



Chapter #5: Tactile Navigation with Whiskers

Tactile Navigation

The Whiskers kit is so named because the kit's bumper switches look like whiskers, though some argue they look more like antennae. At any rate, whiskers give the J-Bot the ability to sense the world around it with tactile inputs. The J-Bot can use these whiskers to navigate only

by touch. Although the activities in this chapter focus on using just the whiskers, they can also be used with other sensors to increase the J-Bot's functionality.

Activity #1: Building and Testing the Whiskers

Parts

- (2) 10 k Ω resistors
- (2) 3-pin headers
- (2) 3/8" 4/40 male/female standoffs
- (2) 1/4" 4/40 machine screws
- (2) J-Bot bumper wires
- (2) Nylon washers size #4

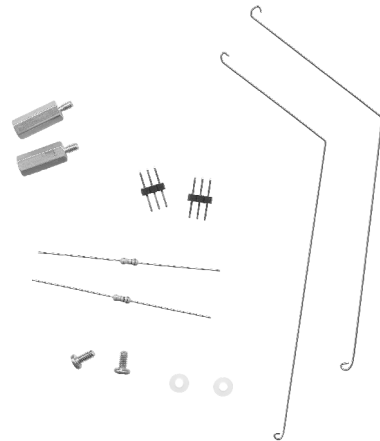


Figure 5.1 Whiskers parts

Build It!

Your #1 point Phillips screwdriver and quarter-inch combination wrench will come in handy here. Before getting started on whisker construction, take a close look at Figure 5.2. Use these pictures as a guide while constructing the mechanical part of the Whiskers kit. Figure 5.3 shows the whiskers wiring diagram. Follow it for making the necessary electrical connections.

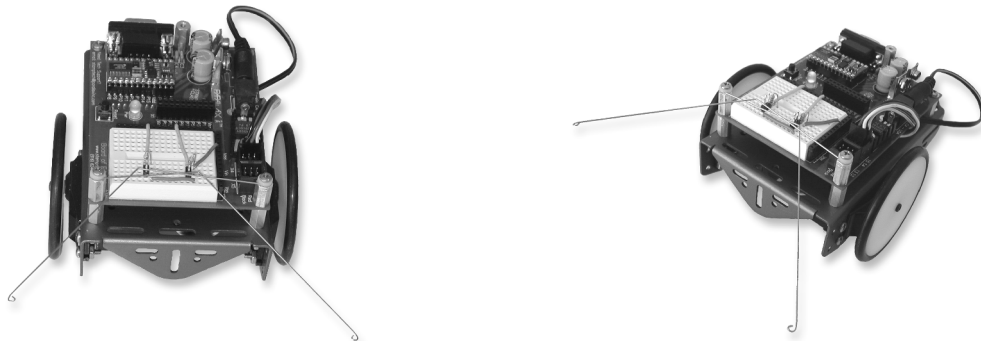


Figure 5.2: Pictures of J-Bot with Whiskers

- ❑ Remove the two front screws that hold your Board of Education to the front standoffs.
- ❑ Screw in the male/female standoffs included in the Whiskers kit in place of the screws that were just removed.

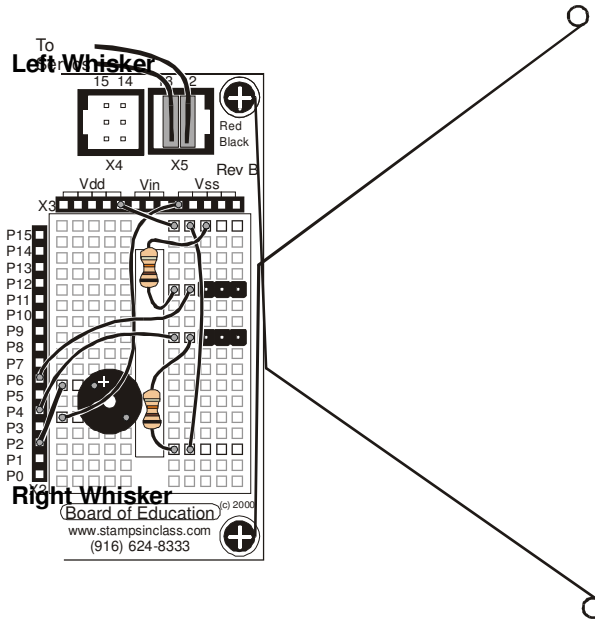


Figure 5.3: Whiskers wiring diagram.

