



Chapter #6: Light Sensitive Navigation with Photoresistors

Chapter #6: Light Sensitive Navigation with Photoresistors

The photoresistors in your kit can be used to make your J-Bot detect variations in light level. With some programming, your J-Bot can be transformed into a

photophile (a creature attracted to light), or a photophobe (a creature that tries to avoid light).

To sense the presence and intensity of light you'll build a couple of photoresistor circuits on your J-Bot. A photoresistor is a light-dependent resistor (LDR) that covers the spectral sensitivity similar to that of the human eye. The active elements of these photoresistors are made of Cadmium Sulfide (CdS). Light enters into the semiconductor layer applied to a ceramic substrate and produces free charge carriers. A defined electrical resistance is produced that is inversely proportional to the illumination intensity. In other words, darkness produces high resistance, and high illumination produces very small amounts of resistance.

The specific photoresistors included in the J-Bot kit are EG&G Vactec (#VT935G). If you need additional photoresistors they are available from Parallax as well as from many electronic component suppliers. See Appendix A: J-Bot Parts Lists and Sources. The specifications of these photoresistors are shown in Figure 6.1:

Figure 6.1: EG&G
Vactec
Photoresistor
Specifications

| Photoresistor Specifications | | | | | | | | |
|------------------------------|-------|------|------|------|---------------------------|------------------|---------------------------------|------------|
| Resistance (Ohms) | | | | | Peak Spectral Response nm | V _{MAX} | Response Time @ 1 fc (ms, typ.) | |
| 10 Lux 2850K | | | Dark | | | | Rise (1-1/e) | Fall (1/e) |
| Min | Typ. | Max. | Min. | Sec. | | | | |
| 20K | 29.0K | 38K | 1M | 10 | 550 | 100 | 35 | 5 |

Illuminance is a scientific name for the measurement of incident light. The unit of measurement of illuminance is commonly the "foot-candle" in the English system and the "lux" in the metric system. While using the photoresistors we won't be concerned about lux levels, just whether or not illuminance is higher or lower in certain directions. The J-Bot can be programmed to use the relative light intensity information to make navigation decisions. For more information about light measurement with a microcontroller, take a look at Earth Measurements Experiment #4, Light on Earth and Data Logging.

Activity #1: Building and Testing Photosensitive Eyes

Parts

Figure 6.2 shows the new parts introduced in this experiment along with their schematic symbols. Below is a list of the parts you'll need. Both parts types of parts are nonpolar, meaning that terminals 1 and 2 as shown may be swapped without affecting the circuit.

- (1) Piezoelectric speaker
- (2) Photoresistors
- (2) 0.1 μF capacitors
- (2) 0.01 μF capacitors
- (2) 220 Ω resistors
- (misc.) jumper wires

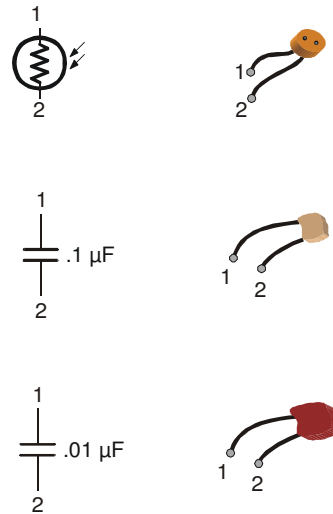


Figure 6.2: Photoresistor and capacitor circuit symbols and parts.

Build It!

Figure 6.3 shows (a) the resistor/capacitor (RC) circuit for each photoresistor and (b) a breadboard example of the circuit. A photoresistor is an analog device. Its value varies continuously as illuminance, another analog value, varies. The photoresistor's resistance is very low when it's light-sensitive surface is placed in direct sunlight. As the light level decreases, the photoresistor's resistance increases. In complete darkness, the photoresistor's value can increase to more than 1 $\text{M}\Omega$. Even though the photoresistor is analog, its response to light is nonlinear. This means if the input source (illuminance) varies at a constant rate, the photoresistor's value does not necessarily vary at a constant rate.

