, creating this program is not nearly as challenging as it might seem. All that your program needs to do is use the Ping))) sensor to repeatedly measure the distance to the nearest object, adjusting the power to the motors with each new measurement. If the distance is too large, your program will need to apply power to the motors to move your robot forward. If the distance is too small, your program will need to apply power to the motors to move your robot backwards. If the distance is just right, your program will need to turn power to the motors off.

Create your program using the following procedure:

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 for the library classes your 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 then print the name of your program.

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

6. Create motor objects for the two motors.

```
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 your program.

    Your 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

The only data type we have used so far to declare integer variables is int. 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 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 time 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 cost of extra computation time.

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 your robot's memory. The RoboJDE virtual machine does not fully support the double data type. The double data type provides the same range and precision as the float data type, but it takes up 64 bits of memory instead of 32 bits used by a float variable. 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 know which is best for a particular variable?

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 as 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." You specify the cast operation by enclosing the target data type 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 be 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, your program must power the motors forward. If the distance to the object is less than desired, your program must power the motors backward. Otherwise, your program must turn the motors 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 your 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 your 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 your robot, power the motors in reverse if there is an object within 5.5 inches of your robot, or stop the motors if there is an object between 5.5 and 6.5 inches of your 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 your robot should move backwards. If your hand is further than 6.5 inches your robot should move forward. Notice that your robot is a bit jerky when it starts and stops. This is because of the way your program is controlling the motors. In the next exercise we will make your 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 a difference (error) between the actual and 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 your robot. Your program attempts to eliminate the error by powering the motors in the direction that will reduce the error. Your program currently 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. Your robot will frequently overshoot the desired position because the motors are either on at full power or turned off. This approach is often called "bang-bang" control because your program bangs the power from one extreme to the other: full power or no power. You can imagine 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 your 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, your program applies a small amount of power to the wheels. If the error is large, your program applies a large amount of power to the wheels. As the error decreases, your program decreases the power. If there is no error, your program turns the motors off. You can accomplish this by making the power proportional to error, as follows:

power = error * gain

The gain value 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 your 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 your robot is exactly six inches away; therefore, the difference between the actual range and the desired range is the error:

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 that the setPower method limits the actual power to the maximum or minimum if the value of the power argument is out of this range.

2. Build, load and test your program.

   Your 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 your 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 your robot to strive to maintain a fixed distance, six inches, from your hand. You used a proportional control algorithm to vary the power to your 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 initially 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 your robot in your hands facing 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. Observe your robot's response to movements of your hand. How does your robot's behavior change if you set the gain constant to 1.0? How does your robot's behavior change if you set the gain constant to 20.0?

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

CHAPTER 6

Line Following

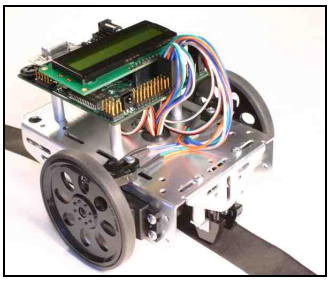


Figure 6-1 - IntelliBrain-Bot Educational Robot Following a Line

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

Line Sensing

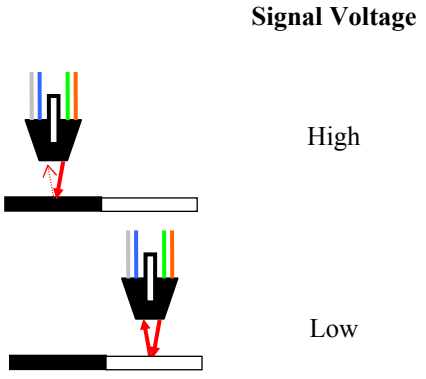


Figure 6-2 - Sensing a Line

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 of the sensor's signal. Your program can do this by using one of the IntelliBrain 2 robotics controller's built-in Analog-to-Digital converter ports to measure the voltage at the signal pin. The IntelliBrain 2 robotics 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. 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 by using the port object's sample method, as follows:

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

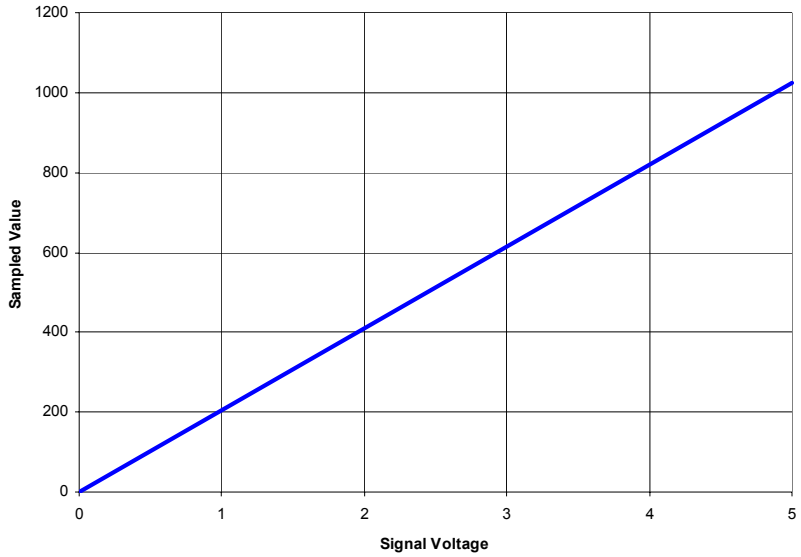


Figure 6-3 - Sampled Value verses Signal Voltage

The sample method uses the IntelliBrain 2 robotics controller's A-to-D converter to measure the analog voltage at the port's signal pin and converts 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 your 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 then 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 your 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 (available from www.ridgesoft.com), 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 your 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 the previous step with the sensor over the background and then with the sensor over the edge between the line and the background.

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 your robot relative to the line. You could program your robot to drive straight ahead when the sensor is over the middle of the line; however, when your robot drifts off the line, your program will have no way to know if it is to the left of the line or to the right of the line. Therefore, your program would have no way of knowing whether to steer left or right to get back on course.

You can solve this problem by programming your robot to follow the edge of the line, rather than the middle of the line. By following the edge of the line, your program will know whether your robot has drifted to the left or to the right of the edge – the sensor will

read low when your robot has drifted away from the edge (over the white background) and it will read high when your robot has drifted over the center of the line (over the black line). You can program your robot to follow either the left or right edge of the line, but 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 your 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 your robot to the right. It can do this by applying more power to the left wheel than to the right wheel. When your robot drifts right, the sensor will move over the black line and the sensor will read high. Your program will need to steer your robot left by applying more power to the right wheel than to the left wheel.

We can use the proportional control technique you learned about in the previous chapter to correct for an error in your robot's position relative to the line. In this case, your program will need to steer your robot left or right depending on the error it senses in your 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 your 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, and then using the offset to offset the power applied to each wheel. You can accomplish this by adding the offset to the power applied to one motor, and subtracting it from the power applied to the other motor. This will cause your robot to steer back toward the edge of the line. The greater the error, the more aggressive your program will be in steering your robot back toward the edge of the line.

You can calculate the power offset as follows:

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

Your program will steer your 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 your robot moves forward. The offset will cause your robot to steer left or right as it proceeds forward.

Use the following procedure to extend your LineFollower1 program to make your robot follow a line:

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

7. Add a statement to your program to declare and initialize a local 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 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. In the go branch of the if statement within the second loop and after the statement that samples the sensor, insert the proportional control equation.

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

9. Following the control calculation, set the power to each motor, adding the offset to the base power setting for the left motor and subtracting it from the base power 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 your robot down with the left line sensor over the left edge of the line. Press the start button once to start your program and a second time to start your robot following the line. Observe your robot's ability to follow the line. If your robot is sluggish in responding and drifts away from the line or over the line, try increasing the gain constant about 10%. If your 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 your robot to follow a line. With two line sensors your program can detect if your robot has drifted to the left, drifted to the right or remains centered over the line. In contrast, with only one sensor your program could only determine if the sensor was over the line or not. Your program had no way to know if your robot was left or right of the center of the line; it had to follow one edge of the line and couldn't recover if your robot drifted all the way across the line. With two sensors, your program can sense which side of the line your robot has drifted to, allowing it to correct when your robot drifts past either edge of the line.

Using a Finite State Machine

By considering the position of the two sensors relative to the line, at any point in time your robot will be in one of the six states listed in Table 6-2. The positions of the sensors relative to the line define each state. The actions listed in the table define the action your program will need to take in order to make your robot follow the line. For example, when both sensors are over the line, your program will need to steer your robot straight ahead. If your 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 your robot drifts so far left that both sensors are left of the line, your program will need to steer

your robot hard to the right to get back to the line. Similarly, your program will need to steer left when your robot drifts right.

The Lost state in Table 6-2 is necessary to account for the possibility that your program won't know if your robot is left or right of the line. This can occur when your program is started when both sensors are off the line. It can also occur if your robot drifts off the line so fast that your program misses whether your robot was drifting left or drifting right. Finally, it can occur if your robot drives off the end of the line. In all of these cases, your robot has gotten lost. Therefore, your program should stop your 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 daunting set of bubbles and connecting arrows, but once you understand the notation it is not hard to comprehend.

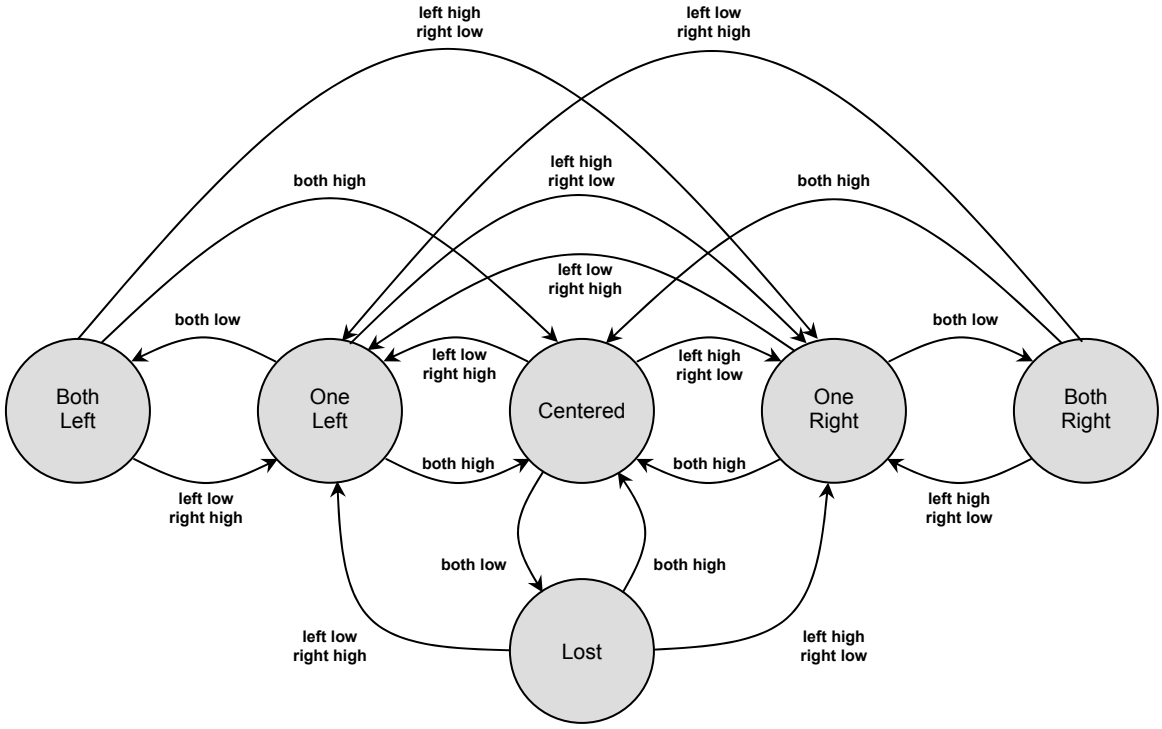


Figure 6-4 - Line Sensor State Diagram

Notice that the six states, which 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 that 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 any time 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, which indicates both sensors are over the line and, therefore, your 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 each state. These transitions occur when the sensor readings aren't the expected readings for that particular state. There are four permutations of sensor readings: (low, low), (low, high), (high, low) and (high, high). For each state, one permutation corresponds to the expected readings for that state. The other three permutations indicate your robot has transitioned to a different state. Therefore, for each state, there are three arrows indicating a transition to another state.

Note that three of the permutations of sensor readings correspond to exactly one state: (low, high) – One Left; (high, high) – Centered; and (high, low) – One Right. Each of the transitions into these states corresponds to the same pair of sensor readings.

Both sensors reading low may indicate any one of three states: Both Left, Both Right or Lost. In this case, the previous state determines the new state. If your robot was drifting left (One Left) when both sensors began reading low, it indicates your robot has drifted further left of the line and entered the Both Left state. Likewise, if your robot was drifting right (One Right) when both sensors began reading low, it indicates your 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 your robot drifted left, right or ran off the end of the line. Therefore, your robot has transitioned to the Lost state.

Table 6-3 - State Transition Table

	Conditions (left sensor, right sensor)			
Current State	**(low, low)**	**(low, high)**	**(high, low)**	**(high, 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 implement in your 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
}
```

Use the following procedure to 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 your robot over the line and observe the sensor readings. Change your robot's position to correspond to each of the four possible combinations of sensor conditions: (high, high) – centered over the line, (low, low) – entirely off the line, (low, high) – slightly left of center and (high, 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 fields you declare in your program that have a value that does not change. You initialize the field 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 fields declared with the "final" keyword. This keyword specifies the values are final and cannot be changed when your program is running. The Java compiler will report an error if you include a statement in your program that attempts to assign a new value to a field you have declared final.

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 an array of bytes in your program using the following statement:

```
byte[] nextState =
      new byte[] { LOST, ONE_LEFT, ONE_RIGHT, CENTERED };
```

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 the element's index. For example:

```
state = nextState[2];
```

After executing this statement, the value of the state variable would be equal to 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 local variable named 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 local 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. Add a switch statement to update the stateName local 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 your robot down with both sensors over the line. Start your program and verify that the second line of the LCD displays "Centered." Without lifting your 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 your robot further left, so both sensors are over the background. Verify the LCD displays, "Both Left." Repeating the same procedure, slide your robot right, verifying the correct state names display. Finally, stop your program, place your robot with both sensors over the background, restart your program and verify the displayed state name is "Lost."

The final step in implementing your LineFollower2 program is to program your robot to perform the actions defined in Table 6-2. We will program your 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, 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 your robot over the line prior to commanding it to begin following the line.

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 then wait for it to be released.

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

7. Just prior to the while loop, define a local 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 your robot over the line, start your program, press the START button a second time and verify your 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 robotics 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 your robot's position relative to the line. By implementing a proportional control algorithm, you were able to program your robot to follow the edge of a line using just one sensor. Subsequently, you wrote another program to use both sensors to implement a state machine which also enabled your 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 your 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 your 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 your program. Observe your 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 your 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			

CHAPTER 7

Infrared Range Sensing

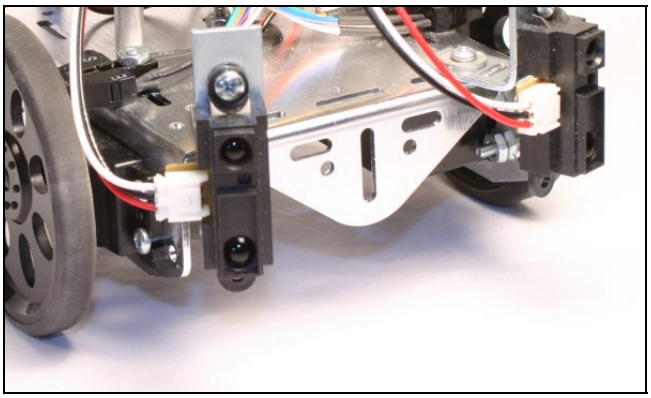


Figure 7-1 - Sharp Infrared Range Sensors

Two Sharp infrared range sensors enable your IntelliBrain™-Bot Deluxe educational robot to sense objects in its path. In this chapter, you will learn how infrared range sensors work. You will then develop a program to make your robot wander around a room using its infrared range sensors to avoid bumping into things.

Understanding Infrared Range Sensors

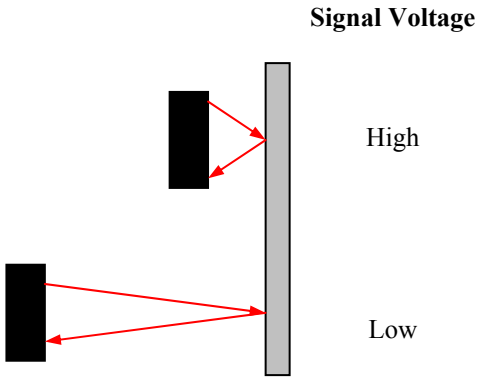


Figure 7-2 - Sensing Distance to an Object

Each infrared range sensor measures the distance to an object by detecting reflected infrared light transmitted by its light emitter. This is illustrated in Figure 7-2. The electronics in the sensor enable it to measure the angle at which the reflected light enters the detector. When the sensor is close to an object, light enters the detector at a sharp angle. When the sensor is far from an object, light enters the detector at a slight angle. The sensor outputs an analog voltage that varies depending on the angle at which the reflected light enters the detector. This technique makes the sensor insensitive to ambient light and the reflectivity of the detected object, ensuring the output voltage is solely a function of the distance to the detected object. Your programs can determine the distance to the nearest object by sampling the output voltage of the sensor.

Figure 7-3 is a graph of the sensor output voltage verses distance.

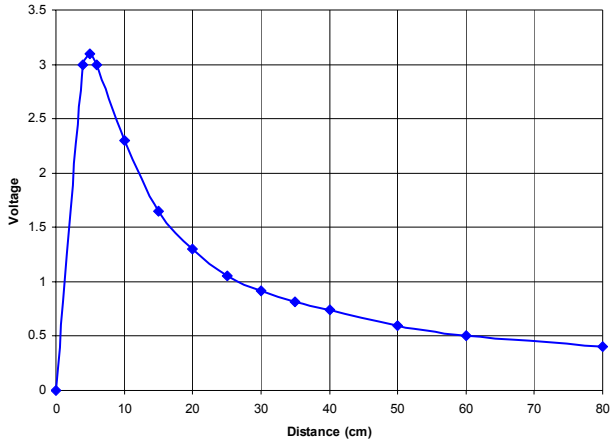


Figure 7-3 - Output Voltage verses Distance for a Sharp Range Sensor

Using the Infrared Range Sensors

When working with any sensor, a good starting point is to write a small program that reads and displays the sensor's output signal. This allows you to test the sensor, experiment with it and become familiar with how it works. We will do this by creating a program that continually samples the sensors and displays the current reading on the LCD screen. This will allow to you experiment by holding your robot varying distances from a wall to observe how the sensor's signal varies with distance.