TIP

Hold the male/female standoff by the servo port while turning the standoff between the JSDB and J-Bot chassis to tighten it. The standoff between the JSDB and chassis won't turn until you loosen the screw that holds it to the chassis. Make sure to retighten it when you're done.

- ❑ Place a nylon washer on top of each standoff.
- ❑ Thread each screw removed in the first step through the open loop of a whisker.
- ❑ Screw each screw into a standoff sandwiching the loop of the whisker between the screw head and the nylon washer. Make sure the whiskers are oriented as shown in Figure 3.2 and 3.3.

Whisker Inputs

Figure 5.4 is a schematic representation of the circuit you've just built. Each whisker is both the mechanical extension and the ground electrical connection of a normally open, single-pole, single-throw switch. The Javelin can be programmed to detect when a whisker is pressed. I/O pins connected to each switch

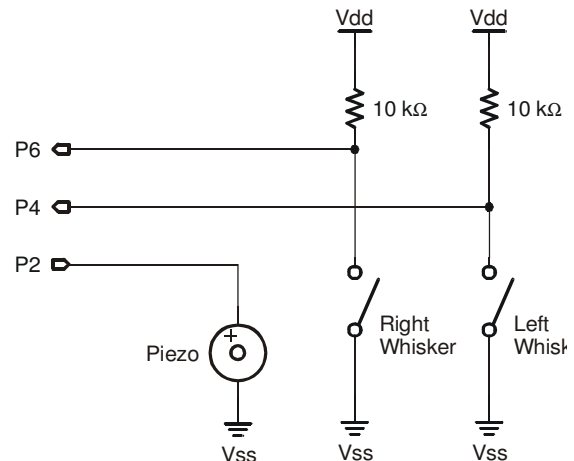


Figure 5.4: Whiskers Schematic.

circuit monitor the voltage at the 10 k Ω pull-up resistor. When a given whisker is not pressed, the voltage at the I/O pin connected to that whisker is 5 V (logic 1). When a whisker is pressed, the I/O line is shorted to ground, so the I/O line sees 0 V (logic 0).

Testing the Whiskers

Testing each whisker can be done with the Message window. This time, instead of displaying a printed message, the Message window interface is used to display the input seen by the I/O pins connected to the whiskers.

Each Javelin I/O pin can be controlled individually using the CPU class object. There are a number of class (static) methods that support reading and writing data to one or more pins. These methods include the following:

```
boolean readPin ( int portPin )
void writePin ( int portPin, boolean value )

byte readPort ( int port )
void writePort ( int port, byte value )

void setInput ( int portPin )
void setOutput ( int portPin )
```

The value of portPin must be one of the defined constants from CPU.pin0 to CPU.pin15. Using a value of 0 to 15 WILL NOT work. The port values are CPU.PORTA and CPU.PORTB. These ports map to the first and last 8 bit pins. The direction of individual pins can be set using setInput and setOutput methods as in CPU.setInput (CPU.pin0). The direction is changed if necessary when using readPin and writePin. The readPin method will be used in this chapter to access the whisker input pins and the writePin method to drive status LEDs.

The whisker test program uses pins 4 and 6 for input.

```
import stamp.core.*;

/**
 * Whisker switch test
 * <p>
 * Shows whisker switch status
 *
 * @version 1.0 9/10/02
 * @author Parallax, Inc.
 */

public class whisker1 {
    public static void main() {
        while ( true ) {
            System.out.print ( "P6=" ) ;
```

```
        System.out.print ( CPU.readPin ( CPU.pin6 ) ? 1 : 0 ) ;  
        System.out.print ( " P4=" ) ;  
        System.out.println ( CPU.readPin ( CPU.pin4 ) ? 1 : 0 ) ;  
        CPU.delay ( 5000 ) ;  
    }  
}  
}
```

The Whisker1 class is designed to test the whiskers to make sure they are functioning properly. It checks and displays the state of the Javelin I/O pins connected to the whiskers. All I/O pins default to input every time a JAVA program starts. This means that the I/O pins connected to the whiskers will function as inputs automatically. As an input, an I/O pin connected to a whisker will cause its bit in the **ins** register to display 1 if the voltage is 5 V (whisker not pressed) and 0 if the voltage is 0 V (whisker pressed). The Message window can be used to display these values.