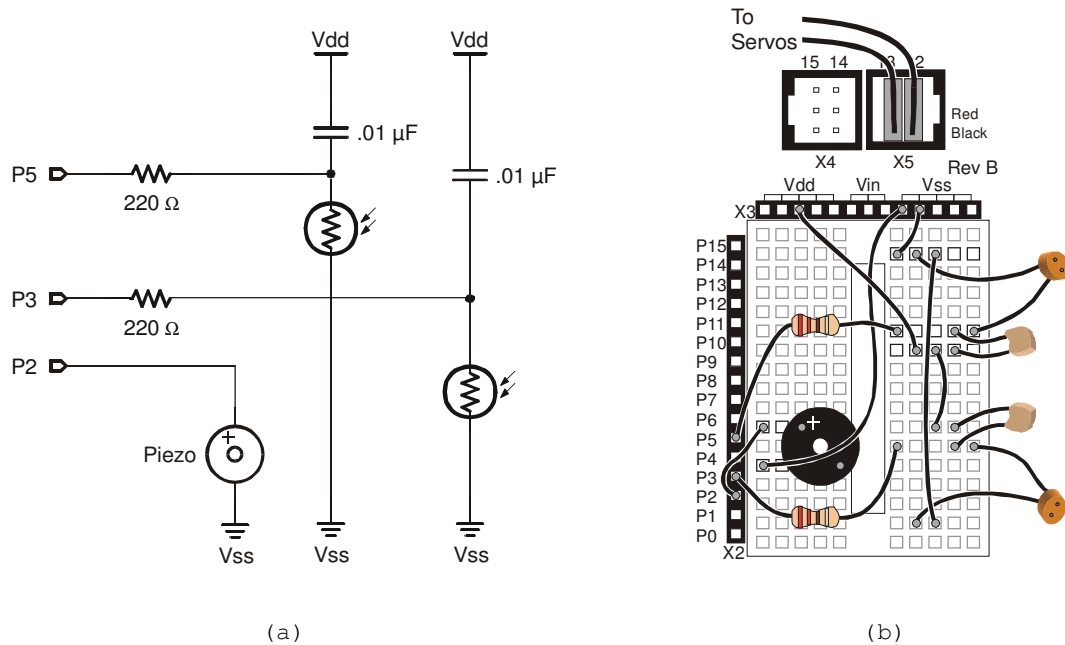


Figure 6.3: (a) Two photoresistor RC circuits for measurement of resistance that varies with light, and (b) breadboard example of the circuit.

TIP

Remember: The servo circuits are not shown in the schematics any more, but they are still shown in the breadboard diagrams. All activities from Chapter #2 onward are designed so that the servos' headers can remain plugged into servo ports 12 and 13 at all times.

Programming to Measure the Resistance

The circuit in Figure 6.3 (a) was designed for use with the **JAVA CPU.rcTime** method. This command can be used with an RC circuit where one value, either R or C, varies while the other remains constant. The **CPU.rcTime** method lends itself to measuring the variable values because it takes advantage of a time varying property of RC circuits. The time it takes for the voltage on an RC circuit to change voltage depends on $R \times C$, the RC time constant. The RC time constant is often denoted by the Greek letter Tau (τ).

For one of the RC circuits shown in Figure 6.3 (a), the first step in setting up the **CPU.rcTime** method measurement is charging the lower plate of the capacitor to 5 V. Setting the I/O pin connected to the lower capacitor plate by the 220 Ω resistor high for a few ms takes care of this. Next, the **CPU.rcTime** method can be used to take the measurement of the time it takes the lower plate to discharge from 5 to 1.4 V. Why 1.4 V? Because that's

the Javelin I/O pin's threshold voltage. When the voltage at an I/O pin set to input is above 1.4 V, the value in the input register bit connected to that I/O pin is "1." When the voltage is below 1.4 V, the value in the input register bit is "0."

The **CPU.rcTime** method for the circuit shown in Figure 6.3 measures how long it takes for the voltage at the lower plate of the capacitor to fall from 5 to 1.4 V. This time varies according to the formula:

$$\frac{t}{R \times C} = \ln\left(\frac{V_{\text{initial}}}{V_{\text{final}}}\right)$$

$$\frac{t}{R \times 0.01 \times 10^{-6}} = \ln\left(\frac{5 \text{ V}}{1.4 \text{ V}}\right) \text{ s}$$

$$t = \ln(3.57) \times R \times 0.01 \times 10^{-6} \text{ s}$$

$$t = 1.27 \times 10^{-8} \times R \text{ s} \quad (4.1)$$

Equation 4.1 indicates that the time it takes the voltage at the lower plate of the capacitor in one of the Figure 4.3 (a) RC circuits to drop from 5 to 1.4 V is directly proportional to the photoresistor's resistance. Since this resistance varies with illuminance (exposure to varying levels of light), so does the time. By measuring this time, relative light exposure can be inferred.

The **CPU.rcTime** method changes the I/O pin from output to input. As soon as the I/O pin becomes an input, the voltage at the lower plate of the capacitor starts to fall according to the time equation just discussed. The Javelin starts counting in 8.68 μ s increments until the voltage at the capacitor's lower plate drops below 1.4 V.

TIP For Best Results: Eliminate direct sunlight; it's too bright for the photoresistor circuits.

- ❑ Run `Photoresistor1.java`. It demonstrates how to use the **CPU.rcTime** method to read the photoresistors. It uses the `Photoresistor` class defined first.
- ❑ This program makes use of the Message window and the debugger, so leave the serial cable connected to the JSDB while `Photoresistor1.java` is running.

```
package JBot ;
```

```

import stamp.core.*;

/**
 * Basic Photoresistor Class
 * <p>
 * Tests the photoresistor circuits using CPU.rcTime.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class Photoresistor {
    public int pin ;
    public int timeout ;
    public int chargeTime ;
    public int bias ;
    public boolean state ;

    /**
     * Gets RC time value in 8.68us units
     *
     * @param pin CPU.pin to use
     * @param state initial RC state
     * @param timeout maximum rcTime return value
     * @param chargeTime msec to charge/discharge RC circuit
     */
    public Photoresistor ( int pin, boolean state, int timeout, int chargeTime,
int bias ) {
        this.pin = pin ;
        this.timeout = timeout ;
        this.chargeTime = chargeTime ;
        this.state = state ;
        this.bias = bias ;
    }

    /**
     * Gets RC time value in 8.68us units
     *
     * @returns RC time
     */
    public int rcTime() {
        // Measure RC time for photoresistor.
        CPU.setOutput ( pin ) ;
        CPU.writePin ( pin, state ) ;    // setup to charge circuit
        CPU.delay ( chargeTime * 10 ) ;  // charge circuit
        int result = CPU.rcTime ( timeout, pin, ! state ) ;
        return ( result > 0 ) ? ( result - bias ) : bias ;
    }
}

```

The Photoresistor class handles most of the work. Each object handles one pin. The constructor saves the details such as the pin number. The work is done by the rcTime method. Normally the RC circuit is the same so setting the parameters once is sufficient. The rcTime method initially charges the capacitor circuit and then uses the CPU.rcTime method to determine how quickly the circuit recovers.

