» The sequence for this handleSerial() method is a bit complicated, so Figure 6-18 shows it in a flowchart. The actual code follows:

```
void handleSerial() {
  inByte = Serial.read();
  delay(2);
  // comma comes only in the CONNECT,<address> message:
  if (inByte == ',') {
    // send an initial message:
    sendData();
    // update the connection status:
    connected = true;
  }

  //S comes only in the DISCONNECT message:
  if (inByte == 'S') {
    // turn off the analog LED:
    analogWrite(analogLed, 0);
    connected = false;
  }
  //R comes only in the NO CARRIER and NO ANSWER messages:
  if (inByte == 'R') {
    // turn off the analog LED:
    analogWrite(analogLed, 0);
    connected = false;
  }

  if (connected) {
    // save only ASCII numeric characters (ASCII 0 - 9):
    if ((inByte >= '0') && (inByte <= '9')){
      inString[stringPos] = inByte;
      stringPos++;
    }
    // if you get an asterisk, it's the end of a string:
    if (inByte == '*') {
      // convert the string to a number:
      int brightness = atoi(inString);
      // set the analog output LED:
      analogWrite(analogLed, brightness);

      // put zeroes in the array
      for (int c = 0; c < stringPos; c++) {
        inString[c] = 0;
      }
      // reset the string pointer:
      stringPos = 0;
      // as this  byte (*) is the end of an incoming string,
      // send out your reading in response:
      sendData();
    }
  }
}
```

handleSerial() calls the sendData() method to read the sensor and send it out. Here it is:

```
void sendData() {
  // indicate that we're sending using the tx LED:
  digitalWrite(txLed, HIGH);
  Serial.print(readSensor(), DEC);
  // string termination:
  Serial.print("*");
  // turn off the tx LED:
  digitalWrite(txLed, LOW);
}
```

That's the whole program. Run this on your microcontroller, filling in the address of your second radio for the address in the remoteAddress array above. Then connect your second radio to your USB-to-serial adaptor, if it's not there already, and open a serial terminal window to it. The microcontroller will continue to try to connect to this radio every five seconds until a connection is established, and then it will start sending sensor values through. Your initial messages should look like this in the serial terminal:

```
CONNECT,00A096152B36
105*
```

You can respond as if you were sending your own sensor values, and the microcontroller will fade the LED on pin 9 accordingly. Try this:

```
12*
120*
255*
1*
```

The LED should start dim, get brighter, then brightest, then get very dim. You should get four sensor readings in response.

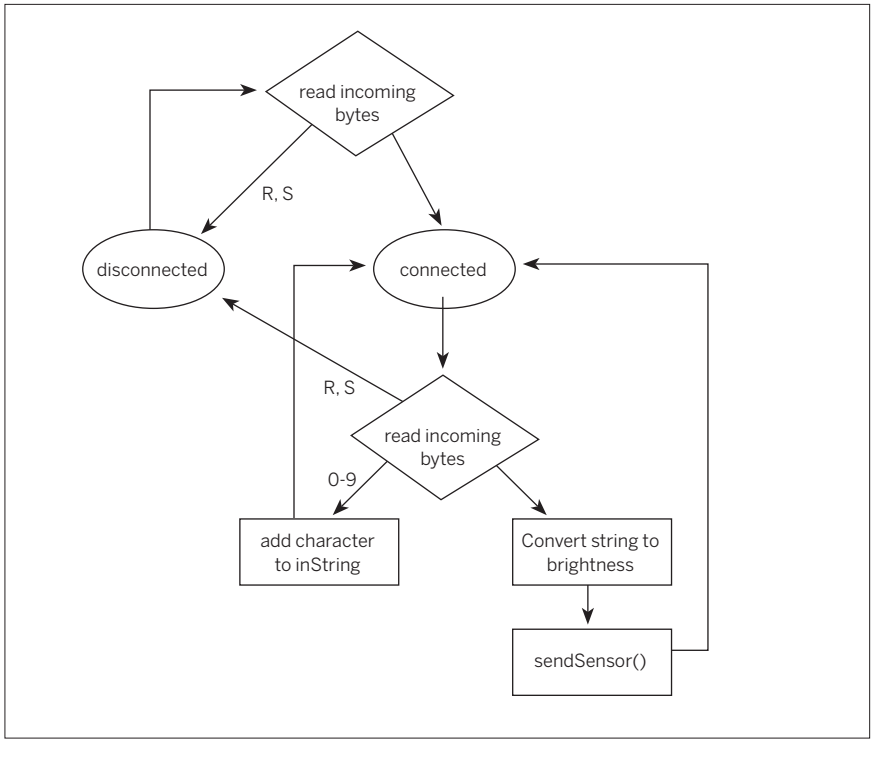


Figure 6-18
Flowchart of the handleSerial() method.