- ❑ Enter and run the Whisker1 program
- ❑ This program makes use of the Message window, so leave the serial cable connected to the BOE while Whisker1.java is running.

- ❑ Note the values displayed in the Message window; it should display that both P6 and P4 are equal to 1.
- ❑ Check Figure 5.3 so you know which whisker is the “left whisker” and which whisker is the “right whisker”.
- ❑ Press the right whisker so that it touches the three-pin header on the right, and note the values displayed in the Message window again. It should now read: P6 = 1 P4 = 0.
- ❑ Press the left whisker into the left three-pin header, and note the value displayed in the Message window again. This time it should read: P6 = 0 P4 = 1.
- ❑ Press both whiskers against both three-pin headers. Now it should read P6 = 0 P4 = 0.

If the whiskers passed all these tests, you’re ready to move on; otherwise, check your program and circuit for errors.

Your Turn

Assume that you may have to test the whiskers at some later time away from a computer. Since the Message window won't be available, what can you do? One solution would be to program the Javelin so that it sends an output signal that corresponds to the input signal it's receiving. One way of doing this would be with a pair of LED circuits and a program that turns the LEDs on and off based on the whisker inputs. Figure 5.5 shows the parts of an LED circuit along with their schematic symbols.

Extra Parts

(2) Red LEDs

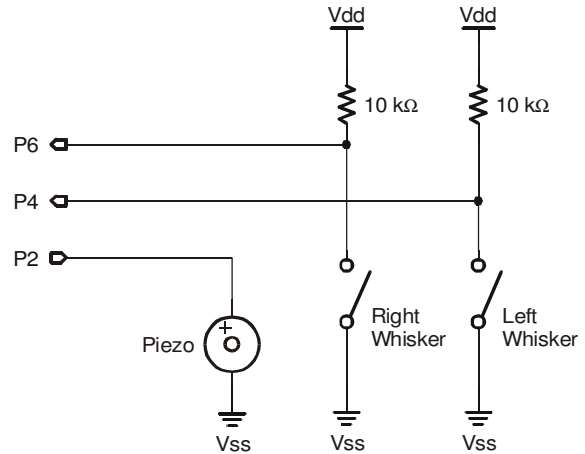
(2) 470 Ω resistors

Figure 5.5 Extra parts for testing the whiskers.

FYI

LED stands for light emitting diode. The terminal labeled 1 in figure 5.6 is the LED's cathode, and the terminal labeled 2 is the LED's anode. You can usually figure out which of the LED's two wire leads is connected to the cathode because it's shorter.

TIP

Just above where the wire leads come out of LED's plastic case, the outer rim looks round, but if you look carefully, there's a small area that's been milled flat. The lead that comes out of the plastic case closest to the milled flat spot is the cathode. If the LED's leads have been cut to the same length, look for the flat spot to figure out which lead connects to the LED's cathode.

- Construct the circuit shown in Figure 5.6.