Create your sensor testing program as follows:

1. Create a new project named "Wanderer."

   In a subsequent procedure we will enhance this program to make your robot wander around a room.

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

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

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

3. At the beginning of the main method, add statements to obtain the object for the LCD display and then output the name of your program on the first line of the display.

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

4. Add statements to retrieve the analog input objects for ports 1 (left sensor) and 2 (right sensor).

```
AnalogInput leftRangeSensor = IntelliBrain.getAnalogInput(1);
AnalogInput rightRangeSensor = IntelliBrain.getAnalogInput(2);
```

5. Add a while loop that will loop forever displaying values sampled from the sensors on the second line of the display.

```
while (true) {
    display.print(1, "L: " + leftRangeSensor +
                     " R: " + rightRangeSensor);
}
```

6. Build, load and test your program.

   Hold your robot within 30 inches (80 cm) of a wall. Observe the sensor readings as you vary the distance from the wall.

Programming Your Robot to Wander

A typical use of your IntelliBrain-Bot educational robot's infrared range sensors is to sense and avoid objects as your robot navigates from place to place. A fun way to illustrate this is to implement a Java program that allows your robot to wander around a room without running into things. You will only need to implement a few additional lines of Java code to employ the algorithm that is described in the next section.

A Wandering Algorithm

You can program your robot to wander without running into obstacles by implementing the simple behaviors illustrated in Figure 7-4: 1) if your program senses an object to the left, rotate right; 2) if your program senses an object to the right, rotate left; 3) otherwise, drive straight ahead. The following pseudo code illustrates this:

```
if (object sensed to the left)
    rotate right
else if (object sensed to the right)
    rotate left
else
    go forward
```

Your robot will drive straight ahead as long as there is nothing in its path. If there is an object in your robot's path, your robot will turn away from it. If both the left and right sensors detect an object at the same time, indicating there is an object straight ahead, your robot will turn right because the algorithm checks the left sensor first.

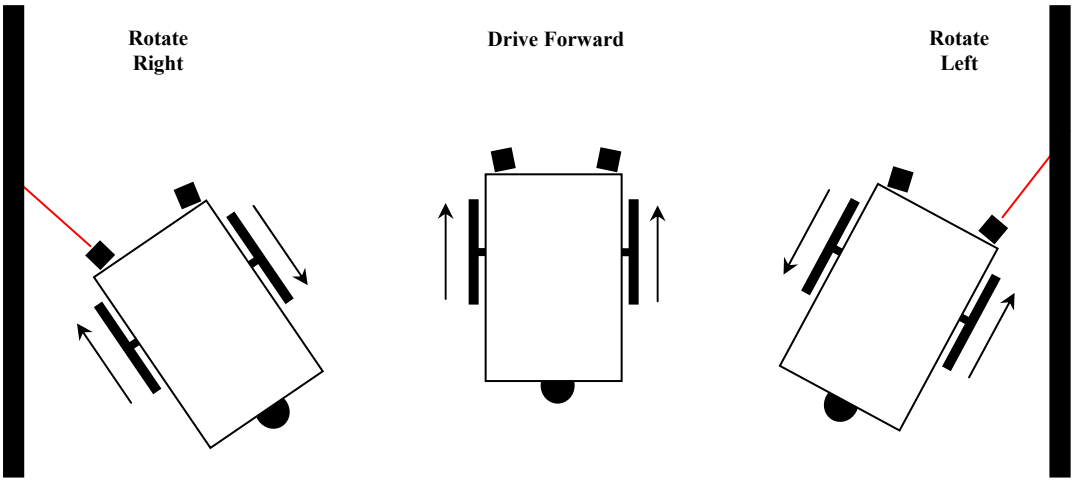


Figure 7-4 - Wanderer Behavior

Implementing the Wanderer Algorithm

Use the following procedure to extend your Wander program to implement the algorithm described above:

1. Add import statements to facilitate using the servo motors.

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

2. Add fields for the left and right motors.

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

3. Create Motor objects for the servos in the initialization portion of the main method.

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

4. Add a goForward method.

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

```
        rightMotor.setPower(MOTOR_POWER);
    }
```

5. Create a constant to specify the value of MOTOR_POWER used in the goForward method.

```
    private static final int MOTOR_POWER = 13;
```

6. Add a call to invoke the goForward method within the while loop in the main method.

```
    goForward();
```

7. Build, load and test your program.

   At this point you have implemented enough of your program that your robot should do two things: display the readings of the two sensors and drive forward forever.

8. Add a method to rotate your robot to the right.

```
    public static void rotateRight() {
        leftMotor.setPower(MOTOR_POWER);
        rightMotor.setPower(-MOTOR_POWER);
    }
```

9. Add a method to rotate your robot to the left.

```
    public static void rotateLeft() {
        leftMotor.setPower(-MOTOR_POWER);
        rightMotor.setPower(MOTOR_POWER);
    }
```

10. Implement the wanderer algorithm in the main method, where you previously inserted the goForward method.

    Recall that the infrared range sensors produce higher sample values when there is an object close ahead and lower values when an object is further away. Your program can determine if there is an object nearby by checking if the sensor reading is greater than a threshold.

```
    if (leftRangeSensor.sample() > THRESHOLD)
        rotateRight();
    else if (rightRangeSensor.sample() > THRESHOLD)
        rotateLeft();
    else
        goForward();
```

11. Create a constant named “THRESHOLD” to specify the object detection threshold.

```
private static final int THRESHOLD = 300;
```

12. Build, load and test your program.

    When you place your robot on the floor and start it, your robot should drive straight ahead until it encounters an object, at which point it should turn to avoid it. When the path forward is once again clear, your robot should continue driving straight ahead.

Summary

Infrared range sensors use reflected infrared light to detect an object in front of the sensor. The Sharp infrared range sensors used by your IntelliBrain-Bot educational robot vary their output voltage based on the distance to objects they sense. They do this by measuring the angle of reflected light. This technique makes them immune to variations of the ambient light conditions and to variations of the reflectivity of different objects.

You wrote a program that uses the range sensors to enable your robot to wander around a room without bumping into obstacles in its path.

It is easy to imagine that the software algorithms that control a robot are highly complex; however, in this chapter, you implemented a very simple program that worked effectively. Not all robotics programs need to be complex to be effective.

Exercises

1. Examine the wiring between the range sensors and the IntelliBrain 2 robotics controller board. Which input ports do the two infrared range sensors connect to?

2. Using your Wanderer program, hold your robot with one of the range sensors facing a wall about 4 inches (10 cm) from the wall. Using a ruler or tape measure to measure the distance between the sensor and the wall, record readings from the sensor at various distances. Create a graph similar to the one shown in Figure 7-3, plotting the sampled value along the y-axis instead of voltage.

3. Create an effective range plot, similar to the plot shown in Figure 5-4 for the Ping))) sensor, but this time for an infrared range sensor.

4. If you change the THRESHOLD constant in your Wanderer program from 300 to 200, will your robot go closer to objects or stay further away?

5. Experiment with the THRESHOLD value in your Wanderer program according to

Table 7-1 and make note in the table how your robot behaves when using each value.

6. Let your robot wander in a room running your Wanderer program. Observe that your robot tends to fear the center of the room, preferring to stay close to the walls. Why is this?

7. Cure your robot's fear of open space by modifying your Wanderer program such that your robot will regularly venture into the center of the room and away from the walls.

   Hint: Change the rotate methods such that your robot rotates a large angle, for example 90 degrees, rather than the minimum angle to avoid the obstacle. Use Thread.sleep within the rotate methods to ensure the rotation angle is large enough.

8. Using your Wanderer program, place your robot such that it drives toward a corner of the room at approximately a 45 degree angle. Repeat this test several times, varying the angle until your robot gets trapped in the corner. How does your robot behave when it gets trapped? Why does this happen?

9. Modify your Wanderer program to prevent your robot from getting trapped in a corner.

   Hint: Use the Random class and Thread.sleep in the rotate methods to cause your robot to rotate a different amount each time it enters these methods, allowing it to find a rotation angle that allows it to escape from a corner.

Table 7-1 - Robot Behavior Verses Threshold Value

THRESHOLD Value	Observed Behavior
200	
300	Robot turns away when it comes within ~5 inches (12 cm) of an object.
400	
500	
600	
700	

CHAPTER 8

Shaft Encoding

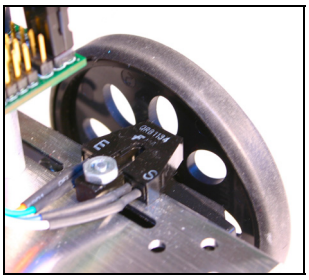


Figure 8-1 – Wheel Position Sensor

Previously, you learned how you can use timed sequences of commands to program your IntelliBrain™-Bot Deluxe educational robot to perform simple maneuvers; however, the maneuvers your robot performed based only on timing where inconsistent. Your robot's performance varied significantly due to changes in battery voltage, friction and other factors. Since your program lacked a way to measure the actual motion of your robot, it could only estimate your robot's position based on timing and the results of prior experiments, which you incorporated into your program. In this chapter you will use your IntelliBrain-Bot educational robot's wheel position sensors, shown in Figure 8-1, to implement shaft encoders. Shaft encoders are sensors that measure the rotation of a shaft or wheel. This will allow your programs to measure movement of your robot's wheels, facilitating far more accurate estimates of position than is possible with timing only. This will enable your robot to maneuver and navigate much more effectively than it could by relying only on predetermined, timed sequences of motor commands.

Your IntelliBrain-Bot educational robot has two plastic wheels, one of which is shown in Figure 8-1. Each wheel has eight spokes separated by eight oblong holes. These spokes and holes are fundamental to sensing movement of the wheel. By using the same type of infrared photo-reflective sensor we used for line sensing, your program can detect whether a spoke or a hole is in front of the sensor. By checking the sensor frequently, your program can detect movement of the wheel, enabling it to track the position of the wheel. This technique is known as "shaft encoding" because it relies on a sensor signal that encodes the angular position of a shaft. In this case the shaft is the axle the wheel is mounted on.

An infrared photo-reflective sensor consists of an infrared light emitting diode (LED) and a phototransistor pair. The LED outputs infrared light and the phototransistor detects reflected infrared light. The phototransistor has an output signal whose voltage varies depending on the intensity of the reflected infrared light striking it. When the sensor is adjacent to a solid surface, infrared light from the LED will reflect back on to the phototransistor, as shown in the upper portion of Figure 8-2. The output voltage of the detector will be low in this case. When there is no surface adjacent to the sensor, infrared light from the LED will not be reflected on to the phototransistor. In this case, the output voltage of the phototransistor will be high. This situation is depicted in the lower portion of Figure 8-2.

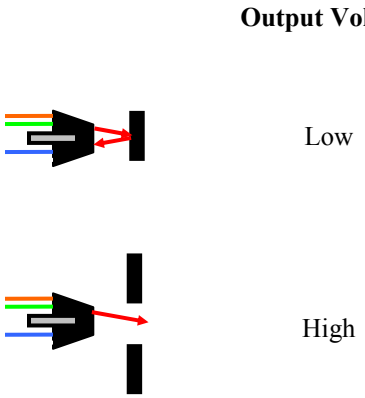


Figure 8-2 – Infrared Photo-reflective Sensor Operation

By mounting a sensor adjacent to each wheel, as shown in Figure 8-1, your program will be able to use the output signal from the detector to sense whether a spoke or hole is adjacent to the sensor at any point in time. The sensor will produce a low reading when a spoke is in front of it and a high reading when a hole is in front of it. By checking the signal frequently your program can sense movement of the wheel.

Testing the Wheel Sensors

Let's start by first writing a program to test the wheel sensors to verify they are functioning properly. This will also allow you to experiment with the sensors to better understand how they behave.

Use the following procedure to create a test program and use it to experiment with the wheel sensors.

1. Create a new project named "WheelSensorTest."

2. Add the following import statements.

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

3. Within the main method, get the display object.

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

4. Get the AnalogInput objects for analog ports 4 and 5, which are used by the left and right wheel sensors, respectively.

```
AnalogInput leftWheelSensor = IntelliBrain.getAnalogInput(4);
AnalogInput rightWheelSensor = IntelliBrain.getAnalogInput(5);
```

5. Add a loop to sample the analog-to-digital converters and display the sampled value for each wheel sensor.

```
while (true) {
    display.print(0, "Left:  " + leftWheelSensor.sample());
    display.print(1, "Right: " + rightWheelSensor.sample());
    try {
        Thread.sleep(1000);
    }
    catch (Exception e) {}
}
```

6. Build, load and test your program.

   Holding your robot in your hand, slowly turn each wheel. Note the highest and lowest value reported by each sensor. Also, note the position of the holes and spokes of the wheel relative to the sensor when the sensor is reading high and when it is reading low.

Collecting and Analyzing Sensor Data

In preparation for implementing a shaft encoder class, it will be helpful to collect and plot wheel sensor data. By doing this, you will better understand the nature of the wheel sensor signal. You can accomplish this by writing a small program to sample a wheel sensor at high frequency, sampling and collecting data into an array. Once the sampling is complete, your program can output the sampled data to the Run window in RoboJDE™ Java™-enabled robotics software development environment. From there you can copy it and paste it into a spreadsheet program for plotting. By storing the samples in an array first and then outputting the data, your program will be able to sample at a higher frequency than if it printed the data out directly. Printing is time consuming.

Create a program to collect and output sensor data using the following procedure.

1. Create a new project named “CollectSensorData.”

2. Add the following import statements.

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

3. Within the main method, use the ContinuousRotationServo class to create a Motor object for the left motor.

```
Motor motor = new ContinuousRotationServo(
                    IntelliBrain.getServo(1), false, 14);
```

4. Follow the previous statement with a statement to obtain the AnalogInput object for the left wheel sensor.

```
AnalogInput sensor = IntelliBrain.getAnalogInput(4);
```

5. Add statements to get the object for the START button and then wait for the START button to be released. Follow this with statements to print a message indicating that pressing the button will start data collection and to wait for the button to be pressed.

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

System.out.println("Press START to\ncollect data");
startButton.waitPressed();
```

6. After the START button is pressed, your program will need to start the motor running at full power then wait about a half second for the wheel to reach full speed. Add the following statements to do this.

```
motor.setPower(Motor.MAX_FORWARD);
try {
    Thread.sleep(500);
} catch (InterruptedException e) {}
```

7. Once the wheel is spinning at full speed, your program can create an array to collect one hundred samples and then loop sampling the sensor every five milliseconds, as follows:

```
int[] samples = new int[100];
long nextTime = System.currentTimeMillis();
for(int i = 0; i < samples.length; ++i) {
    samples[i] = sensor.sample();
    nextTime += 5;
    try {
```

```
            Thread.sleep(
                nextTime - System.currentTimeMillis());
        } catch (InterruptedException e) {}
    }
```

8. Finally, add statements to stop the motor and print out the array of samples.

```
motor.setPower(Motor.STOP);