Step 3: Connecting Two Microcontrollers Via Bluetooth

If you've been following the parallels between the XBee example and this one, you probably know what's coming. Build the same circuit for your second microcontroller, using the second radio. Then change the Bluetooth address in the earlier program to be the address of the first radio, and program the second microcontroller. Then reset both microcontrollers. They will both attempt to connect to the other, and when either connects, they'll begin exchanging data. If you have trouble getting them to connect, change the connectTimeout variable so that they don't have the same value. This program won't work with every Bluetooth connection you have to make. When connecting to personal computers or mobile phones, you have to take different approaches, depending on the specific messages that those devices use. But the basic sequence should be similar enough to this that it will serve as a useful starting place.

X

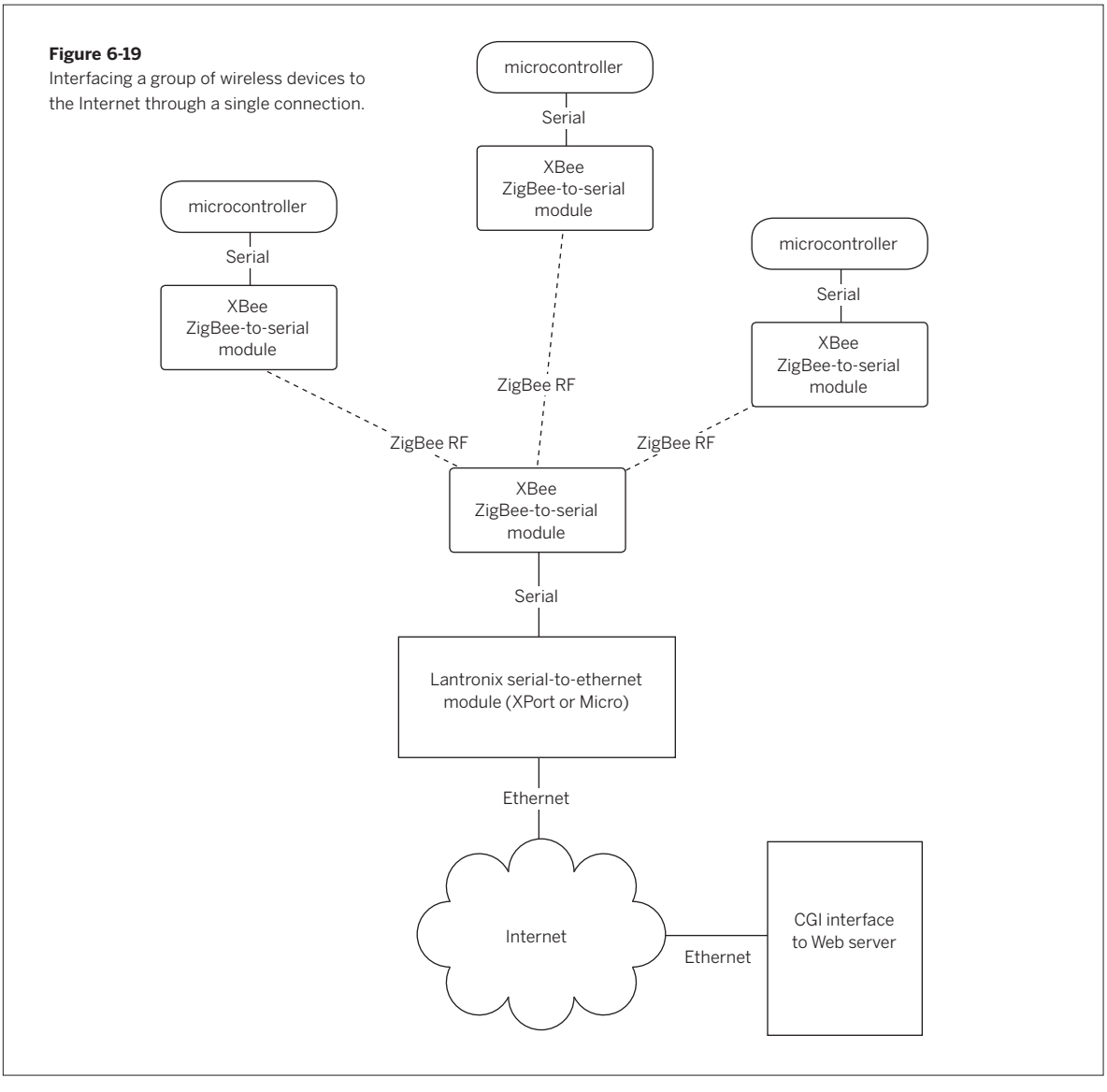


Figure 6-19
Interfacing a group of wireless devices to the Internet through a single connection.