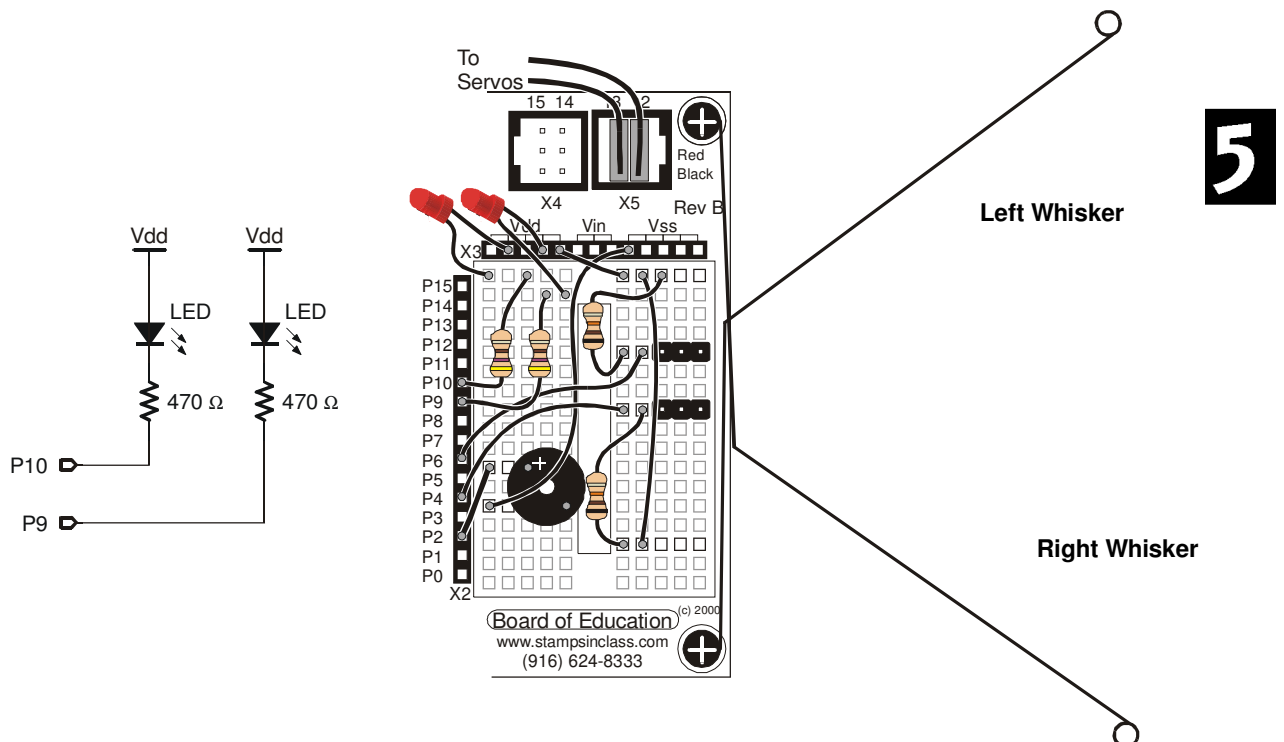


Figure 5.6 (a): add this LED circuit,

(b) so the Whiskers circuit looks like this when you're done.

- Add these commands to the beginning of Program Listing 5.1:

```
CPU.setOutput ( CPU.pin9 ) ;
CPU.setOutput ( CPU.pin10 ) ;
```

These commands change the direction of P9 and P10 from input to output. Now instead of listening for signals, they will be ready to send signals.

- Add these two commands immediately after the **debug** command in Program Listing 5.1:

```
CPU.writePin ( CPU.pin9, CPU.readPin ( CPU.pin4 ) ) ;
CPU.writePin ( CPU.pin10, CPU.readPin ( CPU.pin6 ) ) ;
```

These statements set the output values of P9 and P10 equal to the input values at **CPU.pin4** and **CPU.pin6** respectively. If **CPU.pin4** = 1, **CPU.pin9** is set to 1. This means that when **CPU.pin4** sees 5 V, **CPU.writePin** sends 5 V. If **CPU.pin4** is 0, which means it detects 0 V, then **CPU.writePin** will also be 0, sending 0 V.

- Run your modified version of Whisker1, and test the whiskers using the LEDs to indicate that the Javelin has detected a whisker being pressed.

Activity #2: Single Tasking Whiskers

In Activity #1, the Javelin was programmed to detect whether or not a given whisker was pressed. In this activity, the Javelin will be programmed to use this information to guide the J-Bot. When the J-Bot is rolling along and a whisker is pressed, it means the J-Bot bumped into something. A navigation program needs to take this input, decide what it means, and call a navigational routine to back up from the obstacle and go in a different direction.

Programming the J-Bot to Navigate Based on Whisker Inputs

Whisker2 class is called a roaming program. The program makes the J-Bot go forward until it encounters an obstacle. In this case, the J-Bot knows when it encounters an obstacle by bumping into it with one or both of its whiskers. As soon as the obstacle is detected by the whiskers, navigational routines and subroutines developed in Chapter 3 are used to make the J-Bot back up and turn. Then, the J-Bot resumes forward motion until it bumps into another obstacle.