for (int i = 0; i < samples.length; ++i) {
    System.out.println(
        Integer.toString(i * 5) + '\t' + samples[i]);
}
```

   The `\t` character indicates the tab character.

9. Build, load and test your program.

   Hold your robot in your hand, being sure to keep the left wheel clear so it can spin freely. Leaving the serial cable connected to the host computer, press the START button to start collecting data. Once data collection is complete, the collected data will be displayed in the RoboJDE Run window.

Plot this data by hand or copy it and paste it into a spreadsheet program to create a plot similar to the one shown in Figure 8-3.

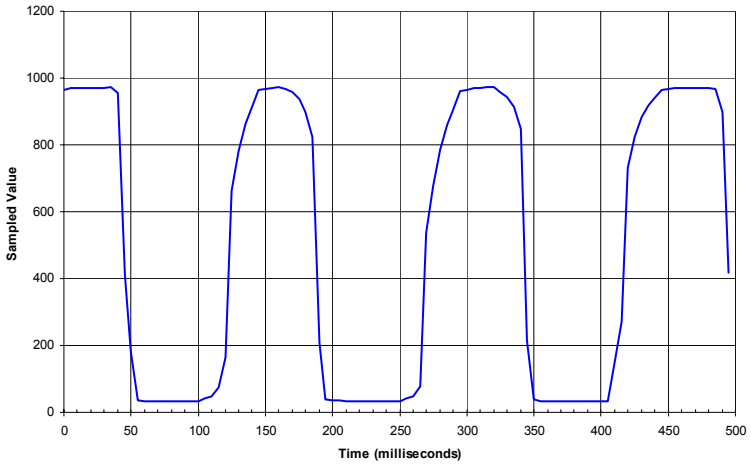


Figure 8-3 - Wheel Sensor Data

Recall from the earlier discussion of the operation of the sensor, the sampled value is low when a spoke is in front of the sensor and high when a hole in the wheel is in front of the sensor. The rising and falling edges of the signal correspond to the edges of the spokes.

Examine the plot and note the time between rising edges is roughly 150 milliseconds. This is the time for one full hole and one full spoke to pass in front of the sensor. This

corresponds to one eighth of a revolution of the wheel. Multiplying by eight, it takes approximately 1.2 seconds for the wheel to rotate one revolution. Considering the servo was at full power and the wheel was spinning freely, this is the top speed of the wheel – approximately 50 revolutions per minute.

Note there are twice as many spoke edges as there are spokes. By sensing the edges of the spokes rather than the spokes themselves, your program can sense the wheel position with twice the accuracy, measuring the wheel position to one sixteenth of a revolution. Your program can easily detect spoke edges by detecting the sensor signal transitions from low to high and high to low.

In order for your program to detect every spoke and every hole, it must sample the sensor at least twice during the period it takes for a spoke and a hole to pass by the sensor. That is, two samples per one eighth turn of the wheel. If your program samples at a lower rate than this, it will miss some spokes or holes and not be able to reliably track the position of the wheel. Since it takes 150 milliseconds for the wheel to rotate one eighth of a revolution at full speed, your program must sample the sensor at least once every 75 milliseconds. This will ensure it detects every spoke and every hole. Unfortunately, there is a downside to sampling too frequently; sampling more frequently will consume more computing power than sampling less frequently. Therefore, sampling more frequently leaves less computing power to allocate to other tasks, so you will want to sample just frequently enough to ensure your encoders do not miscount.

Implementing a Shaft Encoder

Shaft encoding is a method of tracking the angular position and velocity of a rotating shaft. Shaft encoders are very popular and widely used. They are commonly used in odometers, speedometers and tachometers on cars, motorcycles and bicycles.

As its name suggests, a shaft encoder encodes the position of a shaft. The shaft encoder class you will implement in this exercise is simply a counter. The counter counts up as the wheel rotates forward and it counts down as the wheel rotates backwards. The counter records one count, or "click," each time the shaft rotates a fraction of a revolution. In the case of your robot, the passage of each spoke edge in front of the sensor constitutes a "click" that your encoder will count.

An interesting form of an encoder that really does make a clicking sound is that of a playing card attached to a bicycle wheel. You may have added one of these "encoders" to your bicycle when you were a child. When you do this, the playing card makes a clicking sound each time a spoke snaps past the card. The sound made by the card changes as the speed of the wheel changes. Simply by listening to the frequency of the clicking sound, you can sense how fast the wheel is spinning.

Instead of using a playing card to generate clicks, your robot's shaft encoder uses a photo-reflective sensor to generate the electronic equivalent of clicks. The passage of each spoke edge results in a low to high or high to low signal transition, as shown in Figure 8-3, which constitutes one click of the encoder.

As part of the exercise of creating a shaft encoder, we will apply object-oriented programming techniques to create an encoder class that nicely encapsulates the functionality of the encoder and facilitates reuse of the class.

Object Oriented Programming

One of the biggest benefits of the Java language is its object oriented nature. Object-oriented languages provide mechanisms to encapsulate programming logic into classes that represent abstract objects in a software system. Effective object-oriented design results in classes that implement a cohesive set of functions. Well designed classes hide irrelevant details from other parts of the software system. By doing so, they reduce the coupling between classes, minimizing interdependencies and overall complexity of programs.

Based on the exercises you have already completed, you are already familiar with many examples of classes from the RoboJDE class library that effectively encapsulate functionality into a cohesive class. These classes expose only details that are relevant to your programs, hiding many implementation details. For example, you have used AnalogInput input objects to sample signals from sensors connected to analog ports on the IntelliBrain 2 robotics controller. Your programs haven't had to be concerned with the details of what microcontroller chip the robotics controller uses nor have they had to be concerned with particular details of how the microcontroller implements analog-to-digital conversion. The only thing your programs have had to be aware of is that they can sample the attached sensor's signal by using the sample method. The inner workings of the class have been hidden from your programs. This prevents your programs from becoming tightly tied to one particular model of robotics controller.

Similar to the many classes you have already worked with from the RoboJDE class library, you will want to implement a shaft encoder class that follows the principles of object-oriented design, hiding details that aren't relevant outside of the class and providing straight-forward methods to provide access to essential features. In the case of a shaft encoder, the relevant feature to expose is the current value of the tick counter. Your programs typically will not be concerned with how the class maintains the counter value, what type of sensor is used to sense the shaft's position, and what port the sensor connects to. The implementing class should hide these details. The shaft's current position, indicated by the shaft encoder's current count, is what is relevant and should be exposed. The implementing class must make the current count accessible to other parts of your program.

Java provides three features that will facilitate creating a shaft encoder class that follows object-oriented design principles. These Java features are interfaces, classes, and multi-threading.

Interfaces

The Java language's interface concept provides a means of declaring that a class provides a specific set of methods. A class that declares support for an interface must implement

all of the methods defined that interface. Rather than referring to objects by their specific class name, objects can be referenced based on an interface they support. Using interface references rather than specific class references reduces coupling between Java classes. This minimizes interdependencies and avoids sharing irrelevant details between classes, reducing coupling in your program.

The class, IntelliBrainAnalogInput implements the AnalogInput interface, as denoted in the Figure 8-4. A class that implements the AnalogInput interface is required to implement the interface's two methods: getMaximum and sample. However, the interface does not restrict what other methods may be implemented, nor does it place any restrictions on how the class implements the interface's methods. The IntelliBrainAnalogInput class implements many other methods. It provides the implementation of the two AnalogInput interface methods to allow interfacing with the IntelliBrain 2 robotics controller. Different classes for other robotics controllers will implement these methods differently. By using the AnalogInput interface, your programs are isolated from differences between robotics controllers.

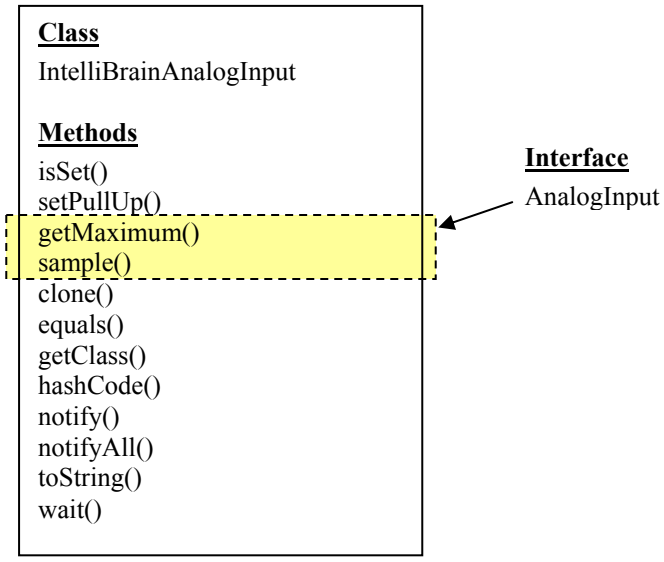


Figure 8-4 - AnalogInput Interface

You have used many interfaces in the exercises you have already completed. These include AnalogInput, Display, LED, Motor, PushButton and Speaker.

To summarize, the major benefit of using interfaces is that they provide loose coupling within your program, reducing coupling between classes, making it easier to reuse classes in other projects and making it easier to maintain your programs. Referencing an object via an interface name, rather than via an implementing class, facilitates code reuse and maintainability. For example, by using the Motor interface to refer to the objects that control your robot's motors, your programs and classes will not be limited to working with just one particular type of motor controller. You can switch to a different type of motor controller with minimal software changes. You can also reuse your classes and programs for other projects such as a robot you build yourself.

The RoboJDE class library defines an interface for shaft encoders named "ShaftEncoder." The ShaftEnocder interface is defined as follows:

```
public interface ShaftEncoder {
    public int getCounts();
    public int getRate();
}
```

The getCounts method returns the current value of encoder's counter. The getRate method returns the current rate at which the encoder's counter is currently changing. You will use this interface when you implement your shaft encoder class.

Classes and Objects

Two fundamental concepts in object oriented programming, and the Java language, are the class and the object. You have already worked with these but may not have thought a whole lot about the distinction between a class and an object. Now that you will be implementing a class that you will reuse in future exercises, it's a good time to clarify this distinction.

A class represents a type of thing. In the case of this exercise, you are going to implement a class to represent a shaft encoder. A convenient way to think of a class is as the Java source code that implements the class. Commonly, each class is implemented in its own Java source file. In this exercise you will create a class named AnalogShaftEncoder in a file named AnalogShaftEncoder.java.

An object is an instance of a class. Objects contain the run-time data for a specific instance of a class. In this exercise, your program will create two instances of your shaft encoder class, one for the left wheel and one for the right wheel.

Multi-threading

When analyzing the data we collected from the sensor, we determined that the wheel sensor needs to be sampled at least once every 75 milliseconds. This is to ensure the encoder does not miscount, which will occur if an entire spoke or hole passes the sensor during the time between any two samples. Your program has to be constructed such that it samples the encoder at least once every 75 milliseconds in order to maintain the encoder counter. To be conservative, it should update each encoder more often than this, say every 30 milliseconds. One way to accomplish this is to have your encoder implement an update method to sample the sensor and update the counter. You would then sprinkle calls to this method around your main program, being careful to ensure that the update method is called frequently enough. If you were to do this, any time you changed your program you would need to reevaluate the placement of method calls to ensure the timing constraint isn't violated. Clearly, this would be awkward and your program would be difficult to maintain. Fortunately, Java provides a much better way to solve this problem, multi-threading.

Multi-threading allows your program to perform multiple tasks, or threads of execution, at the same time. It also allows you to give higher priority to some threads (tasks) than to others. When a higher priority thread is ready to run, the microcontroller's central processing unit (CPU) will stop executing the lower priority thread and begin executing the higher priority thread. When the higher priority thread's work is complete, the CPU will resume execution of the lower priority thread at the point where it previously left off. You can use this technique to implement your shaft encoder class. You can implement your class such that it will wake up periodically to sample its wheel sensor and update its counter and then sleep until it is time to update again. Figure 8-5 depicts two shaft encoder threads, left and right, to monitor the left and right wheels, respectively. These threads run at higher priority than the main thread, allowing them to preempt the main thread when it is time to update the encoder. As depicted in the Figure 8-5, each encoder thread wakes up every 30 milliseconds to update its counter, preempting the main thread. The blocks in the figure indicate when the thread is executing. The thin lines indicate when the thread is not executing. When the encoder threads are sleeping, the CPU executes the lower priority, main thread. Since there is only one CPU on your robot, only one thread can execute at a time. Larger systems with multiple CPUs can execute multiple threads at the same time, one thread on each CPU.

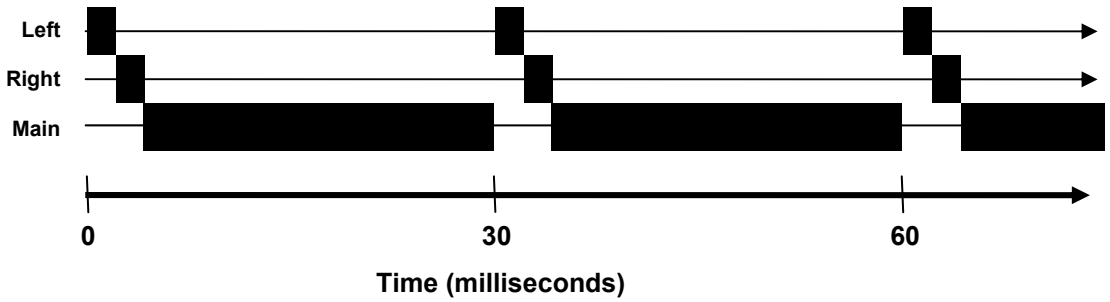


Figure 8-5 - Multi-threading of Shaft Encoders

As a simple analogy to multi-threading, consider a chef cooking the main course of a meal. The chef is analogous to the robotics controller's CPU. He must prepare all of the items that make up the main course at the same time, allowing the entrée, sauces and side dishes that make up the meal to be ready to serve all at once. The chef prepares the meal by breaking his work into preparing each item separately, according to a separate recipe for each. Each of these sub-tasks is a thread of work the chef must complete. Rather than preparing the items sequentially, preparing one item in its entirety before moving on to the next item, the chef prepares all of the items concurrently by working for brief periods on each item, cycling through all of the items, or threads of work, repeatedly. By working in this fashion, the chef performs multiple tasks at the same time. While it seems he works continuously on each item, at any instant in time he really only works on one item. He just switches from item to item quickly, as they need attention. Java programs execute multiple threads in the same way when you make use of multi-threading.

Creating a Shaft Encoder Class

Now that we have discussed all of the major concepts you will need, let's create a shaft encoder class. Your class will duplicate the functionality of the AnalogShaftEncoder class in the RoboJDE class library. By implementing the class yourself you will have the opportunity to appreciate the inner workings of the class.

Use the following procedure to create and test the class.

1. Create a new project named "ShaftEncoderTest."

2. Using the RoboJDE File->New Class menu item, create a new class named AnalogShaftEncoder.

3. Working in the AnalogShaftEncoder class edit window, add statements to import the AnalogInput, ShaftEncoder and DirectionListener interfaces from the RoboJDE class library.

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

4. Update the class definition statement to denote that your AnalogShaftEncoder class will extend the Thread class and implement the ShaftEncoder and DirectionListener interfaces.

```
public class AnalogShaftEncoder
        extends Thread
        implements ShaftEncoder, DirectionListener {
```

   The AnalogShaftEncoder class must implement the methods required by the two interfaces it declares to support: ShaftEncoder and DirectionListener. You can find more information on these interfaces in the RoboJDE class library documentation.

   The DirectionListener interface allows each encoder object to integrate with the associated ContinuousRotationServo object to receive updates when the motor direction is changed. This will allow the encoder to know whether it needs to increment or decrement its counter as it senses spoke edges passing the sensor.

5. Add fields to refer to the AnalogInput used by the encoder, to hold values for high and low thresholds, to keep track of the shaft direction, to hold the counter value and to hold the sampling period.

```
private AnalogInput input;
private int lowThreshold;
private int highThreshold;
private boolean isForward;
private int counts;
```

```
private int period;
```

The threshold values will be used to determine whether the sensor reading is high or low. Two values will be used to provide hysteresis, a retarding effect, that prevents small variations in the sensor signal from being mistaken as a spoke edge passing the sensor.

6. Create a constructor for the AnalogShaftEncoder that takes the following arguments: AnalogInput used by the encoder, high and low thresholds, sampling period and thread priority.

   It will be convenient to use the same variable names when constructing the object as when accessing it later. Therefore, use the same names for the arguments to the constructor as you used for the field names. In the next step, you will use the "this" reference to distinguish between the object's fields and the constructor's arguments, which have the same names.

```
public AnalogShaftEncoder(
    AnalogInput input,
    int lowThreshold,
    int highThreshold,
    int period,
    int threadPriority) {
}
```

7. Within the constructor method, add statements to initialize all of the fields. You will need to use the "this" reference to distinguish between the identically named fields and arguments.

   The this reference is built into the Java language. The Java language uses the this reference implicitly to access fields when your program does not explicitly provide a reference; however, when your program defines a local variable with the same name as a field, the Java language will assume your program is referring to the local variable if it does not explicitly reference the field. In other words, your program must include a this reference when it is referring to a field and not a local variable by the same name.

```
this.input = input;
this.lowThreshold = lowThreshold;
this.highThreshold = highThreshold;
this.period = period;
isForward = true;
counts = 0;
```

8. Follow the field initialization with calls to the parent class's methods to set the priority, to configure the thread to be a daemon thread, and to start the thread.

```
setPriority(threadPriority);
setDaemon(true);
```

```
start();
```

A daemon thread is a thread that does not prevent the program from terminating when all other non-daemon threads have terminated. When the last non-daemon thread returns from its run method, the virtual machine will terminate your program even if daemon threads are still executing. You don't want your shaft encoder class to prevent your programs from exiting. Therefore, you must configure it to be a daemon thread.

9. Add the "getter" method for the encoder's counts value, as required by the ShaftEncoder interface.

```
public int getCounts() {
    return counts;
}
```

10. Add the getter method for the encoder's current counting rate, as required by the ShaftEncoder interface.

```
public int getRate() {
    return 0;
}
```

We will leave the full implementation of this method to be a later exercise.

11. Implement the updateDirection method, as required by the DirectionListener interface.

```
public void updateDirection(boolean isForward) {
    this.isForward = isForward;
}
```

12. Add a run method to extend the Thread class. This method will implement the code to sample the sensor and update the encoder's counter.

```
public void run() {
}
```

13. Click on the build button to build your program. Fix any compilation errors before continuing.

14. Add a try-catch block to the run method to catch and print out a stack trace if any exceptions occur in the Thread's run method.

Include a try-catch block any time you implement a run method for a Thread. If you do not do this and an exception occurs while the thread is running, the thread will exit without any indication, leaving you wondering why your program isn't

functioning properly.

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

15. Within the try block, add statements to create and initialize a local variable to track the last state – high or low – of the sensor's signal.

```
boolean wasHigh = false;
if (input.sample() > lowThreshold)
    wasHigh = true;
```

16. Add a while loop that executes at the specified period and runs forever to sample the sensor and update the encoder's counter.

```
while (true) {
    // insert sampling and update statements here
    Thread.sleep(period);
}
```

17. At the point noted within the while loop above, add a statement to sample the analog input.

```
int value = input.sample();
```

18. Next, add if statements to determine if the encoder's counter needs to be updated.

    If the signal was high and dropped below the low threshold or was low and increased above the high threshold, the counter will need to be updated. The counter must be incremented (increased by one) if the shaft was rotating forward, and decremented (decreased by one) if the shaft is rotating backward.

```
if (wasHigh) {
    if (value < lowThreshold) {
        if (isForward)
            counts++;
        else
            counts--;
        wasHigh = false;
    }
}
else {
    if (value > highThreshold) {
        if (isForward)
            counts++;
        else
            counts--;
        wasHigh = true;
```

```
    }
}
```

19. Click on the build button to build your program. Fix any compilation errors before continuing.

20. Switch to the Edit window tab for the ShaftEncoderTest class and add the following import statements.

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

21. Within the main method, add a statement to get the object for the LCD display.

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

22. Add statements to get the AnalogInput objects for the two wheel sensors.

```
AnalogInput leftWheelSensor =
        IntelliBrain.getAnalogInput(4);
AnalogInput rightWheelSensor =
        IntelliBrain.getAnalogInput(5);
```

23. Add statements to construct shaft encoder objects for the left and right wheels. Specify a low threshold of 250 and a high threshold of 750. Specify a sample period of 30 milliseconds and specify the thread priority to be the maximum priority.

```
ShaftEncoder leftEncoder = new AnalogShaftEncoder(
    leftWheelSensor, 250, 750, 30, Thread.MAX_PRIORITY);
ShaftEncoder rightEncoder = new AnalogShaftEncoder(
    rightWheelSensor, 250, 750, 30, Thread.MAX_PRIORITY);
```

    The threshold values can be chosen by examining the sensor signal plot shown in Figure 8-3. The exact values of the threshold are not critical provided they are well within the range of the sensor signal.

24. Add statements to loop forever, reading the counter values from both encoders and displaying them on the LCD display.