The bias value will be 0 in the first example, Photoresistor1. The value may change for the second example, Photoresistor2. The reason for the difference is that the components are may not be identical. For example, the resistors may be rated at 220 ohms but this value is actually the desired value. The actual value can be with 5% to 20% depending upon the part used. The same is true for the other components like the photoresistors and capacitors. The bias value will allow the program to take these differences into account. This is similar to the calibration of the servos done in earlier chapters.

The Photoresistor1 class file simply uses the Photoresistor class for each input pin.

```
import stamp.core.*;
import JBot.* ;

/**
 * Basic Photoresistor Test Program
 * <p>
 * Tests the photoresistor circuits using CPU.rcTime.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class Photoresistor1 {
    public static void main() {
        Photoresistor leftPhoto = new Photoresistor ( CPU.pin5, true, 250, 2, 0 ) ;
        Photoresistor rightPhoto = new Photoresistor ( CPU.pin3, true, 250, 2, 0 ) ;

        while ( true ) {
            // Measure RC time for right photoresistor.
            // Display RC time measurements

            System.out.print ( "L " ) ;
            System.out.print (leftPhoto.rcTime() ) ;
            System.out.print ( " R " ) ;
            System.out.println (rightPhoto.rcTime() ) ;
        }
    }
}
```

How The Photoresistor Display Works

The Photoresistor1 program creates a pair of Photoresistor objects. It then repeatedly prints out the results of the rcTime method calls.

FYI A result of -1 indicates that the result is out of range. The timeout value (250) can be increased in which case the out of range value may show up less often although the default value used in the program

should be sufficient for the experiments and hardware used in this chapter.

One thing you may notice about the results displayed by the program is that the values may be quite different even if the photoresistors are aimed in the same direction. A variable resistor or capacitor could be used to adjust the values for each sensor so they are the same for similar light conditions but this tends to be expensive and hard to do. What we do instead is use the bias value in the Photoresistor class.

The differences between the left and right values tend to be off by the same relative amount. Keep in mind that there will always be some difference because making lighting conditions identical is actually very difficult. Still, it should be relatively apparent what the difference values are.

Your Turn

- ❑ Try determining the bias value by viewing the results printed by the Photoresistor1 program. The bias value is the absolute difference between the two sensors under the same lighting conditions. The bias value should be applied to the sensor that has the higher value.
- ❑ Modify the Photoresistor1 program so the bias value is printed at the end of the line showing each sensor result. Hint: Store the rcTime results in integer variables left and right.

Photoresistor Bias

The Photoresistor2 program, shown below, makes minor changes to the Photoresistor1 program.

```
import stamp.core.*;
import JBot.* ;

/**
 * Basic Photoresistor Test Program
 * <p>
 * Tests the photoresistor circuits using CPU.rcTime.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class Photoresistor2 {
    public static void main() {
        Photoresistor leftPhoto = new Photoresistor ( CPU.pin5, true, 250, 2, 0 ) ;
        Photoresistor rightPhoto = new Photoresistor ( CPU.pin3, true, 250, 2, 30 )
    ;
}
```

```
while ( true ) {  
    // Measure RC time for right photoresistor.  
    // Display RC time measurements  
  
    System.out.print ( "L " ) ;  
    System.out.print ((leftPhoto.rcTime()+5)/10) ;  
    System.out.print ( " R " ) ;  
    System.out.println ((rightPhoto.rcTime()+5)/10) ;  
}  
}
```

The first change is to the bias value. In this case the bias value is 30 and applied to the right sensor. During the tests the right sensor generated a value of 61 while the left sensor generated a 31 under the same conditions.

The second change occurs where the rcTime results are printed. The equation used divides the results by 10. Adding 5 rounds the result up. For example, a result of 15 prints as 2 while 12 prints as 1.

FYI The Photoresistor2 program rounding does not take in account the out of range value of -1.

The reason for reducing the magnitude of the rcTime result is that the accuracy of hardware is limited. Dividing the result by 10 reduces the number of significant digits by 1. One thing that you will notice is that the change in results occur less often as small changes are made to the lighting conditions. This can be done by moving the light source or putting an obstacle in the way to cast a shadow over the sensors.

Where the original program would print out a range of values for a particular lighting condition, such as 157 to 161, the new program would consistently print a value of 16. Differences for minor changes in lighting conditions would be a difference of 1 or 2 instead of a difference of 1 to 24.

Reducing the sensor variance for minor changes in lighting conditions is critical. Otherwise the J-Bot will react too often to minor changes. Reducing the variance within the sensor code will isolate such differences from the main program that utilizes the sensors.

Your Turn

- ❑ Fix Photoresistor2 so it retains the out of range value (-1). Hint: Create a method that performs the rounding so the code does not have to be replicated for each sensor.
- ❑ Determine an upper threshold value that the sensors return when aimed at a bright light. This value should be low enough that it will always be below either sensor result. For example, if the left result varies from 21 to 24 and the right result is 19 to 22 then the threshold should be 18 or 19.
- ❑ Try replacing one of the 0.01 μF capacitors with a 0.1 μF capacitor. Which circuit fares better in bright light, the one with the larger (0.1 μF) or the one with the smaller (0.01 μF) capacitor? What is the effect as the surroundings get darker and darker? Do you notice any symptoms that would indicate that one or the other capacitor would work better in a darker environment? Did you have to change the charge or timeout values?
- ❑ Make sure to restore your circuit to its original state before moving on to the next activity.

Activity #2: Sensor Class - Photoresistors

The problem with Photoresistor class is that utilizes CPU.delay. Although it uses a small delay value it can impact other tasks in a multitasking system. Unfortunately, using the rcTime method will also impact other tasks but its delay is less than the charge time. It would be nice to have background virtual peripheral support but that is not part of the Javelin's repertoire.