When a whisker is pressed, due to an obstacle, the normally open switch closes. I/O pins P6 and P4 are set to input and used to monitor the states of the whiskers. The two whiskers may be in one of four states:

- (1) Both high - no objects detected
- (2) Left low, right high - object detected on the left
- (3) Right low, left high - object detected on the right
- (4) Both low - indicates a head-on collision with a wide object such as a wall

Whisker2.java shows an example of how the states of the whiskers can be used to select the appropriate J-Bot navigation routine. For example, state 1 means the J-Bot can continue forward. State 2 means that the J-Bot should back up, then turn right. State 3 means the J-Bot should back up and turn left, and state 4 would be a good time to back up and make a U-turn.

- ❑ Run Whisker2, and see how the J-Bot behaves when it bumps into a wall.

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

/**
 * Whisker switch test
 * <p>
 * Shows whisker switch status
 *
 * @version 1.0 9/10/02
 * @author Parallax, Inc.
 */

public class whisker2 {
```



```

public static void main() {
    JBotInterface jbot = new RampingJBot ( new FixedMovementJBot () ) ;

    while ( true ) {
        switch (    ( CPU.readPin ( CPU.pin6 ) ? 1 : 0 )
                  + ( CPU.readPin ( CPU.pin4 ) ? 2 : 0 ) ) {
            case 0:    // both low, backup and turn right
                jbot.stop () ;
                jbot.move ( -3 ) ;
                jbot.pivot ( -2 ) ;
                break ;

            case 1:    // P4 low, backup and turn left
                jbot.stop () ;
                jbot.move ( -2 ) ;
                jbot.pivot ( 2 ) ;
                break ;

            case 2:    // P6 low, backup and turn right
                jbot.stop () ;
                jbot.move ( -2 ) ;
                jbot.pivot ( -2 ) ;
                break ;

            case 3:    // neither low, go forward
                jbot.move ( jbot.continuousForward ) ;
                break ;
        }
    }
}

```

The mechanical design of the whiskers is by no means foolproof. Table 5.1 lists common problems you may encounter with some suggested solutions.

Table 5.1: Whisker Troubleshooting	
Problem	Try This
J-Bot backs up too far/not far enough.	Adjust the for. . . next arguments in the program listing. They may be increased or decreased to increase or decrease how far the J-Bot rotates when it turns/backups up.
J-Bot drives up side of wall because whiskers didn't catch hold of an object.	Increase the resistance of a whisker against the surface of an object by bending whiskers in a different angle. Alternatively, try dipping the whiskers in a coating material such as rubber cement.
Whiskers do not detect dead-center object.	Bend whiskers inward.
Switches don't appear to work properly.	Repeat Activity #1.

How Roaming with Whiskers Works

The **switch** statement in the **main** static method checks the whiskers and generates a number from 0 to 3. If both whiskers are pressed the value is 0 and the J-Bot is stopped, backed away from the obstacle and turned to the right. It could just as easily turn to the left. If just the left whisker is pressed then the value used in the **switch** statement is 1. The action is very similar. Likewise, if just the right whisker is pressed then the value is 2. Finally, a value of 3 indicates neither switch is closed and the J-Bot moves forward.

Note the different types of method calls to the jbot object. The forward movement uses no parameters because it is not moving for a fixed distance. The other movement methods are for a fixed number of inches and the jbot object handles starting and ending the movement. Once the movement is done the sensors are checked again. The J-Bot will move forward if there is nothing in the way.

Your Turn

The jbot movement method distance parameters can change where the J-Bot moves in response to an obstacle.

- ❑ Experiment with the method argument values in navigation routines in Whisker2.java.

Activity #3: Tactile Navigation - Whiskers and Multitasking

The single tasking roaming program is short and easy to understand. The FixedMovementJBot handles most of the work. Unfortunately, if anything else is going on then the program needs to be modified to handle more than one task. The alternative is to use the multitasking support. We start with a WhiskerSensor that extends BasicSensor presented in Chapter 2. We then use the sensor with a multitasking roaming program. We can use a different type of sensor by simply defining a new BasicSensor class. The following is the WhiskerSensor class file.