What About Wi-Fi?

So far, you've seen the most basic serial radios in action in the transmitter-receiver project, and more advanced radios in the transceiver projects. If you're thinking about networks of microcontrollers, you're probably wondering whether you can connect your projects to the Internet and to each other using Wi-Fi. The answer is: yes, you can.

In Project #5, you connected a microcontroller to the internet using a Lantronix Micro serial-to-Ethernet module. You could replace the Micro with a WiMicro and build the exact project, with only some minor configuration changes on the WiMicro.

It's worth mentioning why Wi-Fi isn't more pervasive in embedded wireless projects. First, there's the cost. Most of the serial-to-Wi-Fi modules on the market are more expensive than equivalent transceivers implementing other protocols. For example, the WiMicro costs about $165. DigiConnect's equivalent module, the plaintively named Wi-ME, costs about $130. Other wireless Ethernet-to-serial modules on the market are in the same price range — over $100. When compared to $20 for XBee and other serial transceivers, or even the $60 price range typical for many Bluetooth modules, Wi-Fi isn't exactly a bargain.

Besides the cost, however, there's another factor to consider before going Wi-Fi. Most Wi-Fi modules consume more electrical power than the other radio types mentioned here. Because one of the main reasons for going wireless is to go mobile, you'll be eating up battery life with Wi-Fi radios. So they're great when they're conveniently already built into a product, but if you're building from the ground up, it's worth considering other alternatives. One common solution for wireless projects that need Internet access is to make an RF-to-Ethernet bridge. For example, one of the XBee radios from the earlier project could be interfaced to a Lantronix XPort or Micro module to act as the Internet connection for a whole collection of XBee-enabled objects. Figure 6-19 shows a typical network setup for such a system.
X

Buying Radios

You've seen a few different kinds of wireless modules in this chapter. Though they do the job well, they're not the only options on the market. You should definitely shop around for modules that suit your needs. Following are a few things to consider in choosing your radios.

The wisest thing you can do when buying your radios is to buy them as a set. Matching a transmitter from one company to a receiver from another is asking for headaches. They may say that they operate in the same frequency range, but there's no guarantee. Likewise, trying to hack an analog radio, such as that from a baby monitor or a walkie-talkie, may seem like a cheap and easy solution, but in the end, it'll cost you time and eat your soul. When looking for radios, look for something that can take the serial output of your microcontroller. Most microcontrollers send serial data at TTL levels, with 0V for logic 0 and 3.3V or 5V for logic 1. Converting the output to RS-232 levels is also fairly simple, so radios that can take those signals are good for your purposes.

Consider the data rate you need for your application — and, more specifically, for the wireless part of it. You may not need high-speed wireless. One common use for wireless communication in the performance world is to get data off the bodies of performers without a wired connection, in order to control MIDI performance devices like samplers and lighting dimmers. You might think that you need your radios to work at MIDI data rates to do this, but you don't. You can send the sensor data from the performers wirelessly at a low data rate to a stationary microcontroller, then have the microcontroller send the data on via MIDI at a higher data rate.

Most of the inexpensive radio transmitters mentioned previously send data at relatively low rates (under 9600 bps). Given the noisy nature of RF, it's wise to not attempt to send at the top speed if you don't need to. In the transmitter-receiver project, the radio pair can operate at up to 4800 bps, but you are sending at only 2400 bps. Try it at 4800 bps, and you'll notice more errors. The 2.4Ghz radios used for Bluetooth, ZigBee, and wireless Ethernet are exceptions to this rule. They generally operate reasonably at much higher data rates, because they've got a microcontroller on board to manage the data flow.

Consider the protocols of the devices that you already have at your disposal. For example, if you're building an object to speak to a mobile phone or a laptop computer, and there's only one object involved, consider Bluetooth. Most laptops and many mobile phones already have Bluetooth radios onboard, so you'll need only one radio to do the job. It may take some work to make your object compatible with the commands specific to the existing devices you're working with, but if you can concentrate on that instead of on getting the RF transmission consistent, you'll save yourself a lot of time.

X

Conclusion

Wireless communication involves some significant differences from wired communication. Because of the complications, you can't count on the message getting through like you can with a wired connection, so you have to decide what you want to do about it.

If you opt for the least expensive solutions, you can just implement a one-way wireless link with transmitter-receiver pairs and send the message again and again, hoping that it's eventually received. If you spend a little more money, you can implement a duplex connection, so that each side can query and acknowledge the other. Regardless of which method you choose, you have to prepare for the inevitable noise that comes with a wireless connection. If you're using infrared, incandescent light and heat act as noise, and if you're using radio, all kinds of electromagnetic sources act as noise, from microwave ovens to generators to cordless phones. You can write your own error-checking routines, but increasingly, wireless protocols like Bluetooth and ZigBee are making it possible for you to forget about that, because the modules that implement these protocols include their own error correction.