The PhotoresistorSensor needs a multitasking component, PhotoresistorSensorTask, to reduce its overhead. The two work in concert allowing another task to poll each photoresistor sensor. The sensor object works in a slightly different fashion than the whisker sensor because the photoresistor does not determine the range to an object or a light source but rather the intensity.

For our purposes, we assume that the light intensity will be used to simulate an obstacle. A dark area will be closer to an obstacle while a bright area will be considered an open area. Subsequent example programs will try to move the J-Bot toward the light which would be the same as moving away from an obstacle. Therefore a bright area should not indicate an obstacle. We will have to come up with a threshold value which will be done using the test program that utilizes the PhotoresistorSensor object. In essence, the high values from the rcTime method will have to be inverted since a low value would indicate a close obstacle versus a brighter/higher rcTime value that indicates no obstacle or one that is farther away.

We start our class definitions with the PhotoresistorSensor class based on the BaseSensor class already presented and used.

```
package JBot ;

import stamp.core.*;
import java.lang.Math.* ;

/**
 * Photoresistor Sensor Class
 * <p>
 * Tests the photoresistor sensors using PhotoresistorTask.
 * Indicates there is no obstacle if light intensity is below a threshold.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class PhotoresistorSensor extends BaseSensor {
    protected PhotoresistorSensorTask sensorTask ;

    protected int direction ;
    protected int distance ;
    protected boolean obstacleDetected = false ;
    protected int lowerLimit ;
    protected int deadband ;

    /**
     * Create photoresistor sensor object and support task
     *
     * @param leftPin CPU.pin to use
     * @param rightPin CPU.pin to use
     * @param state initial RC state
     * @param timeout maximum rcTime return value
     * @param chargeTime msec to charge/discharge RC circuit
     * @param bias offset, subtract from left side >0, right <0
     * @param lowerLimit lowest distance value for no obstacle
     */
    public PhotoresistorSensor ( int leftPin
                                , int rightPin
                                , boolean state
                                , int timeout
                                , int chargeTime
                                , int bias
                                , int lowerLimit
                                , int deadband ) {
        sensorTask = new PhotoresistorSensorTask
            ( this
            , leftPin
            , rightPin
            , state
            , timeout
            , chargeTime
            , bias ) ;
        this.lowerLimit = lowerLimit ;
        this.deadband = deadband ;
    }

    /**
     * Indicate whether an obstacle has been detected.
```

```

    * Normally used when polling versus using an event.
    *
    * @returns obstacle detected
    */
public boolean obstacleDetected () {
    sensorTask.checkSensors() ;

    return obstacleDetected ;
}

/**
 * Indicate initial obstacle position.
 * For simple detection systems the detection of an object
 * on the right and left will return front.
 *
 * @returns obstacle's relative direction (left, right, etc.)
 */
public int obstacleDirection () {
    return direction ;
}

/**
 * Get the distance to an obstacle in the specified direction.
 * A value of <code>none</code> indicates no object detected.
 *
 * @param direction to get range for
 *
 * @returns distance to an obstacle for the specified direction
 */
public int obstacleDistance ( int direction ) {
    return distance ;
}

/**
 * Update results based on sensor information.
 * Called by sensor task when results available.
 *
 * @param resultLeft photoresistor rcTime result
 * @param resultRight photoresistor rcTime result
 */
protected void saveResults ( int resultLeft, int resultRight ) {
    // Current results are valid
    // Save obstacle status
    switch ( (( resultLeft > lowerLimit ) ? 1 : 0 )
            + (( resultRight > lowerLimit ) ? 2 : 0 ) ) {
    default:
    case 0:
        obstacleDetected = false ;
        break;

    case 1:
        direction = left ;
        distance = resultLeft ;
        obstacleDetected = true ;
        break;

    case 2:
        direction = right ;
        distance = resultRight ;
        obstacleDetected = true ;
        break;

    case 3:

```

```

        // Both sensors indicate an obstacle
        if ( Math.abs ( resultLeft - resultRight ) < deadband ) {
            // Both distances are close together
            direction = front ;
            distance = ( resultLeft > resultRight ) ? resultLeft : resultRight ;
        } else if ( resultLeft > resultRight ) {
            // Left sensor has a higher value
            direction = left ;
            distance = resultLeft ;
        } else {
            // Right sensor has a higher value
            direction = right ;
            distance = resultRight ;
        }
        obstacleDetected = true ;
        break;
    }
}
}

```

This class creates an object that maintains details about the last obstacle reading. It creates a PhotoresistorSensorTask using the parameters passed to the constructor so the object that creates the PhotoresistorSensor object only needs to deal with one object. The task remains hidden.

The BaseSensor abstract methods are defined here and simply return the last obstacle readings. The obstacleDetected method also calls the task's checkSensors method. From the sensor object's standpoint, the method does something but what it does is irrelevant. In reality the method makes sure the task is working to generate new obstacle information.

The saveResults method is used by the task to set the latest values. The results are already adjusted for bias and rounding so the method only needs to contend with out-of-range results and the brightness threshold.