```
package JBot;

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

/**
 * Basic whisker sensor class
 * <p>
 * Provides obstacle detection using whisker sensors
 * Returns obstacle directions of 45, 90 and 135.
 *
 * @version 1.0 8/23/02
 * @author Parallax Inc.
```

```

*/

public class WhiskerSensor extends BaseSensor {
    /**
     * Indicate whether an obstacle has been detected.
     * Normally used when polling versus using an event.
     *
     * @returns obstacle detected
     */
    public boolean obstacleDetected () {
        return CPU.readPin ( CPU.pin6 ) | CPU.readPin ( CPU.pin4 ) ;
    }

    /**
     * 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 () {
        switch ( ( CPU.readPin ( CPU.pin6 ) ? 1 : 0 )
                + ( CPU.readPin ( CPU.pin4 ) ? 2 : 0 ) ) {
            case 0: // both low, backup and turn right
                return front ;

            case 1: // P4 low, backup and turn left
                return right ;

            case 2: // P6 low, backup and turn right
                return left ;

            default:
            case 3: // neither low
                return none ;
        }
    }

    /**
     * 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 0 ; // distance is always 0 regardless of direction
    }

    /**
     * Set minimum event notification distance.
     * Notification will not occur until an obstacle is
     * outside of this distance. The minimum value is 0.
     *
     * @param minimumDistance minimum number of inches to detect an obstacle
     */
    public void setMinimumEventDistance () {
        /* Default case is to ignore the minimum distance
         * For example, contact oriented sensors can only detect objects
         * when they are in contact with them.
         */
    }
}

```

```
// Protected classes for use by this class or subclasses

/**
 * Set notification event
 *
 * @param event Event object to notify when a change occurs
 */
public void setEvent ( Event event ) {
    this.event = event ;
}

/**
 * Cause event when obstacle status has changed.
 * May be called by subclass methods.
 */
protected void notifyEvent () {
    if ( event != null )
        event.notify () ;
}
}
```

The list for the sensor is longer than the whisker2.java program but this is primarily due to the comments. The obstacle detection is isolated in this class while the movement will be handled by the general multitasking roaming program presented next.

Reprogramming for Roaming with Whiskers

The AvoidObstacleTaskWhiskerTest1.java program operates in a similar fashion to whisker2.java. However, it processes the whisker inputs from the WhiskerSensor object and uses the AvoidObstacleTask to handle the movement. The AvoidObstacleTaskWhiskerTest1.java program is shown below.

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

/**
 * Test AvoidObjstacleTask class
 * <p>
 * Tun the J-Bot so it avoids obstacles.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class AvoidObstacleTaskWhiskerTest1 {
    public static void main () {
        new AvoidObstacleTask
            ( new WhiskerSensor ()
              , new RampingJBot ( new MultitaskingJBot ())) ;

        Task.TaskManager () ;
        System.out.println ( "All done" ) ;
    }
}
```

How Roaming with Whiskers Again Works

The `AvoidObstacleTaskWhiskerTest1` program is very simple because all the work is done by the `AvoidObstacleTask` object and the `WhiskerSensor` object. Additional tasks can be added without having a major affect on the movement and obstacle avoidance tasks. Note that the control event for the `RampingJBot` is a `MultitaskingJBot` task versus the single tasking `FixedMovementJbot`.

Your Turn

- ❑ Set the J-Bot on something so that when the program runs, the wheels don't touch the ground. This is so you can leave the J-Bot plugged into the serial cable while the program runs.
- ❑ Run the program and use the debugger window as you test the whiskers.
- ❑ Modify the `AvoidObstacleTask` so it operates like the `whisker2.java` program.

Activity #4: Tactile Memory - Whiskers and EEPROM

Lifting the wheels off the ground is fine for testing the basic interface but it is impractical to keep the J-Bot connected to the PC when it is roaming. The alternative is to record the obstacle information in the Javelin's EEPROM memory. This is the same memory used to store the Java code but rarely does the program use all the memory. In fact, since the program is actually copied from the EEPROM to RAM for execution then there must be some unused EEPROM space free because RAM is needed for creating objects when the program runs.