```
while (true) {
    display.print(0, "L enc: " + leftEncoder.getCounts());
    display.print(1, "R enc: " + rightEncoder.getCounts());
    try {
        Thread.sleep(1000);
    }
    catch (Exception e) {}
```

```
    }
```

25. Build, load and test your program.

    Holding your robot in your hand, start your program. Observe the encoder counter values are zero for both encoders. Gently turn one of the wheels. Notice the associated counter value will increase as the wheel turns. Repeat for the other wheel and notice counter values for that wheel will increase also.

    The encoder counters will not decrease when you turn the wheel backwards. This is because your program has no way of knowing which way the shaft is turning. In future projects, when your program is controlling the direction in which power is applied to the motor, the DirectionListener interface method, updateDirection will be used to notify the shaft encoder when the motor switches direction.

Summary

You have created a shaft encoder class to monitor the rotation of your robot's wheels. It uses an infrared photo-reflective sensor to sense and count spoke edges as the wheel turns. In the next chapter, you will use this class to enable your robot to track its location.

You have also learned about several important features of the Java language: classes, objects, interfaces and threads.

A class defines a type of thing. In the Java language, a class has a unique name and consists of the code that implements the class. Each class is commonly implemented in a separate Java source file.

An object is an instance of a class. Your ShaftEncoderTest program created two objects that are instances of the AnalogShaftEncoder class, one object to monitor the position of each of the two wheels on your robot.

An interface defines a set of methods that classes may declare they support. By referring to objects using interface names rather than class names, you can greatly reduce the coupling between classes in your programs. This makes your programs easier to maintain and extend.

Multi-threading provides a convenient way to create and configure threads of execution that perform various tasks in your programs. The virtual machine's thread scheduler coordinates the execution of threads. You can give each thread a priority. If multiple threads are ready to run, the virtual machine will execute the thread with the highest priority.

Exercises

1. Trace the wires from each wheel sensor to the controller board. Record the port number on the IntelliBrain 2 robotics controller each sensor connects to.

2. Using your WheelSensorTest program, gently turn each wheel. Record the minimum and maximum sample values that you observe for each wheel sensor. Calculate the signal voltage indicated by each of the sample values.

3. Using your CollectSensorData program, collect and plot data from the wheel sensor to create a graph similar to Figure 8-3.

4. Using the data you plotted in the previous exercise, determine the period from the leading edge of one spoke to the leading edge of the next spoke. Calculate the time for the wheel to turn one full revolution and calculate the number of revolutions per minute (RPM) of the wheel.

5. Using the chart that you created previously, estimate the maximum sampling period your program must use to ensure your encoders do not miscount. Explain the reasoning behind your estimate.

6. Briefly explain how a shaft encoder works. Also, explain how you can use the wheel sensors on your robot to implement a shaft encoder class.

7. Explain the relationship between a class and an object in a Java program.

8. Modify your CollectSensorData program to collect data for the right wheel instead of the left wheel. How many changes did you need to make in your program to accomplish this? Explain how object-oriented design simplified this exercise.

9. Explain the benefit of using the Java interface mechanism in your programs.

10. Explain how your AnalogShaftEncoder class makes use of multi-threading.

11. Enhance your AnalogShaftEncoder class to implement the getRate method such that it returns the number of counts per second the encoder is currently registering. Also, update your test program to display the rate for each encoder.

12. Implement a program that uses the thumbwheel to set the power level of a motor and uses your AnalogShaftEncoder class to measure wheel movement. Output the current power setting on the first line of the LCD screen and the wheel RPM on the second line. Create a chart of the wheel RPM verses motor power level.

13. Create a program to make your robot move straight forward until both encoders have counted at least one hundred counts.

14. Using your program from the previous exercise, adjust the sample period you specify when constructing the AnalogShaftEncoder objects. Set a large period, such as one second, and observe how your robot behaves. How does its behavior change? Why? What is the downside of using a very high sampling frequency and short sampling period?

CHAPTER 9

Tracking Position

The ability to track its own position is often an important requirement for mobile robots. For example, a robotic vacuum cleaner may rely on tracking its current position in order to find its way back to its charging station. One method of tracking position employs odometry and dead reckoning, techniques you will learn about in this chapter.

Localization

Knowing where you are is a problem that has challenged and fascinated humans for ages. We solve this problem by relying on our human senses of sight, sound and smell as well as by using tools, including maps, compasses, odometers and the Global Positioning System (GPS). Likewise, there are a variety of approaches you can use to enable your robot to determine its current location, a process referred to as "localization." Robotic researchers have developed many approaches to localization, including sensing landmarks, dead reckoning and using GPS receivers. In this chapter, we will use shaft encoder sensors in conjunction with dead reckoning to enable your robot to keep track of its position.

Designing Localization Classes

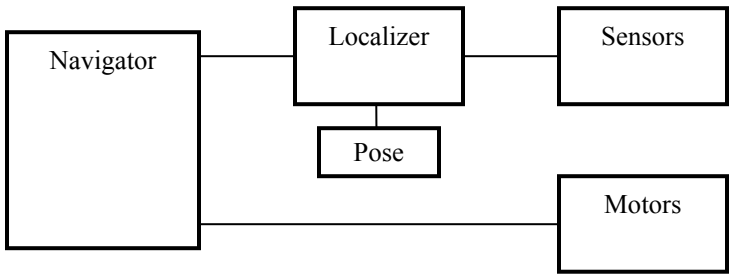


Figure 9-1 – Navigation and Localization Class Diagram

Figure 9-1 is a class diagram depicting navigation and localization classes that you will implement to enable your robot to keep track of its position, and in the next chapter, to navigate from place to place. In this design, the class that implements the "Localizer" interface uses sensors to keep track of your robot's current location and the direction it is

heading. The Localizer monitors the sensors and determines your robot's current position – its "pose" – and makes the position data accessible to other classes through a getter method. The getter method returns the current pose data as an instance of the class named "Pose."

The Navigator relies on the Localizer to keep track of your robot's current position. With knowledge of its current position, the Navigator has the information it needs to navigate your robot from its current location to its intended destination.

As you proceed through this chapter, you will build the localizer framework, which consists of the Localizer interface and the Pose class. You will also create a class that implements the Localizer interface using one of many possible localization techniques, odometry. We will name this class "OdometricLocalizer," because it will use odometry to accomplish localization. Additionally, you will create a test program, "LocalizerTest," that will allow you to build and test your localizer classes. You will create a Navigator class in the next chapter.

Begin by creating a new project and the framework for your test program.

1. Create a new project named "LocalizerTest."

2. Add the following import statements.

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

3. Within the main method, add statements to get the display object, output the name of your program and then loop forever.

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

while(true);
```

4. Build, load and test your program.

Defining Your Pose Class

The purpose of your Pose class is to hold data that indicates your robot's current position and heading. In two dimensions, this consists of three variables: the x and y coordinates, and the heading angle. In order to keep the class simple, we will simply define three fields that provide public access, rather using private fields with access methods. This will make the code slightly more efficient; however, it is important to ensure other classes don't modify the data held by a Pose object. Fortunately, the Java language provides a means to create immutable objects, which are objects that cannot be changed after they've been created. By declaring fields as "final," they will be immutable, which

means they can only be initialized by an object's constructor. They can't be modified later.

Use the following procedure to create your Pose class.

1. Create a new class named "Pose."

2. Define the three fields that represent the position and direction of your robot. Make the fields directly accessible to other classes by using the "public" keyword, but make them immutable by also using the "final" keyword.

```
public final float x;
public final float y;
public final float heading;
```

3. Create a constructor that initializes the three fields.

   Recall that you will need to use the this reference to distinguish between the fields and local variables if you give the constructor's arguments the same name as the fields they initialize.

```
public Pose(float x, float y, float heading) {
    this.x = x;
    this.y = y;
    this.heading = heading;
}
```

4. Click the save all button to save your files.

Defining Your Localizer Interface

The purpose of creating a localizer interface is to define the methods any class implementing a localizer must support. In this case, a class implementing the Localizer interface must support a single method named "getPose." This method provides for retrieval of the current position and heading as a Pose object. Any class that can provide your robot's current position and heading can support the Localizer interface simply by implementing the getPose method. It doesn't matter whether the class uses GPS technology, odometry or some other technique, as long as it returns the current position and heading via the getPose method, it can declare it supports the Localizer interface.

Use the following procedure to define the Localizer interface.

1. Create a new class named Localizer.

   Interfaces are defined similar to classes except the "class" keyword is replaced with the "interface" keyword. An interface only lists required methods; it doesn't provide an implementation of the methods. The method implementation is left to the classes that implement the interface. Using the RoboJDE™ Java™-enabled

robotics software development environment GUI, create a new class named "Localizer," then edit the class definition, replacing the keyword "class" with "interface."

```
public interface Localizer {
}
```

2. Insert the one method required by the interface, getPose.

```
public Pose getPose();
```

3. Click the save all button to save your files.

Dead Reckoning

Deduced reckoning, or "dead reckoning," dates back to the early days of sailing when sailors used it to estimate the current position of their ship. Using measurements of speed and heading, they could estimate the movement of their ship over time. Unfortunately, they also had to be aware of a significant pitfall; errors due to imperfect information about winds and currents would accumulate over time. As they traveled using dead reckoning, their estimated position would become less and less accurate the further they traveled due to accumulated error. They could easily end up being shipwrecked if they relied solely on dead reckoning. Nevertheless, dead reckoning is a powerful localization technique that is in widespread use today. It is effective, provided you take its pitfalls into account.

You will use dead reckoning to implement your Localizer. By monitoring the motion of your robot's wheels, your program can deduce how far your robot has traveled and what direction it is heading. Using Geometry, your program can calculate how the movement of your robot's wheels affects its position and heading.

Your program can track the movement of your robot's wheels by using the AnalogShaftEncoder class, which you developed in the previous chapter. As each wheel rotates forward, its encoder counter counts up. As it rotates backwards, its encoder counter counts down. Each up or down count of the encoder corresponds to a fraction of a full turn of the wheel. The distance the wheel travels across the floor with each count can be calculated by dividing the circumference of the wheel by the number counts the counter increments or decrements with one full revolution of the wheel. The equation for this is:

```
distancePerCount = Pi * diameterWheel / countsPerRevolution
```

When your robot moves in a straight line, the distance it travels is approximately equal to the average number of encoder counts registered for each wheel times the distance your robot travels per encoder count:

```
deltaDistance = (leftCounts + rightCounts) / 2.0 * distancePerCount
```

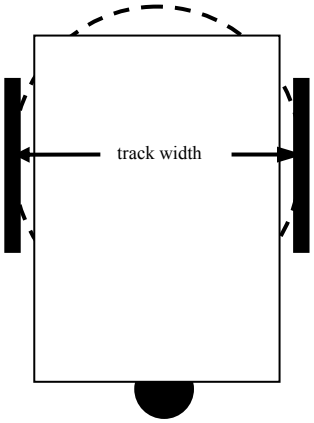


Figure 9-2 - Circle Traced by Robot Turning in Place

If your robot rotates in place by turning its wheels in opposite directions its wheels will trace a circle whose diameter is equal to its track width, as shown in Figure 9-2. Your robot's heading changes as the wheels make their way around this circle. Your robot will rotate 2 Pi radians (360 degrees) when the wheels have traversed the circumference of this circle. Therefore, the number of encoder counts to rotate your robot through a full circle can be calculated by the equation:

```
countsPerRotation = (trackWidth / wheelDiameter) * countsPerRevolution
```

This equation gives the number of counts for a single wheel. If you take the difference in counts between the two wheels and note that each rotation is 2 Pi radians, you can rearrange the equation into the following two equations:

```
radiansPerCount = Pi * (wheelDiameter/trackWidth) / countsPerRevolution

deltaHeading = (rightCounts – leftCounts) * radiansPerCount
```

where deltaHeading is the angle in radians your robot rotates.

Given the left and right encoder counts, these equations enable your program to update its estimate of your robot's position if you assume it moved straight ahead or rotated in place, but what if your robot moves along an arbitrary path?

Your localizer class can estimate your robot's position along an arbitrary path by treating its motion as many small, discrete movements. Provided the updates are small enough, each update can be assumed to be a straight line movement combined with a small change in heading. By adding all of the discrete movements together, the localizer can deduce your robot's position regardless of the curvature of the path it takes.

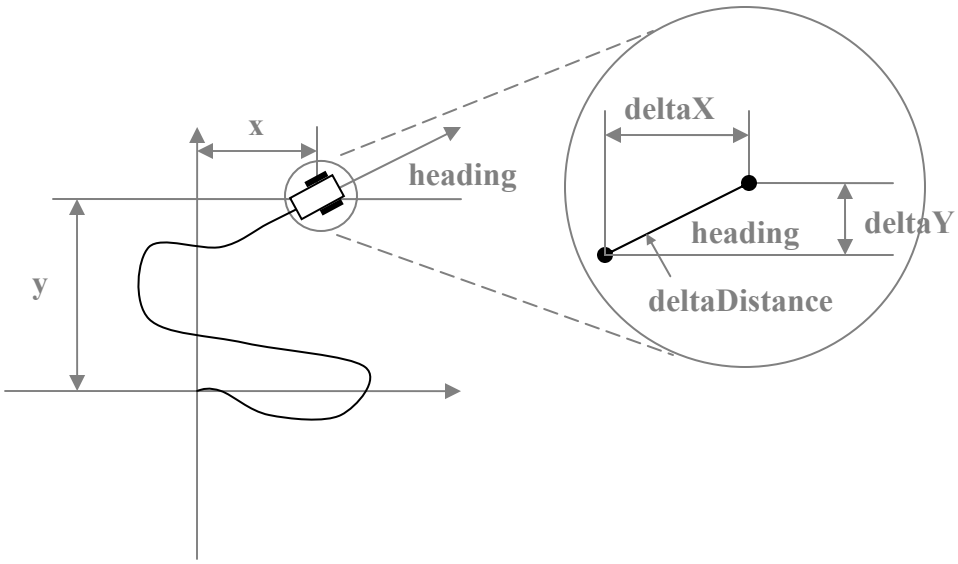


Figure 9-3 – Calculating Discrete Position Changes

As shown in Figure 9-3, imagine your robot moves a small distance, deltaDistance, while it travels forward in the heading direction. Assuming the direction your robot is heading doesn't change significantly during each small movement, the change in the position along the x-axis, deltaX, and change in position along the y-axis, deltaY, can be calculated using the following trigonometric equations:

```
deltaX = deltaDistance * cos(heading)

deltaY = deltaDistance * sin(heading)
```

The deltaX and deltaY values can then be added to the previous position estimate to obtain a new position estimate. By calculating position updates frequently, your localizer class can use dead reckoning to track the position of your robot.

Implementing Your OdometricLocalizer Class

The purpose of your OdometricLocalizer class is to maintain an up-to-date estimate of your robot's current position and heading. Your program will do this by performing dead reckoning calculations using data input from the shaft encoders. Your OdometricLocalizer class will use a separate thread to periodically update its estimate of your robot's current pose. It will take readings from the shaft encoders, using them to perform dead reckoning calculations to accomplish each update.

Create an OdometricLocalizer class using the following procedure.

1. Create a new class named "OdometricLocalizer."

2. Add an import for the ShaftEncoder interface.

```
import com.ridgesoft.robotics.ShaftEncoder;
```

3. Extend the declaration of the class to declare it extends the Thread class and it implements the Localizer interface.

```
public class OdometricLocalizer
    extends Thread implements Localizer {
}
```

4. Define two floating point constants for Pi and 2 Pi.

```
private static final float PI = 3.14159f;
private static final float TWO_PI = PI * 2.0f;
```

5. Define fields to refer to the left and right shaft encoders.

```
private ShaftEncoder leftEncoder;
private ShaftEncoder rightEncoder;
```

6. Define fields to hold the counter values returned by the encoders on the previous update cycle.

```
private int previousLeftCounts;
private int previousRightCounts;
```

7. Define fields to hold the distance each wheel moves per count and how many radians your robot's heading changes if the wheels rotate different amounts.

```
private float distancePerCount;
private float radiansPerCount;
```

8. Define and initialize the three fields required to keep track of your robot's pose.

```
private float x = 0.0f;
private float y = 0.0f;
private float heading = 0.0f;
```

9. Define a field to hold the value of how often the position updates should be done.

```
private int period;
```

10. Create a constructor method to construct an OdometricLocalizer object.

    The constructor will need to include arguments and calculations to initialize encoder object references, previous encoder count values, the distance your robot travels per encoder count and the number of radians it turns per encoder count. The constructor will also need to initialize the thread, setting its priority, making

it a daemon thread and starting it.

```
public OdometricLocalizer(
            ShaftEncoder leftEncoder,
            ShaftEncoder rightEncoder,
            float wheelDiameter,
            float trackWidth,
            int countsPerRevolution,
            int threadPriority,
            int period){
    this.leftEncoder = leftEncoder;
    this.rightEncoder = rightEncoder;
    distancePerCount =
        (PI * wheelDiameter) / (float)countsPerRevolution;
    radiansPerCount =
        PI * (wheelDiameter / trackWidth) / countsPerRevolution;
    this.period = period;
    previousLeftCounts = leftEncoder.getCounts();
    previousRightCounts = rightEncoder.getCounts();
    setDaemon(true);
    setPriority(threadPriority);
    start();
}
```

11. Define the getPose method and add the synchronized keyword to the method.

    The synchronized keyword allows coordinating multiple threads. In this case, we will use it to prevent a thread calling the getPose method from reading the pose data while the OdometricLocalizer object's thread data is updating the data. If this were to happen, the reader thread would get incorrect pose data. Some of the fields would hold values from the previous update while other fields would hold values from the current update. Synchronizing access to the pose data will ensure the getPose method always returns a consistent set of values that are all from the same update.

    ```
    public synchronized Pose getPose() {
    }
    ```

12. Add one line to the getPose method to construct a pose object and return it.

    ```
    return new Pose(x, y, heading);
    ```

13. Define a run method.

    ```
    public void run() {
    }
    ```

14. Add a try-catch block to the run method to catch and report any exceptions that may occur in the run method.

    ```
    try {
    catch (Throwable t) {
    ```

```
        t.printStackTrace();
    }
```

15. Create an update loop within the try block such that the thread will wake up to update the pose data at the specified period.

```
long nextTime = System.currentTimeMillis();
while (true) {
    // insert position update code here...
    nextTime += period;
    Thread.sleep(nextTime - System.currentTimeMillis());
}
```

16. At the position update the loop indicated by the comment text, add statements to calculate the new position of your robot.

    First sample the values of the two encoder counters then calculate the changes since the previous samples.

```
int leftCounts = leftEncoder.getCounts();
int rightCounts = rightEncoder.getCounts();

int deltaLeft = leftCounts - previousLeftCounts;
int deltaRight = rightCounts - previousRightCounts;
```

17. Add statements to calculate the distance your robot moved and then calculate the x and y components of the change.

```
float deltaDistance =
    0.5f * (float)(deltaLeft + deltaRight) * distancePerCount;
float deltaX = deltaDistance * (float)Math.cos(heading);
float deltaY = deltaDistance * (float)Math.sin(heading);
```

18. Add a statement to calculate the change in heading.

```
float deltaTheta =
    (float)(deltaRight - deltaLeft) * radiansPerCount;
```

19. Insert a synchronization block in which your program will update the position estimate maintained in the x, y and heading fields.

    Updating these fields in a synchronization block will ensure that threads calling the getPose method are synchronized with the OdometricLocalizer object's thread so they do not receive inconsistent data. In other words, if another thread calls the getPose method while the run method is updating the x, y and heading fields, the other thread will be made to wait until execution of the code within the synchronized block is complete.

```
synchronized (this) {
}
```

20. Within the synchronization block, add statements to update x, y and heading.

```
x += deltaX;
y += deltaY;
heading += deltaTheta;
```

21. Add statements to limit the value of the heading to the range between –Pi and Pi.

    If your robot has turned clockwise more than Pi radians, switch to the equivalent heading value between 0 and –Pi and vice versa.

```
// limit theta to -Pi <= theta < Pi
if (heading > PI)
    heading -= TWO_PI;
else if (heading <= -PI)
    heading += TWO_PI;
```

22. Add statements to update the fields that hold the previous counter values.

```
previousLeftCounts = leftCounts;
previousRightCounts = rightCounts;
```

23. Save your files.

Testing Your Localizer Class

Your final task is to test and debug you OdometricLocalizer class. Earlier in this chapter you created the framework for the test program, LocalizerTest. You will need to extend that class to instantiate the objects needed for position tracking as well as add statements to cause your robot's position to change. Once it has reached its final location, your test program should use the getPose method to obtain the current position estimate from the localizer in order to display it. You will then be able to compare the estimated position with measurements you make of your robot's actual final position and heading.