The PhotoresistorSensorTask class definition follows.

```

package JBot ;

import stamp.core.*;
import stamp.util.os.* ;

/**
 * Photoresistor Sensor Class
 * <p>
 * Supports PhotoresistorSensor.
 * Should not be called directly by another object.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */
public class PhotoresistorSensorTask extends Task {

```

```

protected PhotoresistorSensor sensor ;

protected int leftPin ;
protected int rightPin ;
protected int timeout ;
protected int chargeTime ;
protected boolean pinState ;
protected int resultLeft ;
protected int bias ;

final static int startChecking = 1 ;
final static int startLeftPin = 2 ;
final static int startRightPin = 3 ;

protected PhotoresistorSensorTask
    ( PhotoresistorSensor sensor
      , int leftPin
      , int rightPin
      , boolean pinState
      , int timeout
      , int chargeTime
      , int bias ) {
    this.sensor = sensor ;
    this.leftPin = leftPin ;
    this.rightPin = rightPin ;
    this.timeout = timeout ;
    this.chargeTime = chargeTime ;
    this.pinState = pinState ;
    this.bias = bias ;
}

/**
 * Check sensors if not already doing so
 */
protected void checkSensors() {
    if ( state == stopped ) {
        // Task was done. Start it again.
        nextState (startChecking) ;
        start () ;
    }
}

/**
 * Check photoresistor sensor.
 * Round and adjust for bias.
 * Handle out of range condition.
 *
 * @param pin pin to check
 * @param bias bias value to be subtracted if positive
 *
 * @result adjusted rcTime result (-1 is out of range)
 */
protected int rcTime ( int pin, int bias ) {
    int result = CPU.rcTime ( timeout, pin, ! pinState ) ;

    // Handle out of range condition. Time exceeds timeout.
    if ( result == -1 ) {
        result = timeout*2 ;    // use big number
    } else if ( bias > 0 ) {
        // Adjust bias if value is positive
        result -= bias ;
    }
}

```

```

        return (result+5)/10 ;
    }

    /**
     * Multitasking support
     */
    public void execute () {
        switch ( state ) {
            case startChecking:
                // Setup to charge circuit
                CPU.setOutput ( leftPin ) ;
                CPU.setOutput ( rightPin ) ;
                CPU.writePin ( leftPin, pinState ) ;
                CPU.writePin ( rightPin, pinState ) ;

                // Setup delay for charging circuit
                sleep(chargeTime,startLeftPin);
                break;

            case startLeftPin:
                // Measure RC time for photoresistor.
                resultLeft = rcTime ( leftPin, -bias ) ;
                nextState(startRightPin);
                break;

            case startRightPin:
                // Post results
                sensor.saveResults ( resultLeft, rcTime ( rightPin, bias ) ) ;
                // Falls through for stop()
            default:
            case initialState:
                stop() ;
                break;
        }
    }
}

```

The constructor saves off the parameters including the matching PhotoresistorSensor object. The sensor object also maintains a reference to the task so the two can interact with each other.

The task has three methods: checkSensors, execute and rcTime. The checkSensors method is called periodically by the matching PhotoresistorSensor object when the sensor object is polled using the obstacleDetected method. The checkSensors method restarts the task in the startChecking state.

The execute task is where the task runs. The initialState stops the task. It is restarted in the startChecking state by the checkSensors method. This state sets the pins used with the photoresistor circuits so they charge the capacitor. There is a delay while the charging occurs but instead of bring the Javelin to a halt, the multitasking system allows other tasks to run. The charging may run longer than necessary but that does not really matter since the capacitor cannot be overcharged.

The startLeftPin state is executed when the sleep timeout occurs. The rcTime method is called to check the left sensor pin. The task then yields control and will execute the startRightPin state after any other tasks have had a chance to run. At this point the right sensor pin is monitored and the results from both sensors are given to the sensor object by calling the saveResults method. The task then stops. Note that the switch case “falls through” to the initialState case where the stop method is actually called.

The task object is periodically started by the sensor object but the task only runs long enough to generate one set of results. This allows the J-Bot program to perform major actions in response to sensor changes without the overhead to check the photoresistor sensors if the sensor object is not used by these actions.

There is a significant amount of complexity using the two object, multitasking approach. Still, the overhead is on par with the Photoresistor class that uses CPU.delay except that the PhotoresistorSensor works nicely in a multitasking environment. It also works well with the BaseSensor interface that can be used with various multitasking J-Bot applications presented in this book.

Two parameters to the PhotoresistorSensor constructor control the sensitivity of the system. These are the lowerLimit and the deadband parameters. The lowerLimit sets the boundary between no obstacle and an obstacle being detected. The deadband controls the sensitivity between a left and right obstacle indication or a front obstacle indication. The higher the value, the more likely an obstacle detection will be indicated in the front instead of one of the sides.

To test these two classes we use the PhotoresistorSensor1 class file presented next.

```
import stamp.core.*;
import stamp.util.os.* ;
import JBot.* ;

/**
 * Basic Photoresistor Test Program
 * <p>
 * Tests the photoresistor circuits using CPU.rcTime.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class PhotoresistorSensor1 extends Task {
    PhotoresistorSensor sensor =
        new PhotoresistorSensor ( CPU.pin5      // left pin
                                , CPU.pin3      // right pin
                                , true          // pin state
                                , 250           // rcTime limit
        )
}
```

```

        , 2           // timeout
        , 30          // bias
        , 3           // lower limit
        , 5 ) ;       // deadband

public void execute() {
    // Only uses one state
    if ( sensor.obstacleDetected () ) {
        System.out.print ( "Dir=" ) ;
        System.out.print ( sensor.obstacleDirection() ) ;
        System.out.print ( " Dist=" ) ;
        System.out.println ( sensor.obstacleDistance(sensor.obstacleDirection()) ) ;
    } else {
        System.out.println ( "." ) ;
    }
}

public static void main() {
    new PhotoresistorSensor1 () ;

    Task.TaskManager () ;
}
}

```

As with prior multitasking applications, this class definition combines a Task class and a main method that runs the Task.TaskManager. The execute method for the task is very simple. It checks to see if an obstacle is detected and prints out the current status. The PhotresistorSensor1 task object has one object variable, sensor.