The obstacle detection is handled by the `AvoidObstacleTask`. This class is extended and called the `AvoidObstacleRecordingTask` show next.

```
package JBot;

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

/**
 * Simple obstacle avoidance task with EEPROM recording
 * <p>
 * This tries to stay away from obstacles using a sensor object.
 * The J-Bot will be moved in fixed increments.
 * Obstacle information is stored in EEPROM
 */
```

```

*
* @version 1.0 8/23/02
* @author Parallax Inc.
*/

public class AvoidObstacleRecordingTask extends AvoidObstacleTask {
    int memoryIndex ;
    int memorySize ;

    public AvoidObstacleRecordingTask ( BaseSensor sensor, JBotInterface jbot ) {
        super ( sensor, jbot ) ;

        memoryIndex = 1 ;
        memorySize = ( EEPROM.size () - 1 ) / 2 ;

        // Only record the first few obstacles. Must be under 255
        if ( memorySize > 10 ) {
            memorySize = 10 ;
        }

        // Store number of recordings that will be saved
        EEPROM.write ( 0, (byte) memorySize ) ;
    }

    protected void execute () {
        final int turnAround = 1 ;

        switch ( state ) {
        case initialState:
            if ( sensor.obstacleDetected () ) {
                int direction = sensor.obstacleDirection () ;

                // Exit if EEPROM area filled
                if ( memorySize == 0 ) {
                    stop () ;
                    break ;
                }

                // Record obstacle details in EEPROM
                EEPROM.write ( memoryIndex, (byte)direction ) ;
                EEPROM.write ( memoryIndex + 1, (byte)sensor.obstacleDistance
( direction ) ) ;
                memoryIndex += 2 ;
                -- memorySize ;

                if ( sensor.obstacleDistance ( direction ) < 2 ) {
                    // Too close, back up 2 inches, then turn around

                    jbot.move ( -2 ) ;
                    jbot.wait ( turnAround ) ;
                } else {
                    // Enough room to pivot away from object

                    if ( direction < 75 ) {
                        // Something to the left
                        jbot.pivot ( -2 ) ;
                    } else {
                        // Something in front or to the right
                        jbot.pivot ( 2 ) ;
                    }
                    jbot.wait ( turnAround ) ;
                }
            }
        }
    }
}

```

```

    } else {
        // Nothing detected. Move forward 1 inch

        jbot.move ( 1 ) ;
        jbot.wait ( initialState ) ;
    }
    break;

case turnAround:
    // J-Bot has backed up. Time to pivot 180 degrees

    jbot.pivot ( 4 ) ;
    jbot.wait ( initialState ) ;
    break;

default:
    // default catches bad states
    stop () ;
    break;
}
}
}

```

Three things are added to this class. First, are some new object variables. Second, the constructor initializes these variables using the EEPROM class methods. The `memorySize` variable is used to store the number of entries that will be recorded. The maximum value is 255 because EEPROM data is store in bytes with a value between 0 and 255. It is possible to store a 16-bit integer in two bytes but we leave this as an exercise. The memory Index starts at an offset of 1 so the number of entries can be stored in the first byte at offset 0. The obstacle information will be stored after this byte.

Finally, data is stored in the `execute` method's `initialState`. This is where obstacles are detected using a sensor. At this point the sensor is the `WhiskerSensor`. The J-Bot will stop after the set number of obstacle entries are stored. We start with a value of 10 in the constructor so the J-Bot will not run too long before it stops.

Once the J-Bot stops it can be picked up and connected to the PC. The J-Bot can be turned off if necessary because data stored in the EEPROM is maintained even when power is off. The next step is to download the reporting program that will read the EEPROM contents and display the information in the Message window. The data saved in the EEPROM will not be overwritten by the new program because it is loaded at the other end of memory. The following is the `DumpObstacle` program that displays the information from the EEPROM.

```

import stamp.core.*;

/**
 * Dumps data stored in EEPROM by AvoidObstacleRecordingTask.
 * <p>

```



```
* Reads the data from the EEPROM memory and displays it
* in the Message window.
*
* @version 1.0 9-20-02
* @author Parallax, Inc.
*/

public class DumpObstacle {
    public static void main() {
        int memoryIndex = 1 ;

        for ( int memorySize = (int) EEPROM.read ( 0 )
              ; memorySize > 0
              ; -- memorySize ) {

            System.out.print    ( memoryIndex ) ;
            System.out.print    ( " Direction: " ) ;
            System.out.print    ((int) EEPROM.read ( memoryIndex )) ;
            System.out.print    ( " Distance: " ) ;
            System.out.println  ((int) EEPROM.read ( memoryIndex + 1 )) ;
            memoryIndex += 2 ;
        }
    }
}
```

This program is very simple. It assumes that the first EEPROM byte is the number of 2 byte entries that start at offset 1. The offset and the two bytes are printed on each line. The two values are prefixed by Direction or Distance so you don't have to remember which number is which. The information may scroll by very quickly but the Message window will retain the information. Scroll the window up to see information printed when the program starts.