Use the following steps to complete your test program and test your OdometricLocalizer class.

1. Add import statements for the motor, servo and shaft encoder classes your test program will be using.

   You can choose to use the AnalogShaftEncoder class you created as you completed the previous chapter, or you can use the identical class from the class library. This procedure illustrates using the class from the class library. If you would prefer to use the class you created, copy your AnalogShaftEncoder.java file into the folder for the current project and omit the import statement for the

AnalogShaftEncoder, below.

```
import com.ridgesoft.robotics.Motor;
import com.ridgesoft.robotics.ContinuousRotationServo;
import com.ridgesoft.robotics.ShaftEncoder;
import com.ridgesoft.robotics.AnalogShaftEncoder;
import com.ridgesoft.robotics.DirectionListener;
```

2. Define fields to hold references to the left and right motor objects.

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

3. Within the main method, instantiate the objects your test program will need. First, instantiate the shaft encoder objects. Use the same argument values to the AnalogShaftEncoder constructor that you used for your test program in the previous chapter.

```
ShaftEncoder leftEncoder = new AnalogShaftEncoder(
    IntelliBrain.getAnalogInput(4), 250, 750, 30,
    Thread.MAX_PRIORITY);
ShaftEncoder rightEncoder = new AnalogShaftEncoder(
    IntelliBrain.getAnalogInput(5), 250, 750, 30,
    Thread.MAX_PRIORITY);
```

4. Construct an instance of the OdometricLocalizer class.

   Provide references to the two shaft encoders constructed in the previous step. Provide the dimensions for the wheel diameter and track width. For your IntelliBrain-Bot educational robot, these are 2.65 inches and 4.55 inches, respectively. (If you prefer to use metric units, simply enter the metric values, 6.73 and 11.6 centimeters, instead.) Specify the value for the number encoder counts per revolution, which is 16 for your robot.

   Keeping track of its position is a high priority task for your robot. You will need to specify a high value for the priority of the localizer's thread. This will allow the localizer's thread to run ahead of other lower priority threads when multiple threads are ready to run at the same time; however, the localizer will not work correctly if it does not receive accurate measurements from the wheel encoders. Therefore, the wheel encoders' threads must be given higher priority than the localizer. Set the thread priority for the localizer to one priority level below the priority you gave to the shaft encoders.

   Finally, specify 30 milliseconds for the period at which the localizer should update its position estimate.

```
Localizer localizer = new OdometricLocalizer(
    leftEncoder, rightEncoder,
```

```
    2.65f, 4.55f, 16,
    Thread.MAX_PRIORITY - 1, 30);
```

5. Instantiate objects to control the left and right motors.

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

6. Add a method that uses the motors to move your robot forward for a specified number of milliseconds.

```
public static void forward(int time) {
    leftMotor.setPower(Motor.MAX_FORWARD);
    rightMotor.setPower(Motor.MAX_FORWARD);

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

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

7. Add a method that uses the motors to rotate your robot in place for a specified number of milliseconds.

```
public static void rotate(int time) {
    leftMotor.setPower(-8);
    rightMotor.setPower(8);

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

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

8. Use calls from the main method to these two methods to instruct your robot to move forward for 5 seconds, rotate for half a second and then move forward for another 3 seconds.

```
forward(5000);
rotate(500);
forward(3000);
```

9. Add statements to get and display your robot's pose once the movement is complete.

```
Pose pose = localizer.getPose();
    display.print(0,
        "x: " + (int)pose.x + ", y: " + (int)pose.y);
    display.print(1,
        "heading: " + (int)Math.toDegrees(pose.heading));
```

10. Build, load and test your program.

    Your robot should move straight ahead approximately 2½ feet, turn slightly left and precede straight another one foot or so.

    Using a measuring tape, measure the x and y coordinates of your robot's final location relative to its starting location. Use the center point between the wheels and along the line of the axles when determining the starting and ending locations. The origin of the x and y axis is the location of the center point when your robot started out. The positive x-axis is straight ahead in the direction your robot was facing when it started. The positive y-axis is 90 degrees counterclockwise from the x-axis. The values displayed on the LCD screen should be approximately equal to the distances you measured between the starting point and the ending point. The heading value displayed on the LCD screen should reflect the angle your robot is facing in its final position relative to the x-axis.

Localization Errors

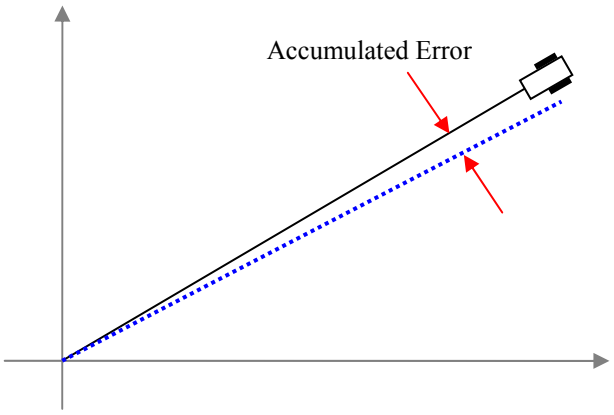


Figure 9-4 – Error Accumulation

Undoubtedly, your results from the previous exercise illustrate your localizer is not perfectly accurate. In fact, the errors can be significant. If you modify your test program such that your robot navigates a longer and more complex path, you will find the error increases as the length and complexity of its path increases. This is because errors accumulate when you use dead reckoning. Consider the simple situation illustrated in Figure 9-4. The robot started out by rotating approximately 40 degrees then drove straight ahead, as indicated by the solid line. The localizer's position estimates are indicated by the dotted line. In this hypothetical example, the localizer slightly under

estimated the angle of the initial turn by a few degrees. The gap between the two lines, which indicates the accumulated error, grew as the robot continued to travel. As the robot traveled further, the magnitude of the error grew.

It seems that this problem should be easy to fix by simply eliminating the errors. Unfortunately, doing so isn't easy. One way to reduce the errors is through more advanced hardware – for example, more precise sensors; however, more advanced sensors will likely add cost and may consume more power, take up more space or add weight, which are all things that may run counter to requirements for your robot.

The following paragraphs discuss a few sources of error for the localizer you have implemented.

Errors Due to Sensor Precision

The wheel position sensors and AnalogShaftEncoder objects used in this project measure the position of each wheel to the accuracy of 1/16 of a rotation, or 22.5 degrees. This comes about because the shaft encoder sensor works by counting spoke edges, of which each wheel has 16. Therefore, the shaft encoders can only determine the wheel position to a precision of plus or minus 1/16 of a revolution of the wheel. While this is reasonable for measuring straight line distances, it is more problematic when calculating your robot's heading. Previously, we determined the change in a differential drive robot's heading can be determined by the equation:

```
radiansPerCount = Pi * (wheelDiameter/trackWidth) / countsPerRevolution
```

Plugging in the values for wheelDiameter, trackWidth and counts per revolution for your IntelliBrain-Bot (2.65, 4.55 and 16, respectively) reveals that an error of one encoder count results in an error of approximately twenty degrees when estimating the heading of your robot. This is not very high precision when it comes to using dead reckoning to track position.

Examining the above equation, we see there are three variables that affect the precision to which your robot's heading can be estimated: wheel diameter, track width and encoder counts per wheel revolution. Changing any, or all, of these will affect the precision of heading estimates. For example, replacing the encoders with encoders that generate more counts per wheel revolution will increase the precision of the heading estimate. Replacing the wheel encoders with Nubotics WheelWatcher WW-01 encoders, which count 128 counts per wheel revolution, will result in heading estimates that are eight times more precise. With better encoders, errors will still accumulate, but the overall magnitude of the accumulated error will be significantly reduced.

Another way to improve the accuracy of the localizer is to change the geometry of your robot, increasing its track width or reducing the size of its wheels. Either of these changes will increase the precision of heading estimates.

Finally, rather than estimating the heading by deducing it from the wheel movements, you could add a compass sensor, using it to measure the heading and using the wheel encoders to measure only distance. A compass sensor, such as the Devantech CMPS03, would enable your robot to track its heading relative to an external reference, the Earth's magnetic field. Because the Earth's magnetic field is an external reference, the heading estimate will not be subject to accumulated errors, as is the case with dead reckoning. However, a magnetic compass is subject to other errors, such as local interference caused by ferrous metals or electrical equipment.

Errors Due to Algorithm Assumptions

The dead reckoning algorithm has several limitations that lead to errors. First, it does not use an external reference to determine position. Therefore, there is no way to identify and correct for errors, they simply accumulate. For example, when you travel in a car in the country, you may estimate your current location based on time and average speed. If you travel for a long time without seeing a landmark or a sign, your estimate of where you are will become less accurate. This is usually not a big problem because you will frequently see landmarks, intersections and signs that enable you to make adjustments to correct for errors. Similarly, if your robot knows about landmarks in its environment and can sense them, it could use that information to correct its position estimate. For example, it could use its sonar sensor to precisely measure the distance to walls, using this information to correct its position estimate.

Secondly, dead reckoning, as we have implemented it, uses the assumption that your robot travels in a straight line between position updates. This is a reasonable assumption if the position updates are frequent, but if the position updates are not frequent enough, the assumption will not hold. Clearly, if you adjusted the update period so your robot only updated its position estimate every few minutes, then programmed it to follow a wandering path during that period, the final position estimate would be totally erroneous. Errors due to the straight line assumption can be reduced by frequent updates, but each update costs computing time, taking computing time away from other tasks your program might need to do.

Summary

In this chapter you created a class that used dead reckoning to track your robot's position. Dead reckoning is dependent on accurate measurement of incremental movements of your robot. The precision of measurements will greatly impact the accuracy to which your robot can track its position. Furthermore, with dead reckoning, errors accumulate as your robot moves. As a result, the position calculated by your localizer will continually drift from your robot's actual position as your robot travels further.

You can counter the problem of accumulated errors by enhancing your robot to sense external references such as landmarks or the Earth's magnetic field. By using external references, your robot can detect and remove accumulated errors. You can also use higher precision encoders or change the geometry of your robot to reduce, but not eliminate, accumulated errors.

Exercises

1. Measure and record the diameter of your robot's wheels and the track width of your robot.

2. How far will your robot travel if both wheels rotate one revolution in the same direction?

3. How many degrees will your robot's heading change if both wheels rotate one revolution in opposite directions?

4. If you put 25% larger wheels on your robot, how will it affect the number of degrees your robot's heading will change when the wheels rotate one revolution in opposite directions?

5. If you increase the track width of your robot by 50% how will it affect the number of degrees your robot's heading will change when the wheels rotate one revolution in opposite directions?

6. If both wheels on your robot turn at the same speed, how far will your robot move with each encoder count?

7. How many counts do the encoders on your robot increase or decrease with each full revolution of the wheel?

8. What is the maximum distance your robot can move without incrementing or decrementing the value of either encoder's counter?

9. What would be the maximum distance your robot could move without incrementing or decrementing either encoder's counter if you were to switch to using encoders that count 128 counts per wheel revolution?

10. If the wheels on your robot each rotate one full encoder count in opposite directions, how many degrees will your robot's heading change?

11. What is the maximum possible angle your robot can rotate in place without the encoder counter values changing?

    Hint: Each wheel can rotate a maximum of just under one encoder count in opposite directions without incrementing or decrementing either encoder's counter value.

12. What effect will errors in your measurements of wheel diameter and track width have on dead reckoning position calculations?

13. How will switching to encoders that count 128 counts per revolution affect the precision to which your OdometricLocalizer class can measure the heading of

your robot?

14. How would doubling the track width of your robot affect the precision to which your OdometricLocalizer class can measure the heading of your robot?

15. How would decreasing the diameter of the wheels by 25% affect the precision to which your OdometricLocalizer class can measure the heading of your robot?

16. Double the time values used to control your robot's motion in your LocalizerTest program. Record actual position and estimated position for 5 test runs before and 5 test runs after you make the change. Calculate the average error of the x-coordinate for both sets of test runs. How did doubling the times affect the total error?

CHAPTER 10

Navigation

Whether you are building a robot to explore another planet, compete in the DARPA Grand Challenge, rescue victims of a disaster or just satisfy your own curiosity, it's likely you'll want it to have the ability to navigate from place to place on its own. This chapter will show you how to do that by building on concepts and software components from previous chapters.

The chapter on maneuvering your IntelliBrain™-Bot Deluxe educational robot demonstrated how you could turn the motors on and off using timing to accomplish simple maneuvers; however, timed maneuvering fails to account for varying conditions your robot will encounter. Your robot's actual performance will vary significantly as battery level, floor texture, and other factors change. The timed maneuvering technique has no way to compensate for these variations because it does not incorporate feedback of your robot's actual performance.

In this chapter, you will learn how to use position feedback to improve your robot's ability to navigate. By incorporating position feedback, your robot will be able to act on real-time measurements of its actual performance rather than blindly following an inflexible sequence of operations.

Fundamentals of Navigation

In order to navigate effectively, your robot must:

1. know where it wants to go,
2. know its current location and heading,
3. determine the direction to its destination,
4. steer to and maintain the heading to its destination, and
5. stop when it has reached its destination.

Knowing Where to Go

Knowing where to go is easy, at least for your navigator, because you will tell it where to go. Your navigator's job is to get your robot to its destination, not to determine its destination. In this chapter, you will develop a program to test the navigator you develop. Your test program will tell the navigator where to go.

Considering this, your navigator will only need to provide a means for it to be told where your robot should go next. In other words, it will need to provide methods that allow higher level software to set the next goal. The following four methods will serve this purpose:

1. moveTo - move to a specified location,
2. turnTo - turn to face a particular direction,
3. go - move continuously in one direction, and
4. stop.

Creating Your Navigator Interface

Let's begin developing your navigator by developing a Navigator interface. This interface will define the methods listed above. You will also need to create a project and a test program that will allow you to test your navigator.

Use the following procedure to create a project, a test program and the Navigator interface.

1. Create a new project named "NavigatorTest."

2. Create a new class named "Navigator" and adjust the declaration to specify it is an interface, not a class.

```
public interface Navigator {
}
```

3. Add the four methods identified above to the interface. In addition to the obvious arguments for these methods, include a boolean argument in the moveTo and turnTo methods to indicate if the method should wait until the movement completes before returning, or return immediately.

```
public void moveTo(float x, float y, boolean wait);
public void turnTo(float heading, boolean wait);
public void go(float heading);
public void stop();
```

4. Add statements to the main method of your NavigatorTest class to display the name of your program on the LCD screen. Also, add the needed import statements.

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

5. Build, load and test your program.

   At this point, your program will only display its name and exit.

Implementing Your Navigator Class

In order for your Navigator interface to be useful, at least one class must implement it. Your Navigator interface is very general. It could be used for a large variety of robot designs, from two legged robots, to car like robots, to differential drive robots such as your IntelliBrain-Bot educational robot. You must take into account the mechanics of your robot when you design and implement your navigator. The mechanical design dictates how your navigator can steer your robot. Designing a navigator for a robot that has four wheels and steers like a car is significantly different from designing a navigator for a differential drive robot. Fortunately, the Navigator interface is general. It isn't tied to one specific mechanical design or another. It defines methods that are equally relevant to navigating robots that may use significantly different mechanical designs.

Since your IntelliBrain-Bot educational robot is based on the very common differential drive design, it is preferable to implement a navigator that is suitable to any differential drive robot rather than restrict it to just your IntelliBrain-Bot educational robot.

Use the following procedure to create the basic structure for your DifferentialDriveNavigator class.

1. Create a new class named "DifferentialDriveNavigator." Your navigator will need its own thread to manage navigation tasks while your robot is doing other things, such as tracking its location and planning where to go next. Declare the class as an extension of the Thread class. Also declare that it implements your Navigator interface.

   ```
   public class DifferentialDriveNavigator
       extends Thread implements Navigator {
   }
   ```

2. Add import statements for the Motor and Pose classes as well as the Localizer interface.

   Instead of using these classes from the class library, you may choose to use your implementation, which you developed in previous chapters, of the same classes. If you do this, you must omit the import statements for those classes.

   ```
   import com.ridgesoft.robotics.Motor;
   import com.ridgesoft.robotics.Localizer;
   import com.ridgesoft.robotics.Pose;
   ```

3. Define constants for Pi and 2 Pi. You will need these later.

   ```
   private static final float PI = 3.14159f;
   private static final float TWO_PI = PI * 2.0f;
   ```

4. Define fields to refer to the two motor objects.

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

5. Define the run method including statements to catch exceptions, as well as loop periodically to perform the control operations to navigate your robot.

```
public void run() {
    try {
        while (true) {
            // insert navigation control code here
            Thread.sleep(period);
        }
    }
    catch (Throwable t) {
        t.printStackTrace();
    }
}
```

6. Define the field for control thread's period.

```
private int period;
```

7. Define a simple state machine to carry out navigation control functions on each iteration of the while loop. Your navigator needs to support the following four states: moving to a particular point, rotating to face a certain direction, going straight ahead and stopped.

```
switch (state) {
    case MOVE_TO:
        goToPoint();
        break;
    case GO:
        goHeading();
        break;
    case ROTATE:
        doRotate();
        break;
    default:      // stopped
        break;
}
```

8. Add statements to declare constants for the four states.

```
private static final int STOP = 0;
private static final int GO = 1;
private static final int MOVE_TO = 2;
private static final int ROTATE = 3;
```

9. Declare the field to hold your navigator's state.

```
private int state;
```

10. Define the state machine's three methods.

    Since these methods will be run by your navigator's thread, they will need to be synchronized with other threads calling the four methods you defined in the Navigator interface.

```
private synchronized void goToPoint() {
}

private synchronized void doRotate() {
}

private synchronized void goHeading() {
}
```

11. Define the four methods required by your Navigator interface. Declare them as synchronized so execution of these methods will be synchronized with your navigator's internal control operations.

```
public synchronized void moveTo(float x, float y, boolean wait) {
}

public synchronized void turnTo(float heading, boolean wait) {
}

public synchronized void go(float heading) {
}

public synchronized void stop() {
}
```

12. Build your program.

Knowing Your Robot's Location

In the previous chapter you learned how to track your robot's position using shaft encoders and dead reckoning. Your robot can keep track of its position using AnalogShaftEncoder objects and an OdometricLocalizer object. You can feed position data back into your navigator by interfacing it to an instance of your OdometricLocalizer class, as shown in Figure 10-1.

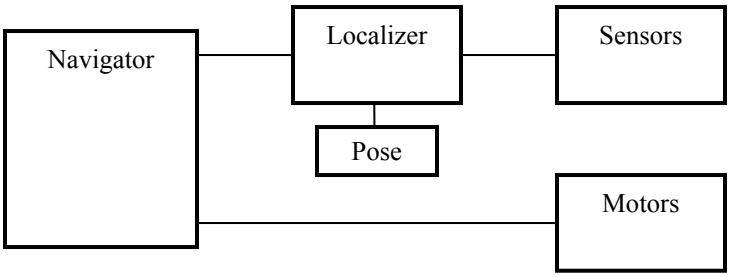


Figure 10-1 - Navigation and Localization Class Diagram

Extend your NavigatorTest program to instantiate position tracking objects using the following procedure.