Your Turn

- ❑ You will need an area that is well lit and where you can cast a shadow over the photoresistors. This is necessary to test the program properly. Start the program and watch how the display information changes as you cast a shadow with your hand over one or both of the sensors.
- ❑ Make a record of the direction and approximate distance results as you adjust the amount of light falling on each sensor. The distance value will be higher when the amount of light falling on the sensor is low.
- ❑ Make sure the left and right sensors are setup properly. If the left sensor is in the shadows and the left is not then the obstacle direction will be 45 degrees. If both sensors are in the shadows then the angle will be 90 and if the right sensor is in the shadows and the left is not then the angle is 135. No other values will be presented since the sensor object only returns these fixed values.

Activity #3: A Light Compass

If you focus a flashlight beam in front of the J-Bot, the circuit and programming techniques just discussed can be used to make the J-Bot turn so that it's pointing at the flashlight beam. Make sure the photoresistors are pointed so that they can make a light comparison. Aside from each being pointed 45° outward from the center-line of the J-Bot, they also should be oriented so they are pointing 45° downward from horizontal. In other words, point the faces of the photoresistors down toward the table top. Then, use a bright flashlight to make the J-Bot track the direction of the light.

Programming the J-Bot to Point at the Light

Getting the J-Bot to track a light source is a matter of programming it to compare the value measured at each photoresistor. Remember that as the light gets dimmer, the photoresistor's value increases. So, if the photoresistor value on the right is larger than that of the photoresistor on the left, it means it's brighter on the left. This comparison is already done by the PhotoresistorSensor object. It will indicate an obstacle direction of 45 degrees if there is more light on the right side than the left. Likewise, there is more light on the left side if the obstacle direction is 135 degrees. The object also has a deadband when it reports an obstacle directly in front, or 90 degrees. In this case both sensors must detect light but they do not have to be identical.

- ❑ Enter and run PhotoCompass1 program.
- ❑ Shine a bright flashlight in front of the J-Bot. When you move the flashlight, the J-Bot should rotate so that it's pointing at the flashlight beam.
- ❑ Instead of using a flashlight, use your hand to cast a shadow over one of the photoresistors. The J-Bot should rotate away from the shadow.

```
import stamp.core.*;
import stamp.util.os.* ;
import JBot.* ;

/**
 * Basic Photoresistor Compass Test Program
 * <p>
 * Tests the photoresistor circuits using CPU.rcTime.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class PhotoCompass1 extends Task {
    protected JBotInterface jbot = new BasicJBot ( new MultitaskingJBot() ) ;
    protected PhotoresistorSensor sensor =
```

```

        new PhotoresistorSensor ( CPU.pin5      // left pin
                                , CPU.pin3      // right pin
                                , true          // pin state
                                , 250           // rcTime limit
                                , 2             // timeout
                                , 30            // bias
                                , 3             // lower limit
                                , 5 ) ;         // deadband

    public void execute() {
        // Only uses one state
        if ( sensor.obstacleDetected () ) {
            if ( sensor.obstacleDirection() <= 45 ) {
                // Obstacle to the left, turn right
                jbot.pivot(jbot.continuousRight) ;
            } else if ( sensor.obstacleDirection() >= 135 ) {
                // Obstacle to the right, turn left
                jbot.pivot(jbot.continuousLeft) ;
            } else {
                // Directly in front
                jbot.stop() ;
            }
        } else {
            // No obstacle located
            jbot.stop() ;
        }
    }

    public static void main() {
        new PhotoCompass1 () ;

        Task.TaskManager () ;
    }
}

```

How the Light Compass Works

The PhotoCompass1 program utilizes the multitasking system but not the AvoidObstacleTask used earlier. This is because the AvoidObstacleTask uses fixed movements while the PhotoCompass1 task uses fine grain movements that are not part of an integral rotation. The PhotoCompass1 task still uses the PhotoresistorSensor object but adds the BasicJBot object for motor control.

The execute method handles the interface between the PhotoresistorSensor and the BasicJBot objects. The execute method does not use the state variable since it remains in one state forever so the initialState works just fine.

The execute method checks to see if an obstacleDetected returns true. It then checks to see if the obstacle is located in front, to the left or to the right. In this case the obstacle is where it is darker. If the obstacle is not in front then the J-Bot pivots in the opposite direction. It stops when no light is detected or the light is directly in front.

The `pivotLeft` and `pivotRight` methods start the J-Bot turning in the respective direction. It does not stop until it detects no light or the light is directly in front so usually the J-Bot does not pivot indefinitely. Note that calling either method will not cause a pulse to be generated. Instead the underlying PWM object will be generating the pulses based on the last setting. Resetting the values only causes the values to be used when the next pulse will be generated. All this is hidden by the `BasicJBot` class.

Your Turn

In a darker area, not only will the photoresistor values be larger, so will the difference between them. The sensitivity of the `PhotoresistorSensor` object is based on the `lowerLimit` parameter. It is possible to raise and lower this value.

The `lowerLimit` value is currently set to "3" in `PhotoresistorSensor` constructor.

- ❑ Experiment with different ambient light levels and their effect on `lowerLimit` by trying this experiment in lighter and darker areas. In lighter areas, the `lowerLimit` value can be made smaller, even zero. In darker areas, the `lowerLimit` value should be increased.
- ❑ Swap the `pivotLeft` and `pivotRight` method calls in the `PhotoCompass1` program. Then re-run the program. Now your J-Bot points away from the light.