Your Turn

The pair of classes, `AvoidObstacleRecordingTask` and `DumpObstacle`, must use the same offsets and entries if the proper information is to be observed later. It may also be useful to add movement and pivot steps to this information. Remember to make the same kind of changes to both classes.



Summary and Applications

In this chapter, instead of navigating from a pre-programmed list, the J-Bot was programmed to navigate based on sensory inputs. In this case, the sensory inputs were whiskers. The Javelin was programmed to test these whisker sensors and display the test results using two different media, the Message window and LEDs. The obstacle information using the multitasking version of the program was also modified to store information in the EEPROM.

When properly wired, these switches can show one voltage (5 V) at the switch's contact point when it's open, and a different voltage (0 V) when it's closed. Voltages of 5 and 0 V are transistor-transistor logic (TTL) levels, and the Javelin's input registers interpret these levels as "1" and "0," respectively.

JAVA programs were developed to make the Javelin check the whiskers between each servo pulse. Based on the state of the whiskers, the programs' **main:** routines either made the J-Bot continue forward, or called navigational routines developed in the previous chapter to guide the J-Bot away from obstacles.

Real World Example

Automated sensors are all around you. When you go to a grocery or convenience store, sensors are often responsible for opening the door for you. Microcontrollers scan keypads in a fashion similar to the way the Javelin scans the whiskers to detect whether or not they have been pressed. The information is processed and results as an output. In the case of a door opener, the result is a servo- or motor-controlled door being opened.

Robotic machinery of many shapes and sizes also relies on a variety of tactile switches wired similarly to the whiskers. Robotic arms sometimes use these switches to detect when they've encountered the object they are programmed to pick up and place elsewhere. Factories use these switches to count objects on a production line, and also for aligning objects for industrial processes. In all these instances, the switches provide inputs that dictate some other form of programmed output. Be it a calculator, robot or a production line, the switch input is electronically monitored. Based on the state of the switches, the calculator display updates, the robot arm grabs the object, or the factory production line reacts with motors or servos to guide the product in a pre-programmed fashion.

J-Bot Application

This chapter introduced input-based J-Bot navigation using real sensors. The next three chapters will focus on using different types of sensors to give the J-Bot vision. Both vision and touch open up a lots of opportunities for the J-Bot to navigate in increasingly complex environments.



Questions and Projects

5

Questions

1. What kind of electrical connection is a whisker?
2. If an I/O pin is set to output, what register does this effect?
3. When a whisker is pressed, what voltage occurs at the I/O pin monitoring it? What binary value will occur in the input register? If I/O pin P8 is used to monitor the input pin, what value does `in8` have when a whisker is pressed, and what value does it have when a whisker is not pressed?
4. What direction does an I/O pin have to be set to make an LED circuit function?
5. Which whisker is `CPU.pin6` connected to? How about `CPU.pin4`?

Exercises

1. What happens if the `CPU.delay` call is removed from `whisker1.java`?
2. Implement `whisker2.java` using if statements instead of the switch statement.
3. The `AvoidObstacleRecordingTask` saves an obstacles direction and distance in the EEPROM. Add the movements used in response to an obstacle. Remember to change the `DumpObstacle` file as well.

Projects

1. Modify `whicker2.java` so that the J-Bot moves backward slowly while both whiskers are pressed. Otherwise, it stays still. Modify the program further so that the J-Bot rotates counterclockwise when the left whisker is pressed and clockwise then the right whisker is pressed. When the program is finished, fine tune the speed response so that it appears that you are pushing the J-Bot around by its whiskers.
2. Challenge: `whisker2.java` so that the J-Bot travels in a circular path. Modify it so that if you tap the inside

whisker, the circular path will become 5 cm. narrower in diameter. Also make it so that if you tap the whisker on the outside of the J-Bot's circular path, the diameter will increase by 5 cm.

When you've got whisker control over the diameter of the J-Bot's circle, program the J-Bot to remember this diameter, even after the power is reset. The **write** command can be used to save data to EEPROM. EEPROM data is called non-volatile. Whereas the Javelin's RAM is erased with each reset (volatile), the EEPROM can save the data for use the next time the Javelin gets power (non-volatile).