1. Add statements to create AnalogShaftEncoder objects for the left and right wheels.

   You can use the class you created or use the identically named class from the class library. If you use your own class, ensure you place a copy of your AnalogShaftEncoder Java source file in the folder for this project. If you choose to use the class from the class library, add the required import statement.

   ```
   ShaftEncoder leftEncoder = new AnalogShaftEncoder(
       IntelliBrain.getAnalogInput(4), 250, 750, 30,
       Thread.MAX_PRIORITY);
   ShaftEncoder rightEncoder = new AnalogShaftEncoder(
       IntelliBrain.getAnalogInput(5), 250, 750, 30,
       Thread.MAX_PRIORITY);
   ```

2. Add a statement to create an OdometricLocalizer object. Again, you can use the class you created or the identically named class from the class library.

   ```
   Localizer localizer = new OdometricLocalizer(
       leftEncoder, rightEncoder,
       2.65f, 4.55f, 16,
       Thread.MAX_PRIORITY - 1, 30);
   ```

3. Build your program.

Steering to Follow a Heading

In order to follow a heading in a more-or-less straight line, your navigator will need to steer your robot by varying the power applied to each wheel. Your robot will move approximately straight ahead when your navigator applies the same power to both wheels. Your navigator can steer your robot left or right as it moves forward by applying slightly different amounts of power to each wheel. This will cause one wheel to turn slightly faster than the other, making your robot change direction. The further off course your robot is, the more aggressively your navigator will need to steer. Therefore, the larger the error, the larger the power differential your navigator will need to apply to steer your robot back on course. As you did in the line following chapter, you can use a proportional control algorithm to accomplish steering.

Consider the situation shown in Figure 10-2. Your robot is heading forward, but slightly off its desired heading. In this case, your navigator can correct for the error by applying slightly more power to the right wheel than to the left wheel. Your navigator can calculate how much power to apply to each wheel by using the following equations:

leftWheelPower = drivePower – offset
rightWheelPower = drivePower + offset

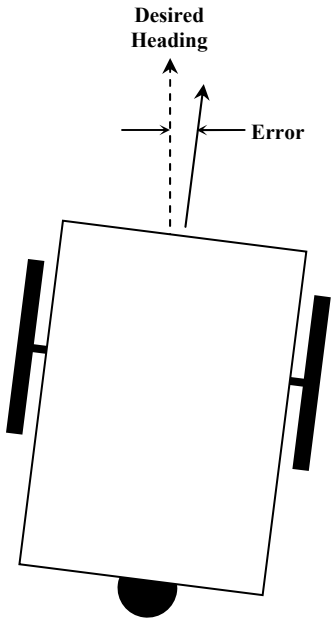


Figure 10-2 - Heading Error

The drivePower is the base power level your navigator will apply to move your robot straight ahead. The offset value controls the difference in power applied to the two wheels. The power differential can be made proportional to the heading error using the equation:

offset = gain * error

The difference in power applied to the wheels will become larger as the heading error increases, causing your navigator to respond more forcefully when the error is large. When the error is zero, your navigator will apply the same level of power to both wheels.

The gain value in the previous equation is a constant that controls how aggressively your navigator will respond to error. Your navigator will respond aggressively if you use a large value for the constant; however, too large a gain will cause your robot to over-steer and oscillate wildly. Choosing a gain constant that is too small will cause your navigator to be sluggish in responding to error. Experimentation has shown that a gain value of 25.0 works well for this particular situation.

In order to keep your robot heading to its destination, your navigator will need to periodically adjust the power it applies to the wheels. Otherwise, your robot will quickly wander off course. Your navigator can do this by periodically taking into account its current position and repeating all of the navigation calculations. You have already set this up by calling the goHeading method periodically from the run method of your DifferentialDriveNavigator class. Each time the run method calls the goHeading method it will adjust the power levels according to the heading error.

Implement and test your goHeading method using the following procedure.

1. Define fields to hold: a reference to the localizer object, the target heading, the base power level and the controller gain.

```
private Localizer localizer;
private float targetHeading;
private int drivePower;
private float gain;
```

2. Add statements to your goHeading method to get the pose from the localizer and then determine the heading error.

```
Pose pose = localizer.getPose();
float error = targetHeading - pose.heading;
```

3. Add statements to normalize the error to a value between -Pi and Pi.

```
if (error > PI)
    error -= TWO_PI;
else if (error < -PI)
    error += TWO_PI;
```

4. Add a statement to calculate the offset in power level to be applied to the wheels.

   Integer values are required when setting the motor power. Therefore, convert the result to an integer. Add 0.5 prior to converting to an integer to round to the nearest integer.

```
int offset = (int)(gain * error + 0.5f);
```

5. Add statements to update the power being applied to the motors.

```
leftMotor.setPower(drivePower - offset);
rightMotor.setPower(drivePower + offset);
```

6. Create a constructor for your DifferentialDriveNavigator class.

   The constructor will need to take references to motor objects, a reference to the localizer object, the base motor power level, the controller gain, the thread priority and the thread period as arguments.

```
public DifferentialDriveNavigator(
    Motor leftMotor, Motor rightMotor,
    Localizer localizer, int drivePower, float gain,
    int threadPriority, int period) {
}
```

7. Add statements to initialize the field using the arguments provided to the constructor and also initialize and start your navigator's thread.

```
this.leftMotor = leftMotor;
this.rightMotor = rightMotor;
this.localizer = localizer;
this.drivePower = drivePower;
this.gain = gain;
this.period = period;
state = STOP;
setPriority(threadPriority);
setDaemon(true);
start();
```

8. Next begin implementing the go method (not the goHeading method) by adding a statement to set the target heading field.

   Use a helper method to normalize the heading argument to a value between -Pi and Pi.

```
targetHeading = normalizeAngle(heading);
```

9. Add another statement to the go method to set the navigator state to "GO."

```
state = GO;
```

10. Implement the normalizeAngle helper method.

```
private float normalizeAngle(float angle) {
    while (angle < -PI)
        angle += TWO_PI;
    while (angle > PI)
        angle -= TWO_PI;
    return angle;
}
```

11. Add statements to the stop method to stop the motors and to set your navigator's state.

```
  leftMotor.setPower(Motor.STOP);
  rightMotor.setPower(Motor.STOP);
  state = STOP;
```

12. Add statements to your NavigatorTest class to construct motor objects.

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

13. Follow this with a statement to construct a DifferentialDriveNavigator object. Use 8 for the power level and 25.0 for the gain. Set the thread priority one priority level below that of the OdometricLocalizer.

```
Navigator navigator = new DifferentialDriveNavigator(
    leftMotor, rightMotor,
    localizer,
    8, 25.0f,
    Thread.MAX_PRIORITY - 2, 50);
```

    Setting the priority level below that of the OdometricLocalizer will ensure the wheel encoders and localizer take priority over the navigator. The navigator will not function properly if the localizer doesn't provide accurate, up-to-date information. Therefore, the localizer should run at a higher priority than the navigator.

14. In your test class, implement a simple method to instruct your navigator to steer to a heading for some number of seconds and then stop.

```
public static void navigateHeading(
        Navigator navigator, float degrees, int seconds) {
    navigator.go((float)Math.toRadians(degrees));
    try {
        Thread.sleep(seconds * 1000);
    }
    catch (Exception e) {}
    navigator.stop();
}
```

15. In the main method, following the construction of the Navigator object, add a call to the navigateHeading method to instruct your navigator to steer your robot at a heading of 45 degrees for 10 seconds.

```
navigateHeading(navigator, 45.0f, 10);
```

16. Build, load and test your program.

    Your robot should immediately turn counterclockwise 45 degrees from its starting position and then drive in straight line for 10 seconds.

Determining Where to Head

If your navigator knows your robot's current position and heading as well as its desired destination, it can use trigonometry to calculate the heading that gives the shortest path to the destination.

Figure 10-3 depicts your robot heading toward its destination. The location of the destination relative to your robot's current position in Cartesian coordinates is (xError,

yError), the x and y components of the error, which, in this case, is the difference between where your robot is and where it wants to go. The heading is the angle to the destination.

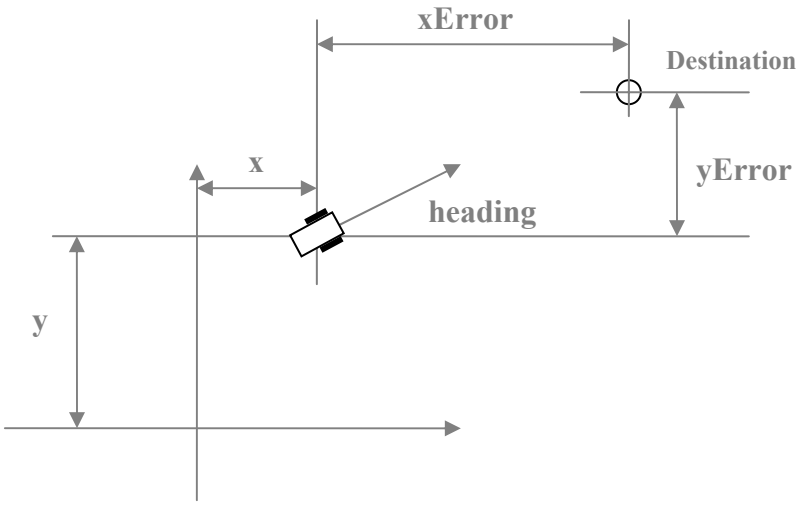


Figure 10-3 - Navigating to a Point

Your navigator can calculate xError and yError by subtracting the coordinates of your robot's current position from the coordinates of the destination:

xError = destinationX – x
yError = destinationY – y

Recalling from trigonometry, the heading is the arc tangent of yError divided by xError:

heading = arctan(yError / xError)

Determining When to Stop

Your navigator will need to stop your robot when it reaches its destination. Stopping is simply a matter of turning off the motors once the destination has been reached; however, your robot will rarely reach its precise destination. Getting close to the desired destination usually isn't difficult, but getting exactly to the precise point of the destination is problematic. Your robot is not an extremely high precision machine; it will not navigate with perfect accuracy. Although it can navigate to close proximity of its destination, if you insist on extreme precision, you will discover that your robot will get close to its destination and then begin to flounder around attempting to reach the precise destination. Instead of insisting on perfect navigation, your robot will perform better if you relax your accuracy requirements and program your robot to stop within reasonable proximity of its destination. Fortunately, in this case, close is good enough.

Your robot could use the Pythagorean Theorem to determine its proximity to its destination. Unfortunately, this calculation requires calculating a square root, which

takes a lot of time to compute. Instead, you can use the sum of the absolute values of the two error terms, xError and yError, as a rough approximation of how far your robot is from its destination. This will not yield the exact distance, but it will be good enough and far easier to calculate.

Coordinating Threads

The moveTo method we defined in the Navigator interface includes an argument to indicate whether the method should wait until the operation completes before returning. If the caller indicates it wants to wait, your moveTo method must suspend execution of the calling thread while it waits for your navigator to move your robot to its destination. The Java language provides two methods that facilitate this very thing: wait and notify. These methods are part of the Object class, so they are available on every object. The way they work is that one thread will call the wait method of an object. This will suspend the thread until another thread calls the notify method. You can use this facility to coordinate two threads. By placing a call to the wait method in the moveTo method and placing a call to the notify method where your navigator recognizes your robot has reached its destination, you can suspend a thread that calls the moveTo method until the navigator completes the move.

Going to a Point

With the knowledge of how to determine where to head and when to stop, you can now implement the goToPoint method in your DifferentialDriveNavigator class.

Use the following procedure to implement the goToPoint method.

1. Add fields to hold the target destination.

```
private float destinationX;
private float destinationY;
```

2. Add a field to hold the threshold that determines when your robot is close enough to its destination that it should stop.

```
private float goToThreshold;
```

3. Begin implementing the goToPoint method by adding statements to get the current pose from the Localizer and calculate the x and y components of the positional error.

```
Pose pose = localizer.getPose();
float xError = destinationX - pose.x;
float yError = destinationY - pose.y;
```

4. Add statements to determine if your robot is close enough to its destination to stop or not.

Use the sum of the absolute values of the two error components, as discussed above, to determine if your robot is close enough to its destination. If this value is less than the threshold value, then your robot is close enough to stop.

```
float absXError = (xError > 0.0f) ? xError : -xError;
float absYError = (yError > 0.0f) ? yError : -yError;
if ((absXError + absYError) < goToThreshold) {
    // stop
}
else {
    // adjust heading and continue on
}
```

5. Add statements to stop the motors and switch the state to STOP when the destination has been reached.

```
leftMotor.setPower(Motor.STOP);
rightMotor.setPower(Motor.STOP);
state = STOP;
```

6. Add a call to the notify method after stopping the motors.

```
notify();
```

   If a thread was suspended in the moveTo method, this call will resume that thread, allowing the moveTo method to return.

7. Add statements to update the target heading to the destination and then head there if the destination hasn't yet been reached.

```
targetHeading = (float)Math.atan2(yError, xError);
goHeading();
```

8. Add statements to the moveTo method to set the destination and to change your navigator's state.

```
destinationX = x;
destinationY = y;
state = MOVE_TO;
```

   Changing your navigator's state to MOVE_TO will cause your navigator's thread to begin maneuvering your robot toward the destination.

9. Add statements to the moveTo method to call the wait method if the caller has specified it wants to wait for the move to be completed.

```
if (wait) {
    try {
        wait();
```

```
        }
        catch (InterruptedException e) {}
    }
```

The call to the wait method will cause the calling thread to suspend until your navigator's thread calls the notify method. The wait method may be interrupted, which is why the try-catch block is needed.

10. Update the constructor of your DifferentialDriveNavigator class to include an argument to initialize the threshold.

```
public DifferentialDriveNavigator(
    Motor leftMotor, Motor rightMotor,
    Localizer localizer, int drivePower, float gain,
    float goToThreshold,
    int threadPriority, int period) {
}
```

11. Within the constructor, add a statement to initialize the threshold.

```
this.goToThreshold = goToThreshold;
```

12. Add the argument for the threshold to the call to the constructor in your NavigatorTest class. Use 0.5 inches as the stop threshold.

```
Navigator navigator = new DifferentialDriveNavigator(
    leftMotor, rightMotor,
    localizer,
    8, 25.0f,
    0.5f,
    Thread.MAX_PRIORITY - 2, 50);
```

13. Implement a driveForward method that will instruct the navigator to move your robot forward a specified distance.

```
public static void driveForward(
        Navigator navigator, float distance) {
    navigator.moveTo(distance, 0.0f, true);
}
```

14. Disable the call to navigateHeading by inserting comment marks at the beginning of the line and by adding a call to driveForward, instructing your robot to drive straight ahead 100 inches.

```
//      navigateHeading(navigator, 45.0f, 10);
        driveForward(navigator, 100.0f);
```

15. Build, load and test your program.

    Your robot should drive straight ahead, stopping after it has traveled 100 inches.

Rotating in Place

The final feature you will need to implement to complete your Navigator class is the ability to rotate in place. You will need to implement the doRotate and turnTo methods. These methods are analogous to the goToPoint and moveTo methods you just implemented, but they will rotate your robot in place rather than moving it to a new location.

Use the following procedure to implement and test in place rotation.

1. Add statements to the turnTo method to set the target heading and the navigator's state. Also implement statements to wait if the caller requests to wait.

```
targetHeading = normalizeAngle(heading);
state = ROTATE;
if (wait) {
    try {
        wait();
    }
    catch (InterruptedException e) {}
}
```

2. Add fields for the motor power level to use when rotating and the threshold to indicate when to stop rotating.

```
private int rotatePower;
private float rotateThreshold;
```

3. Update the constructor to take arguments to initialize the new fields.

```
public DifferentialDriveNavigator(
    Motor leftMotor, Motor rightMotor,
    Localizer localizer,
    int drivePower, int rotatePower, float gain,
    float goToThreshold, float rotateThreshold,
    int threadPriority, int period)
```

4. Within the constructor, initialize the new fields.

```
this.rotatePower = rotatePower;
this.rotateThreshold = rotateThreshold;
```

5. Add statements to the doRotate method to retrieve the current heading from the localizer and then calculate the error between the current heading and the desired

heading. Normalize the error angle to be between -Pi and Pi.

```
Pose pose = localizer.getPose();
float error = targetHeading - pose.heading;
if (error > PI)
    error -= TWO_PI;
else if (error < -PI)
    error += TWO_PI;
```

6. Add statements to determine whether the navigator should: stop, rotate counterclockwise or rotate clockwise.

```
float absError = (error >= 0.0f) ? error : -error;
if (absError < rotateThreshold) {
    // stop
}
else if (error > 0.0f) {
    // rotate counterclockwise
}
else {
    // rotate clockwise
}
```

7. Add statements within the stop branch to stop your robot and then notify the waiting thread.

```
leftMotor.setPower(Motor.STOP);
rightMotor.setPower(Motor.STOP);
state = STOP;
notify();
```

8. Add statements to the counterclockwise branch to power the motors for counterclockwise rotation of your robot.

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

9. Add statements to the clockwise branch to power the motors for clockwise rotation.

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

10. Update the call to the DifferentialDriveNavigator constructor in your NavigatorTest class to provide the two new arguments. Specify 6 for the power to use when rotating and 0.08 radians (~5 degrees) for the threshold.

```
Navigator navigator = new DifferentialDriveNavigator(
        leftMotor, rightMotor,
        localizer,
```

```
            8, 6,
            25.0f,
            0.5f, 0.08f,
            Thread.MAX_PRIORITY - 2, 50);
```

11. Implement a method in your test class to rotate your robot a specified number of degrees.

```
public static void rotate(
        Navigator navigator, float degrees) {
    navigator.turnTo((float)Math.toRadians(degrees), true);
}
```

12. Comment out the call to the driveForward method and then add a call to the rotate method to rotate your robot 180 degrees.

```
//  driveForward(navigator, 100.0f);
    rotate(navigator, 180.0f);
```

13. Build, load and test your program.

    Your robot should rotate in place approximately 180 degrees.

Navigating Patterns

You can program your robot to navigate more complex patterns using sequences of the four basic navigation functions provided by your navigator. We will implement methods in your test program to navigate your robot through a square pattern and a figure eight pattern.

Navigating a Square

Navigating a square is a simple matter of instructing your robot to move to each of the four corners of the square in the correct sequence. Conveniently, you don't need to give the navigator detailed instructions of what movements are required to get from one corner of the square to the next. Instead, your program can simply tell the navigator to go to one corner after another. Your DifferentialDriveNavigator class takes care of the details of getting to each destination.