Activity #4: Follow the Light!

Simply by adding some forward motion to your J-Bot, you can turn it into a light-seeking robot, a photophile. An interesting experiment to try is to program the J-Bot to move forward and seek out light. Then, take it into a dark room with the door open to a brighter room. Assuming there are no obstacles in its way, the J-Bot will make its way to the door and exit the dark room.

Programming for Light Following

Programming the J-Bot to follow light requires that only a few modifications to the PhotoCompass1 class be made. The two changes occur where the jbot object was stopped. Let's see how it works.

```
import stamp.core.*;
import stamp.util.os.* ;
import JBot.* ;

/**
 * Basic Photoresistor Compass Test Program
 * <p>
 * Tests the photoresistor circuits using CPU.rcTime.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class PhotoCompass2 extends Task {
    protected JBotInterface jbot = new BasicJBot ( new MultitaskingJBot () ) ;
    protected PhotoresistorSensor sensor =
        new PhotoresistorSensor ( CPU.pin5      // left pin
                                , CPU.pin3      // right pin
                                , true           // pin state
                                , 250           // rcTime limit
                                , 2             // timeout
                                , 30            // bias
                                , 3             // lower limit
                                , 5 ) ;        // deadband

    public void execute() {
        // Only uses one state
        if ( sensor.obstacleDetected () ) {
            if ( sensor.obstacleDirection() <= 45 ) {
                // Obstacle to the left, turn right
                jbot.pivot(jbot.continuousRight) ;
            } else if ( sensor.obstacleDirection() >= 135 ) {
                // Obstacle to the right, turn left
                jbot.pivot(jbot.continuousLeft) ;
            } else if ( sensor.obstacleDistance ( 90 ) > 20 ) {
                // Too dark. Just stop and wait
                jbot.stop() ;
            } else {
                // Directly in front
                jbot.move(jbot.continuousForward) ;
            }
        } else {
            // No obstacle located
            jbot.move(jbot.continuousForward) ;
        }
    }

    public static void main() {
        new PhotoCompass2 () ;

        Task.TaskManager () ;
    }
}
```

How the Light Follower Program Works

First the easy one. The `jbot.stop()` method for the else clause of the initial `obstacleDetected` call is replaced by a `jbot.forward()` call. This allows the J-Bot to move forward when there is a bright light in front of it.

The second change provides a bit more control. In this case, the `sensor.obstacleDistance` method is called when the `obstacleDirection` is 90 (actually not under 45 or over 135 degrees). The if statement that checks the distance either stops the J-Bot or has it move forward. The distance threshold allows the J-Bot to stop when it reaches an area that is too dark. The condition can be eliminated and only the `jbot.forward` call used if the J-Bot should not stop moving. Changing the distance threshold controls when the J-Bot will stop.

FYI The PhotoCompass programs can be tested with the J-Bot tethered to the PC if it is raised so the wheels do not touch the ground.

Your Turn

- ❑ Repeat the previous Your Turn exercise. You can now lead your J-Bot around with a flashlight.
- ❑ Instead of pointing the photoresistors at the surface directly in front of the J-Bot, point them upward and outward as shown in Figure 4.3 on Page xxx. With the photoresistors adjusted this way, the J-Bot will roam on the floor and try to always find the brightest place.

Activity #5: Line Following

If the J-Bot can be programmed to follow a flashlight beam focused in front of it, why can't it follow a white stripe on a black background? The answer is, there's no good reason. The J-Bot can follow a white stripe on a black background, and it's a project in this chapter's Projects section. By the same token, the J-Bot should be able to follow a black stripe on a white background. Regardless of the color of the stripe, this activity is generically referred to as "line following."

The recommended width for the black stripe is about 5 cm. Either construction paper or electrical tape works fine. With some calibration along with controlled lighting conditions, the J-Bot is a very faithful stripe follower.

- ❑ Shadows and bright lights can be misleading, so try to keep the lighting as uniform as possible. For example, overhead fluorescent lights with no light from windows will work well.
- ❑ Also, make sure to bend the photoresistors as far over the front of the J-Bot as possible. In other words, readjust the photoresistors from flashlight beam following.

Programming for Line Following

By changing various parameters from the previous example program, the J-Bot can now follow bold, black stripes on a white background. The LineFollower1 program demonstrates this. The lowerLimit and deadband parameters for the PhotoresistorSensor constructor were reduced. When the difference is larger, the deadband will have to be increased for better performance. In some instances, the sensitivity reduction done within the PhotoresistorSensor class may have to be increased so the lowerLimit and deadband parameters provide more control.

TIP

In brightly lit rooms, decreasing the deadband value may not be enough. The 0.1 μF capacitors can be substituted for the 0.01 μF capacitors in the J-Bot's RC circuits. This will increase the RC times by a factor of 10. Keep this in mind when adjusting the deadband.

```
import stamp.core.*;
import stamp.util.os.* ;
import JBot.* ;

/**
 * Basic Photoresistor Compass Test Program
 * <p>
 * Tests the photoresistor circuits using CPU.rcTime.
 *
 * @version 1.0 10/2/02
 * @author Parallax, Inc.
 */

public class LineFollower1 extends Task {
    protected JBotInterface jbot = new RampingJBot ( new MultitaskingJBot () ) ;
    protected PhotoresistorSensor sensor =
        new PhotoresistorSensor ( CPU.pin5      // left pin
                                , CPU.pin3      // right pin
                                , true          // pin state
                                , 250           // rcTime limit
                                , 2             // timeout
                                , 30            // bias
                                , 1             // lower limit
                                , 2 ) ;        // deadband

    public void execute() {
        // Only uses one state
    }
}
```

```

    if ( sensor.obstacleDetected () ) {
        if ( sensor.obstacleDirection() <= 45 ) {
            // Obstacle to the left, turn right
            jbot.pivot(jbot.continuousRight) ;
        } else if ( sensor.obstacleDirection() >= 135 ) {
            // Obstacle to the right, turn left
            jbot.pivot(jbot.continuousLeft) ;
        } else if ( sensor.obstacleDistance ( 90 ) > 20 ) {
            // Too dark. Just stop and wait
            jbot.stop() ;
        } else {
            // Directly in front
            jbot.move(jbot.continuousForward) ;
        }
    } else {
        // No obstacle located
        jbot.move(jbot.continuousForward) ;
    }
}

public static void main() {
    new LineFollower1 () ;

    Task.TaskManager () ;
}
}