Use the following procedure to program your robot to navigate a square pattern.

1. Create a method in your NavigatorTest class that will navigate a square of a specified size.

```
public static void navigateSquare(
        Navigator navigator, float size) {
    navigator.moveTo(size, 0.0f, true);
    navigator.moveTo(size, -size, true);
    navigator.moveTo(0.0f, -size, true);
    navigator.moveTo(0.0f, 0.0f, true);
```

```
        navigator.turnTo(0.0f, true);
    }
```

The call to the turnTo method instructs your robot to turn so it faces the direction it was facing when your program started. If your navigator works perfectly, your robot will end up where it started, facing the same direction it was facing when it started.

2. Comment out the call to the rotate method and add a call to the new method specifying 16 inches as the size of the square.

```
//    rotate(navigator, 180.0f);
    navigateSquare(navigator, 16.0f);
```

3. Build, load and test your program.

   Your robot should navigate a 16 inch square pattern.

Navigating a Figure Eight

You can program your robot to navigate a figure eight pattern in the same way you programmed it to navigate a square. You can form a simple figure eight by connecting the four corners of a rectangle, as shown in Figure 10-4.

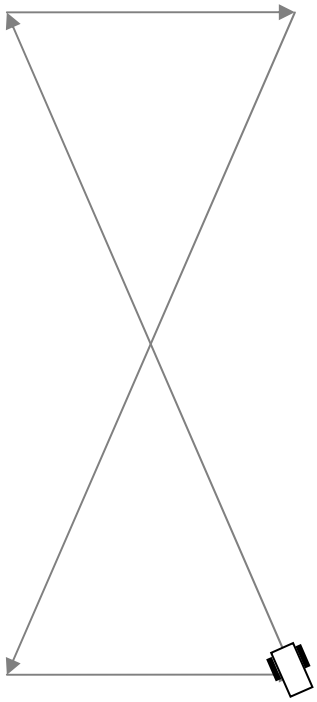


Figure 10-4 - Navigating a Figure Eight

Use the following procedure to program your robot to navigate a figure eight pattern.

1. Create a method in your NavigatorTest class that will navigate a figure eight of a specified size.

```
public static void navigateFigureEight(
        Navigator navigator, float height, float width) {
    navigator.moveTo(height, width, true);
    navigator.moveTo(height, 0.0f, true);
    navigator.moveTo(0.0f, width, true);
    navigator.moveTo(0.0f, 0.0f, true);
    navigator.turnTo(0.0f, true);
}
```

2. Comment out the call to the navigateSquare and add a call to the new method specifying 48.0 inches as the height and 32.0 inches as the width.

```
//    navigateSquare(navigator, 16.0f);
    navigateFigureEight(navigator, 48.0f, 32.0f);
```

3. Build, load and test your program.

   Your robot should navigate a figure eight pattern similar to the one shown in Figure 10-4.

Precision and Accuracy of Navigation

As you test your navigator, you will undoubtedly observe that your robot's precision and accuracy of navigation are far from perfect because no robot is a perfect machine. Its sensors will never yield perfectly accurate measurements. Sensors have limited precision. In addition, the algorithms you use to control your robot are based on assumptions and simplifications. While it is tempting to try to build a perfect robot, this will drive up cost. A great deal of current robotics research focuses on developing algorithms that allow imperfect robots to work effectively despite significant limitations.

Table 10-1 lists a number of the most significant sources of error that affect the precision with which your robot can navigate.

Table 10-1 – Major Sources of Error

Error Sources	Description
Calibration	The accuracy of the wheel diameter and wheel base measurements affects the accuracy of the localizer.
Encoder Quantization	The accuracy to which the wheel encoders are able to measure the position of the wheels directly affects the accuracy of the localizer. Using wheel encoders that provide more counts per revolution, such as the Nubotics WheelWatcher WW-01 encoder, will reduce the error due to encoder quantization.
Wheel Slippage	Any slippage of the wheels will result in localization errors.

Localization Technique	The dead reckoning localization method is based on self-centric encoder measurements. This results in accumulation of error because your robot has no external reference from which it could recalibrate its position. Incorporating external references such as landmarks, Earth's magnetic field (compass) or satellites (GPS) would help improve the accuracy of the localizer.

Summary

In this chapter you learned four fundamentals of basic robot navigation: 1) knowing your robot's destination, 2) knowing your robot's location, 3) knowing which direction your robot needs to head, and 3) knowing when your robot has reached its destination.

You defined a Navigator interface. It specifies the basic functions of a navigator without making assumptions about the design of your robot or the techniques an implementing class will employ. By designing it in this way, you ensured other classes will not become tied to the specific design of one particular robot or to a specific navigation technique. The method you wrote to navigate a square pattern is a clear example of this; it could be used without modification to instruct an entirely different type of robot, such as a two legged robot to navigate a square path. The Navigator interface hides the details of whether navigating from place to place involves moving legs or turning wheels, making it easier to reuse and maintain classes that use the interface.

You learned how you can use trigonometry to calculate a heading that leads to a particular destination, as well as how you can use proportional control to steer a differential drive robot to follow a heading.

You used the wait and notify methods to enable one thread to suspend while it waits for another thread to complete some operation. A thread that needs to wait calls the wait method of an object. This will cause the thread to suspend until another thread calls the notify method of the same object.

Using the basic navigation methods defined by the Navigator interface, you wrote methods to perform more complex maneuvers by writing methods to navigate a square and navigate a figure eight.

As you experimented with your navigation program you observed that your robot isn't able to navigate perfectly. The facts that robots aren't perfect and building higher precision robots adds cost are fundamental issues robotics software developers have to contend with. While it is tempting to try to produce a robot that is perfect, that isn't possible. Precision can be increased and errors can be reduced, but not eliminated. Creating strategies to cope with the practical limitations of robots is an interesting problem that will keep robotics researchers busy for many years to come.

Exercises

1. Experiment with the pattern navigation methods you created. Does your robot navigate as you expect it to? Does your robot navigate the pattern perfectly each time? Explain why or why not.

2. Using your driveForward method, program your robot to drive 100 inches. Measure the average distance your robot moves over several runs of your program. Calibrate the value of the wheel diameter such that your robot moves an average of 100 inches over several runs.

3. Using your rotate method, calibrate the track width value such that your robot is as accurate as possible at rotating an average of 180 degrees over several runs.

4. Modify your test program to make your robot travel in a square which is 10 inches on a side. Run your program while observing how close your robot gets to its starting point. Modify your program to increase the square to 50 inches on a side and rerun the test. Is your robot more able to get back to its starting point when the square is larger or smaller? Why?

5. Program your robot to navigate a new shape that you choose.

6. Try increasing and decreasing the stop thresholds your navigator uses. How does increasing the stop threshold affect your robot's navigation? How does decreasing the stop threshold affect your robot's navigation?

7. Change the drive power. How does this affect your robot's navigation?

8. Change the rotate power. How does this affect your robot's navigation?

9. Double the gain constant. How does this affect your robot's navigation?

10. Halve the gain constant. How does this affect your robot's navigation?

CHAPTER 11

Behavior-based Control

Behavior-based control is a frequently used method of controlling robots that is rooted in modeling the behavior of animals. Robotics researchers drew inspiration from observations regarding animal behavior. Researchers have observed that animals exhibit a number of distinct behaviors that operate independent of each another. For example, two behaviors ground squirrels exhibit are foraging and flight. Squirrels spend much of their time searching for and gathering food – foraging behavior; however, if a squirrel senses it is threatened by another animal, it will cease foraging and instead scurry to safety – flight behavior. These two behaviors seem to be completely independent of each other. It would be unusual to see a squirrel stop to pickup another nut while it senses an immanent threat to its safety. It appears squirrels totally suppress their foraging behavior when they sense their survival is at stake. The behavior-based control methodology is based on this concept of combining multiple independent simple behaviors, whereby activation of a higher priority behavior suppresses all lower priority behaviors.

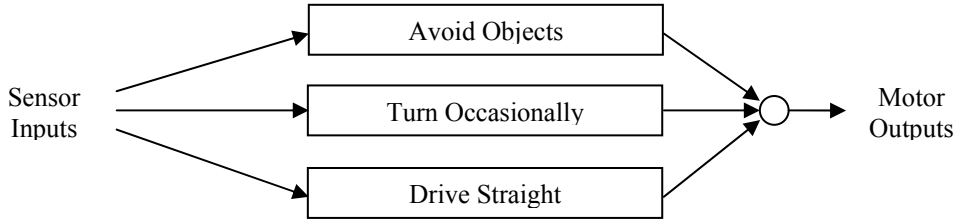


Figure 11-1 – Vacuum Cleaner Behaviors

In this chapter, you will program your IntelliBrain™-Bot Deluxe educational robot using a behavior-based control scheme such that it behaves as if it were a robotic vacuum cleaner. Figure 11-1 illustrates how you can combine three simple behaviors. Through the interaction of these three behaviors, a more complex overall behavior emerges. The general idea is for your vacuum cleaner to wander around somewhat randomly such that it eventually visits the entire surface of the floor. While doing this, your vacuum cleaner needs to avoid running into things. If your vacuum cleaner senses it is going to run into something, it should suppress its normal behaviors to give priority to avoiding a collision. The overall behavior of your vacuum cleaner can be realized by combining three simple behaviors: 1) driving straight ahead, 2) occasionally making a random turn and 3)

avoiding obstacles. Figure 11-1 depicts this. The diagram is arranged with the highest priority behavior, Avoid Objects, at the top and the lowest priority behavior, Drive Straight, at the bottom.

Following the principles of behavior-based control, each of the three behaviors will operate independently, reading sensor inputs and making its own control decisions. Your program will need a way to combine these behaviors, arbitrating between them to select which behavior is given control of your robot at any particular instant. The circle at the right in Figure 11-1 represents the arbitration function.

Defining Behavior Objects

In order to implement this behavior-based control scheme, we will need to translate the behavior-based control concept into a set of Java objects that will work together to control your robot. Figure 11-2 illustrates the object model we will use to construct your program. The blocks labeled *Localization & Navigation* and *Sensors & Actuators* depict objects and topics we've covered in previous chapters. Our focus in this chapter will be the block at the left, labeled *Behavior Control*.

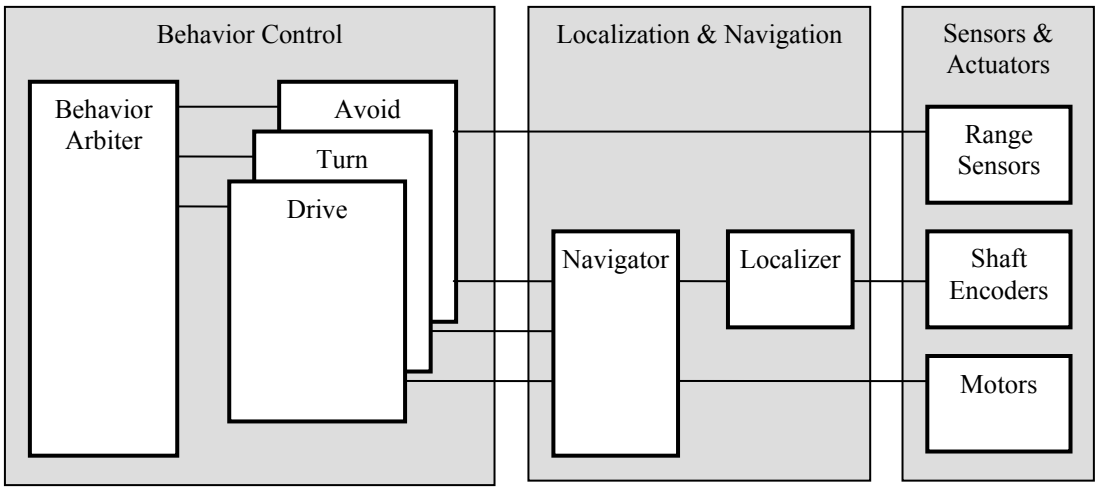


Figure 11-2 – Behavior-based Control Class Diagram

The behavior control objects consist of the three simple behaviors described previously: Avoid, Turn and Drive, as well as the Behavior Arbiter which oversees all of the individual behaviors.

Implementing Behavior Objects

Ideally, your BehaviorArbiter class should not know specifics about each behavior. It should simply implement a generic way of interacting with individual behaviors. By keeping the interface between the arbiter and the behaviors generic, it will be easy to add, remove or change behaviors. The BehaviorArbiter class will not need to change to support such modifications to the overall program.

The key to accomplishing this is defining a Behavior interface that must be implemented by each individual behavior. This will allow the BehaviorArbiter to interface with the behaviors without being exposed to their specific details.

Defining Your Behavior Interface

The Behavior interface will define all of the methods the BehaviorArbiter will use to interface to individual behavior objects. These consist of two methods, one which the arbiter calls periodically for the behavior to perform its function, and a second to tell the behavior whether it should take control of your robot. We will name the former method "poll" and the later method "setActive."

The BehaviorArbiter will periodically call the poll method whether the behavior wants control of your robot, or not. If the behavior has previously been activated, the behavior may also issue control commands to your robot in the poll method. The poll method returns true if the behavior wants control of your robot.

The setActive method activates or deactivates the behavior. If the behavior is inactive, it must not attempt to control your robot. If the behavior is active it should take control of your robot. The BehaviorArbiter decides which behavior should be active at any point in time by activating the highest priority behavior that wants control. This method of behavior arbitration relies on the individual behaviors working cooperatively at the direction of the BehaviorArbiter.

Create a new project and define your Behavior interface using the following procedure.

1. Create a new project named "VacuumCleaner."

2. Create a new class named Behavior and change the keyword class to interface.

```
public interface Behavior {
}
```

3. Define the poll and setActive methods.

   The poll method does not require any arguments. The setActive method requires a boolean argument to indicate whether it is being activated or deactivated.

```
public boolean poll();
public void setActive(boolean isActive);
```

4. Save your files.

Implementing Your Behavior Arbiter

Your BehaviorArbiter class will need to periodically loop through the list of behaviors, calling each behavior's poll method and selecting which behavior will acquire control of your robot. The behaviors need to be prioritized relative to each other, enabling the

BehaviorArbiter to activate the highest priority behavior that wants control at any point in time. This is easily accomplished by organizing the behaviors in a priority ordered array, with the highest priority behavior first in the array and the lowest priority behavior last in the array. Your BehaviorAribiter can then poll each behavior from the beginning to the end of the array, giving control to the first behavior that indicates it wants to control your robot.

As we did in previous exercises, we will use a separate thread to carry out the BehaviorArbiter object's function.

Create your BehaviorArbiter class according to the following procedure.

1. Create a new class with the name BehaviorArbiter and declare that it extends the Thread class.

```
public class BehaviorArbiter extends Thread {
}
```

2. Add fields to refer to the list of behaviors and to hold the period at which the thread will repeat processing.

```
private Behavior[] behaviors;
private int period;
```

3. Add a constructor to with arguments for the list of behaviors and the period at which the BehaviorArbiter thread will run.

```
public BehaviorArbiter(Behavior[] behaviors, int period) {
}
```

4. Initialize the fields in the constructor.

```
this.behaviors = behaviors;
this.period = period;
```

5. Add a run method with a try-catch block to catch and report any exceptions.

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

6. Within the try block, declare and initialize local variables to keep track of the active behavior and the time for the next iteration of the main loop.

```
int activeBehavior = -1;
long nextTime = System.currentTimeMillis() + period;
```

7. Create the main update loop, including statements to make the thread sleep the appropriate amount of time between iterations.

```
while (true) {
    // insert behavior processing code here
    long currentTime = System.currentTimeMillis();
    long sleepTime = nextTime - currentTime;
    if (sleepTime > 0) {
        nextTime += period;
        Thread.sleep(sleepTime);
    }
    else {
        nextTime = currentTime + period;
    }
}
```

8. Create two loops, one to poll behaviors until it finds the first behavior that wants control of your robot, and the second one to poll the remaining behaviors.

```
int i = 0;
while (i < behaviors.length) {
    if (behaviors[i].poll()) {
        // found 1st behavior that wants control
    }
    ++i;
}
++i;

// Poll remaining behaviors
while (i < behaviors.length)
    behaviors[i++].poll();
```

9. In the first loop, add statements to deactivate the behavior that was previously active and activate the behavior that is being given control.

```
if (i != activeBehavior) {
    if (activeBehavior >= 0)
        behaviors[activeBehavior].setActive(false);
    activeBehavior = i;
    behaviors[activeBehavior].setActive(true);
}
break;
```

10. Save your files.

In the next sections we will test your BehaviorArbiter by implementing behaviors, one at a time, and completing your VacuumCleaner controller class.

Implementing Your Drive Behavior

We will begin by implementing the default behavior that will take control of your robot if no other behavior wants control. When this behavior is active, it will power your robot's motors to drive straight ahead in the direction your robot was facing when the behavior was activated. When it is inactive, it will do nothing but indicate it wants to take control. Since it will be the lowest priority behavior it will only get control when other behaviors do not want control. Initially, it will be the only behavior, so it will always be in control of your robot.

Implement your DriveBehavior class using the following procedure.

1. Create a class named DriveBehavior, declaring that it extends the Behavior interface.

```
public class DriveBehavior implements Behavior {
}
```

2. Add import statements for Localizer and Navigator interfaces.

```
import com.ridgesoft.robotics.Localizer;
import com.ridgesoft.robotics.Navigator;
```

3. Add fields to track whether this behavior is active, to hold references to the localizer and navigator, and to hold the current heading.

```
private boolean isActive;
private Localizer localizer;
private Navigator navigator;
private float heading;
```

4. Implement a constructor with arguments for the localizer and navigator. Initialize the fields that need initialization.

```
public DriveBehavior(
        Localizer localizer, Navigator navigator) {
    this.localizer = localizer;
    this.navigator = navigator;
    isActive = false;
    heading = 0.0f;
}
```

5. Implement the poll method.

   If the behavior is active, command the navigator to drive straight ahead. Return true in all cases to indicate this behavior always wants control of your robot.

```
public boolean poll() {
    if (isActive)
```

```
            navigator.go(heading);

        return true;
    }
```

6. Implement the setActive method.

   If this behavior previously was inactive and it is being set to active, record the current heading in the corresponding field. This behavior will instruct your robot to drive in this direction for as long as it retains control.

```
public void setActive(boolean isActive) {
    if (isActive && !this.isActive)
        heading = localizer.getPose().heading;
    this.isActive = isActive;
}
```

7. Save your files.

Implementing Your Controller Class

Now let's implement the details which will tie the pieces of your behavior-based control program together. Your VacuumCleaner class needs to create all of the objects shown in Figure 11-2. You are familiar with creating most of these classes from previous exercises in previous chapters. The only part that is new is creating the behavior objects and the BehaviorArbiter. Initially, we will focus on only the one behavior you have developed so far, the DriveBehavior class. We will add more behaviors in subsequent sections.

Use the following procedure to complete the implementation of your VacuumCleaner class for the first behavior.

1. Add import statements to your VacuumCleaner class for all of the library classes your program will use.

```
import com.ridgesoft.intellibrain.IntelliBrain;
import com.ridgesoft.io.Display;
import com.ridgesoft.robotics.Motor;
import com.ridgesoft.robotics.AnalogInput;
import com.ridgesoft.robotics.ContinuousRotationServo;
import com.ridgesoft.robotics.ShaftEncoder;
import com.ridgesoft.robotics.AnalogShaftEncoder;
import com.ridgesoft.robotics.DirectionListener;
import com.ridgesoft.robotics.Localizer;
import com.ridgesoft.robotics.OdometricLocalizer;
import com.ridgesoft.robotics.Navigator;
import com.ridgesoft.robotics.DifferentialDriveNavigator;
```

2. In the main method, add statements to output the name of your program.

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

```
display.print(1, "");
```

3. Add statements to instantiate objects for range sensing, shaft encoding, localization and navigation.

```
AnalogInput leftRange = IntelliBrain.getAnalogInput(1);
AnalogInput rightRange = IntelliBrain.getAnalogInput(2);

ShaftEncoder leftEncoder =
    new AnalogShaftEncoder(IntelliBrain.getAnalogInput(4),
            250, 750, 30, Thread.MAX_PRIORITY);
ShaftEncoder rightEncoder =
    new AnalogShaftEncoder(IntelliBrain.getAnalogInput(5),
            250, 750, 30, Thread.MAX_PRIORITY);

Localizer localizer =
    new OdometricLocalizer(leftEncoder, rightEncoder,
            2.65f, 4.55f, 16, Thread.MAX_PRIORITY - 1, 30);

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

Navigator navigator =
    new DifferentialDriveNavigator(leftMotor, rightMotor,
            localizer, 8, 6, 25.0f, 0.5f, 0.08f,
            Thread.MAX_PRIORITY - 2, 50);
```

4. Add statements to create an array of behaviors, initializing it with an instance of the one behavior you have created so far, the DriveBehavior.

```
Behavior behaviors[] = new Behavior[] {
    new DriveBehavior(localizer, navigator),
};
```

5. Add statements to create a BehaviorArbiter, passing the array of behaviors and the iteration period as arguments.

```
BehaviorArbiter arbiter =
    new BehaviorArbiter(behaviors, 200);
```

6. Add statements to initialize and start the BehaviorArbiter object's thread, setting the priority of the thread to one priority level below that of the navigator's thread.

```
arbiter.setPriority(Thread.MAX_PRIORITY - 3);
arbiter.start();
```

7. Build, load and test your program.

   Your robot should drive straight ahead.

Implementing Your Turn Behavior

In order to be an effective vacuum cleaner, your robot will need to sweep over the entire area of the floor. One way to accomplish this would be to have your robot maintain a map of the entire space it is assigned to vacuum and keep track of where it has and has not vacuumed. It could then plan its path to vacuum every location it hadn't previously covered. This would be fairly complex to implement, and your robot would either have to be pre-programmed with the map, or it would have to develop a map on its own.

A simpler approach is to program your vacuum cleaner robot to occasionally make a random turn. If the turns are truly random, eventually your robot will go over the entire floor. In this way, your robot will not need to have a map of the area it is intended to vacuum, nor will it need to keep track of which locations it has and has not vacuumed, avoiding the added complexity this would require. We will implement the turn behavior such that the behavior attempts to cause a random turn every ten seconds. We will use the Random class to facilitate choosing a random turn angle between 45 and 315 degrees.

Use the following procedure to add a turn behavior to your program.

1. Create a new class named TurnBehavior declaring the class implements the Behavior and NavigatorListener interfaces.

```
public class TurnBehavior
    implements Behavior, NavigatorListener {
}
```

2. Add import statements for Localizer, Navigator, NavigatorListener and Random classes.

```
import com.ridgesoft.robotics.Localizer;
import com.ridgesoft.robotics.Navigator;
import com.ridgesoft.robotics.NavigatorListener;
import java.util.Random;
```

3. Define the fields the behavior will need.

```
private boolean isActive;
private Localizer localizer;
private Navigator navigator;
private int period;
private long nextTime;
private boolean turning;
private Random random;
```

4. Create a constructor that provides arguments for the localizer and navigator objects, the period between random turns and a seed for the random number generator.

```
public TurnBehavior(
        Localizer localizer, Navigator navigator,
        int periodSeconds, long seed) {
    this.localizer = localizer;
    this.navigator = navigator;
    period = periodSeconds * 1000;
    nextTime = System.currentTimeMillis() + period;
    isActive = false;
    turning = false;
    random = new Random(seed);
}
```

5. Create the poll method.

```
public boolean poll() {
}
```

6. Add statements to the poll method to indicate that the behavior desires to maintain control of your robot while the navigator completes a turn that is in progress.

```
if (turning)
    return true;
```

7. Add statements to the poll method to determine if it is time for another turn and, if so, request control.

```
if(System.currentTimeMillis() > nextTime) {
    // add statements here to take control if active
    return true;
}
return false;
```

8. Insert statements to determine if the behavior is active and, if so, update the turning field and also schedule the next turn.

```
if (isActive) {
    turning = true;
    nextTime = System.currentTimeMillis() + period;
    // add statements here to initiate turn
}
```

9. Within the if statement, add a statement to randomly choose a new heading between 45 and 315 degrees from the current heading.

```
float newHeading =
    localizer.getPose().heading +
```

```
        (float)Math.toRadians(random.nextInt(270) + 45.0f);
```

10. Follow the previous statement with a statement to command the navigator to make the turn and to indicate this object's listener method should be called when the operation completes or is terminated.

```
    navigator.turnTo(newHeading, this);
```

11. Implement the method required by the NavigatorListener interface, navigationOperationTerminated.

    When the navigator terminates the turn operation, update the state of this behavior by setting the turning field to false regardless of whether the operation was completed or interrupted by another command being issued to the navigator.

```
public void navigationOperationTerminated(
        boolean completed) {
    turning = false;
}
```

12. Implement the setActive method.

```
public void setActive(boolean isActive) {
    this.isActive = isActive;
}
```

13. Construct and add a TurnBehavior object to the list of behaviors in your VacuumCleaner class.

    Specify ten seconds as the time between random turns. Seed the random number generator with the product of readings from the range sensors. This will seed the random number generator so your robot will execute a different sequence of turns on each run of your program.

```
Behavior behaviors[] = new Behavior[] {
    new TurnBehavior(
        localizer, navigator, 10,
        (leftRange.sample() * rightRange.sample())),
    new DriveBehavior(localizer, navigator),
};
```

14. Build, load and test your program.

    Your robot should drive straight making a random turn of 45 to 315 degrees from the current heading every ten seconds. Notice if the random turn is greater than 180 degrees the navigator will choose to turn counterclockwise to achieve the new heading.

Implementing Your Avoid Behavior

With the two behaviors we've implemented so far, your robot will wander around the floor in a random pattern, but will still crash into obstacles that are in its way. Our next step is to add a behavior to sense and avoid objects in its path. This new behavior, which we will name AvoidBehavior, will use the range sensors to sense obstacles ahead of your robot. If the AvoidBehavior detects an object, it will indicate it wants control of your robot. If it is given control, it will instruct the navigator to turn and then drive away for a short period of time to facilitate driving around small objects, as shown in Figure 11-3. The ability to navigate around objects will prove useful as you extend your program later in this chapter.

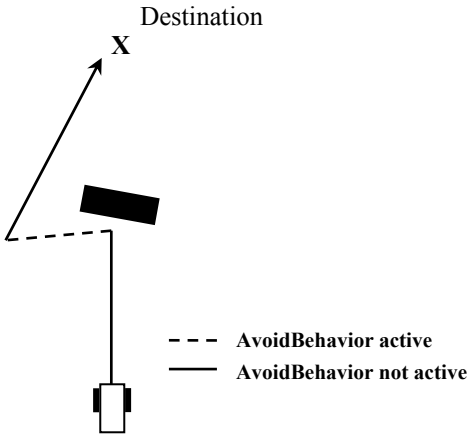


Figure 11-3 - Object Avoidance

Use the following procedure to implement your AvoidBehavior class.

1. Create a new class named "AvoidBehavior."

2. Declare the class implements the Behavior interface.

```
public class AvoidBehavior implements Behavior {
}
```

3. Add import statements for Localizer, Pose, Navigator and AnalogInput classes.

```
import com.ridgesoft.robotics.Localizer;
import com.ridgesoft.robotics.Pose;
import com.ridgesoft.robotics.Navigator;
import com.ridgesoft.robotics.AnalogInput;
```

4. Define fields the class will need.

```
private boolean isActive;
private Localizer localizer;
private Navigator navigator;
private AnalogInput leftRange;
private AnalogInput rightRange;
```

```
private int threshold;
private float turnAmount;
private int holdTime;
private long holdUntil;
private float heading;
```

5. Create a constructor for the class with arguments to provide externally defined objects and parameters.

```
public AvoidBehavior(
        Localizer localizer,
        Navigator navigator,
        AnalogInput leftRange,
        AnalogInput rightRange,
        int threshold,
        float turnAmount,
        int holdTime) {
    this.localizer = localizer;
    this.navigator = navigator;
    this.leftRange = leftRange;
    this.rightRange = rightRange;
    this.threshold = threshold;
    this.turnAmount = turnAmount;
    this.holdTime = holdTime;
    isActive = false;
    holdUntil = 0;
    heading = 0.0f;
}
```

6. Create the poll method.

```
public boolean poll() {
    boolean wantControl = false;
    // add behavior logic here
    return wantControl;
}
```

7. Add statements to sample both range sensors and then determine if there is an object in range, in which case the behavior wants control.

```
int leftValue = leftRange.sample();
int rightValue = rightRange.sample();

if ((leftValue > threshold) || (rightValue > threshold))
    wantControl = true;
```

8. Add statements to control your robot if the behavior is active.

   If the behavior has sensed an object, it will need to instruct the navigator to turn away. If there is no object ahead, but your robot hasn't driven long enough to facilitate going around an object, it will need to instruct the navigator to continue driving away from the object. Otherwise, the behavior should instruct the

navigator to stop until another behavior assumes control.

```
if (isActive) {
    if (wantControl) {
        // object ahead, turn away
    }
    else if (System.currentTimeMillis() < holdUntil) {
        // object out of view, continue driving away
    }
    else
        navigator.stop();
}
```

9. Add statements to the turn-away branch of the code above to instruct the navigator to turn to a new heading.

   The direction of the turn should be chosen based on which sensor detects the object. If the left sensor detected the object, your robot should turn right. If the right sensor detected the object, your robot should turn left.

```
Pose pose = localizer.getPose();
if (leftValue > rightValue)
    heading = pose.heading - turnAmount;
else
    heading = pose.heading + turnAmount;
navigator.go(heading);
holdUntil = System.currentTimeMillis() + holdTime;
```

10. Add statements to the continue-to-drive-away branch above to indicate the behavior wants to retain control and to instruct the navigator to continue driving away from the object.

```
wantControl = true;
navigator.go(heading);
```

11. Implement the setActive method.

```
public void setActive(boolean isActive) {
    this.isActive = isActive;
}
```

12. Add your AvoidBehavior as the highest priority behavior in your VacuumCleaner class.

```
Behavior behaviors[] = new Behavior[] {
    new AvoidBehavior(localizer, navigator,
            leftRange, rightRange,
            200, 0.7f, 3000),
    new TurnBehavior(localizer, navigator, 10,
            (leftRange.sample() * rightRange.sample())),
    new DriveBehavior(localizer, navigator),
```

```
};
```

13. Build, load and test your program.

    Your robot should drive around the floor making a random turn every ten seconds; however, if it encounters a wall or some object in its path, it should turn away from the object to avoid colliding with it.

Implementing Your Return Home Behavior

Some vacuum cleaner robots include a feature to return home to their charging station when their batteries are nearly drained. Let's add another behavior to your program to make your robot return home after it has vacuumed for a short period of time. For the purpose of this exercise, we will consider home to be the spot where your robot was when your program started.

Similar to the Turn behavior, this new behavior will be triggered by time. When it is time to return home, the Return Home behavior will need to override the Turn and Drive behaviors; however, we do not want it to override the Avoid Objects behavior since we don't want your robot to run into anything on its way home. Therefore, the Return Home behavior should be higher priority than the Turn and Drive behaviors, but lower priority than the Avoid Objects behavior, as illustrated in Figure 11-4.

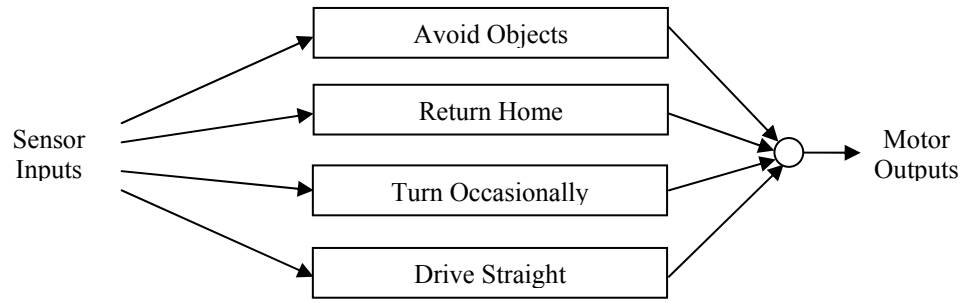


Figure 11-4 – Enhanced Vacuum Cleaner Behaviors

Use the following procedure to implement this new feature.

1. Create a new class named "ReturnHomeBehavior."

2. Declare that the class implements the Behavior and NavigatorListener interfaces.

```
public class ReturnHomeBehavior
        implements Behavior, NavigatorListener {
}
```

3. Add import statements for Navigator and NavigatorListener interfaces.

```
import com.ridgesoft.robotics.Navigator;
import com.ridgesoft.robotics.NavigatorListener;
```

4. Add fields to hold a reference to the navigator, store the return home time and track whether the behavior is active.

```
private Navigator navigator;
private long returnHomeTime;
private boolean isActive;
```

5. Add a constructor that allows the caller to provide a reference to the navigator and also specify how many seconds should pass before the behavior triggers.

```
public ReturnHomeBehavior(
        Navigator navigator, int returnHomeAfterSeconds) {
    this.navigator = navigator;
    returnHomeTime =
        System.currentTimeMillis() +
        returnHomeAfterSeconds * 1000;
    isActive = false;
}
```

6. Add the poll method.

   The method should return false, indicating it does not want control if the trigger time has not been reached. If the behavior is activated, it should command the navigator to return to the starting point.

```
public boolean poll() {
    if (System.currentTimeMillis() < returnHomeTime)
        return false;

    if (isActive)
        navigator.moveTo(0.0f, 0.0f, this);

    return true;
}
```

7. Add the setActive method.

```
public void setActive(boolean isActive) {
    this.isActive = isActive;
}
```

8. Add the navigationOperationTerminated method.

   Once your robot has reached home, your program should exit.

```
public void navigationOperationTerminated(
        boolean completed) {
    if (completed)
        System.exit(0);
```

```
}
```

9. Add the ReturnHomeBehavior to the list of behaviors in your VacuumCleaner class, specifying a trigger time of 30 seconds.

```
Behavior behaviors[] = new Behavior[] {
    new AvoidBehavior(localizer, navigator,
        leftRange, rightRange, 200, 0.7f, 3000),
    new ReturnHomeBehavior(navigator, 30),
    new TurnBehavior(localizer, navigator,
        10, (leftRange.sample() * rightRange.sample())),
    new DriveBehavior(localizer, navigator),
};
```

10. Build, load and test your program.

    Your robot should exhibit the same vacuum cleaner behavior as previously, but after 30 seconds it should turn and head back to the point where it started. It should stop near its starting point, although it will normally not return to the exact point where it started due to accumulated localization error.

Summary

The behavior-based control approach is modeled after animal behavior. Using this approach you break down the overall behavior of your robot into simple behaviors that function independent of each other. By executing all of the behaviors concurrently and giving each behavior a priority relative to all the other behaviors, a much more complex overall behavior emerges.

You observed how combining simple behaviors can be used to create a more complex emergent behavior as you developed your VacuumCleaner program. Initially, you developed a program with just one simple behavior. The emergent behavior was the same as the simple behavior; your robot simply drove forever in the direction it was facing when your program started. Adding a second simple behavior to turn occasionally resulted in a more complex emergent behavior, whereby your robot wandered around the floor in a random pattern. Adding a third simple behavior to avoid obstacles, resulted in your robot wandering without running into things. Finally, adding a fourth behavior resulted in the emergent behavior whereby your robot would wander around for a while then return to where it started.

Exercises

1. What was the basis of inspiration for the behavior-based method of robot control?

2. Describe how behavior-based control works. What is a behavior? How can you combine simple behaviors to form a more sophisticated emergent behavior?

3. Create a behavior that will navigate your robot to a specified point. Use the behavior to create a program that will cause your robot to navigate a triangle, as

illustrated in Figure 11-5.

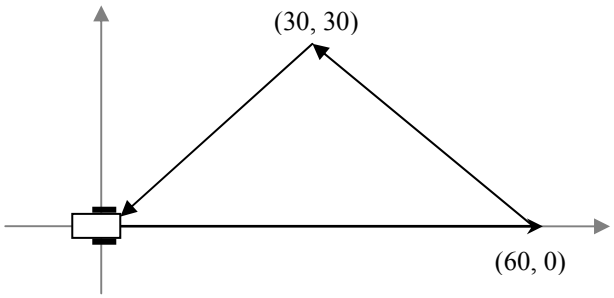


Figure 11-5 – Triangular Path

Hints: Implement the behavior such that it will cease to want control once your robot has reached the specified point. Use three instances of the behavior, arranged in priority, such that your robot will navigate between points in the desired order. Also, create a stop behavior that will terminate your program if no other behavior wants control. Specify the stop behavior as the lowest priority behavior.

4. Extend the previous program using your AvoidBehavior class to enable your robot to navigate around obstacles placed in its path. Place an obstacle along the path, as illustrated in Figure 11-6 to verify your robot will navigate around it as it drives between the specified points.

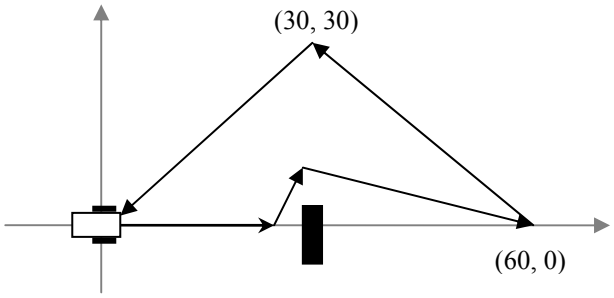


Figure 11-6 – Triangular Path with an Obstacle

5. Extend the previous program to make your robot navigate a square pattern while avoiding obstacles in its path.