```

How the Black Stripe Follower Program Works

The line follower program just required changes to the sensitivity because the actions being performed by the photo compass programs were already correct. The main difference was the intensity of the light being used, or, in this case, the difference in the amount of light being reflected from the floor.

If you turn the lights out the J-Bot should stop. This is because of the range test when an obstacle is detected in front. This can be used to stop the J-Bot when it reaches the end of a path. You may have to adjust the value in the range test depending upon the light conditions.

Your Turn

Try a black stripe with a 45° turn in the middle of it.

Try a black strip with a 90° turn in it, and see if you can pick a deadband that will navigate it.

Remember, you may need to adjust your deadband to succeed in these maneuvers.

For either or both of the maneuvers above, find the upper and lower limits of deadband values with which the J-Bot still can successfully navigate.



Summary and Applications

This chapter focused on measuring the difference in light intensity and using it as a guide for the J-Bot. The `JAVA CPU.rcTime` method was used in conjunction with an RC circuit to measure each photoresistor. The exact resistance value of each photoresistor was disregarded in favor of the relative difference between the two values. This difference is a simple subtraction problem, but it can be used to gauge which direction is brighter.

Real World Example

Light has many applications in robotics and industrial control. Some examples include sensing the edge of a roll of fabric in the textile industry, determining when to activate streetlights at different times of the year, when to take a picture, or when to deliver water to a crop of plants.

Deadband is often a problem in navigation control systems. In terms of tracking and controlling machinery, deadband can result from the uncertainty in measurements due to mechanical connections. The result is that deadband is the area you don't know about and try to develop creative ways of dealing with it. On the other hand, deadband is also the way a thermostat works. In the context of maintaining temperature, differential gap control uses a built-in deadband region where no correction is made to the temperature.

J-Bot Application

As you can see, the J-Bot can do an interesting variety of tricks with a pair of photoresistors as its guide. It can point at light, move itself from a dark place into a light place, follow a guiding flashlight beam and follow a black stripe with turns in it on a white piece of paper. That's not bad for some inexpensive photoresistors, capacitors and resistors.



Questions and Projects

6

Questions

1. Name and describe the element in the photoresistors that changes resistance in response to illuminance.
2. What does the Javelin measure to infer the resistance in an RC circuit? What value must remain fixed in an RC circuit to infer a variable resistance? Why?
3. What are the increments of the **CPU.rcTime** measurement?
4. When the value of a photoresistor increases, what does that indicate?
5. How does the program for a light following J-Bot differ from that of a dark following J-Bot?
6. What role does deadband play in the J-Bot's tendency to move forward? What role does it play in the J-Bot's tendency to change direction?

Exercises

1. If you have a 10 μF capacitor and your **CPU.rcTime** value is 150, what is the resistance of the photoresistor? Hint: Use equation 4.1.
2. Re-derive Equation 4.1 using a 0.1 μF capacitor. What kinds of problems arise if the 0.1 μF capacitors replace the 0.01 μF capacitors? What effect does the increased RC value have on the measurement time? What effect does the measurement time have on servo performance?

Projects

1. Add sound to the J-Bot PhotoCompass2 program so a tone is sounded for a fixed duration after the J-Bot has been moving forward for at least one second. Use the TaskToneGenerator class to generate a half second tone.

The PhotoCompass2 task timer can be used since there are no sleep methods called that utilize the timer. Hint: Call `timer.mark()` when the J-Bot moves forward. A

`timer.timeoutSec(1)` will indicate when at least one second has elapsed.

Use boolean variables to keep track when the timer is running and a tone has occurred.

This program should never generate a continuous tone. Why? Hint: Consider the tone and the timeout duration.

2. Implement the previous exercise using different states instead of boolean variables to keep track of when forward movement is being tracked and if a tone has been played.
3. Add Whiskers to the J-Bot. Develop a line following track with obstacles placed in the way. Program the J-Bot to follow the line and also to check the whiskers to monitor for obstacles. Develop routines that guide the J-Bot around obstacles and back to the line.

TIP

Make sure to wrap each whisker with electrical tape around any part that might contact other circuits. The only things a whisker should be allowed to touch are obstacles and its own three-pin header post.

4. One of the interesting facets of relying on deadband for line following is that it can be adjusted purely in software. This project explores the relationship between deadband settings and stripe width.

Repeat the Your Turn exercises in Activity #4 with a 3.75 cm. wide black stripe. Do not adjust the width of your photoresistors; just the deadband settings. Repeat this activity again for a 2.5 cm. wide stripe. Make notes on the upper and lower deadband limits for each stripe width. In other words, find the highest and lowest deadband settings that work for successful stripe following. Graph your results. Is there any apparent mathematical relationship between deadband and stripe width? Use the graph to approximate a linear relationship, and develop a deadband equation. Test the equation on a 4.4 cm. wide stripe.