Just as you started learning about networks by starting with the simplest one-to-one network in Chapter 2, you started with wireless connections by looking at simple pairs in this chapter. In the next chapter, you'll look at peer-to-peer networks, in which there is no central controller, and each object on the network can talk to any other object. You'll see both Ethernet and wireless examples.

X

Urban Sonar by Kate Hartman, Kati London, and Sai Sriskandarajah.
The jacket contains four ultrasonic sensors and two pulse sensors. a microcontroller in the jacket communicates via Bluetooth to your mobile phone. The personal space bubble as measured by the sensors and your changing heart rate as a result of your changing personal space paint a portrait of you that is sent over the phone to a visualizer on the internet.


FRONT
LEFT
RIGHT
BACK
START
RESET
Oct 23, 2006 8:54:55 PM
1. FRONT: 214
2. RIGHT: 101
3 .BACK 122
4. LEFT 122
HEART RATE: 83

7

MAKE: PROJECTS 

Sessionless Networks

So far, the network connections you've seen in this book have mostly been dedicated connections between two objects. Serial communications involve the control of a serial port. Mail, web, and telnet connections involve a network port. In all of these cases, there's a device that makes the port available (generally a server), and a client that requests access to the port (a client). The networked pong application in Chapter 5 was a classic example of this. In that application, the server handled all the communications between the other devices. In this chapter, you'll learn how to make multiple devices on a network talk to each other directly, or talk to all the other devices at once.

Perform-o-shoes by Andrew Schneider.
The shoes exchange messages with a multimedia computer via XBee radio. When you moonwalk in the shoes, your pace and rhythm controls the playback of music from the computer.

Look, Ma: No Microcontroller!

Since the beginning of the book, you've been working with programmable microcontrollers, writing the whole program yourself. You don't always have to do this. The various network devices you've been working with — the Lantronix devices, and XBee and Bluetooth Radios — all have their own microcontrollers built in. Some of them have their own digital and analog input and output pins. You can configure these devices to activate and respond to these I/O pins with network messages, but you need to learn their protocols first. To give you some examples of how you can use these network modules to their full potential, none of the projects in this chapter use programmable microcontrollers.

Sessions versus Messages

In Chapter 5, you learned about the Transmission Control Protocol, TCP, which is used for much of the communication on the Internet. To use TCP, your device has to request a connection to another device. The other device opens a network port, and the connection is established. Once the connection is made, information is exchanged, then the connection is closed. The whole request-connect-converse-disconnect sequence constitutes a session. If you want to talk to multiple devices, you have to open and maintain multiple sessions. Sessions characterize TCP communications.

Sometimes you want to make a network in which objects can talk to each other more freely, switching conversational partners on the fly, or even addressing the whole group if the occasion warrants. For this kind of communication, there's another protocol used on the Internet, called the User Datagram Protocol, or UDP.

Unlike the session-based TCP, UDP communication is all about messages. UDP messages are called datagrams. Each datagram to be sent is given a destination address and is sent on its merry way. Once the sender sends a message, it forgets about it. There is no two-way socket connection between the sender and receiver. It's the receiver's responsibility to decide what to do if some of the datagram packets don't arrive, or if they arrive in the wrong order.

Because UDP doesn't rely on a dedicated one-to-one connection between sender and receiver, it's possible to send a broadcast UDP message that's sent to every other object on a given subnet, For example, if your address is 192.168.1.45, and you send a UDP message to 192.168.1.255, everybody on your subnet receives the message. Because this is such a handy thing to do, a special address is reserved for this purpose: 255.255.255.255, which is the limited broadcast address, goes only to addresses on the same LAN, and does not require you to know your subnet address. This address is useful for tasks like finding out who's on the subnet.

The advantage of UDP is that data moves faster, because there's no error checking. It's also easier to switch the end device that you're addressing on the fly. The disadvantage is that it's less reliable byte-for-byte, as dropped packets aren't resent. UDP is useful for streams of data where there's a lot of redundant information, like video or audio. If a packet is dropped in a video or audio stream, you may notice a blip, but you can still make sense of the image or sound.

The relationship between TCP and UDP is similar to the relationship between Bluetooth and 802.15.4. Bluetooth devices have to establish a session to each other to communicate, whereas 802.15.4 radios like the XBee radios in Chapter 6 communicate simply by sending addressed messages out to the network without waiting for a result. Like TCP, Bluetooth is reliable for byte-critical applications, but less flexible in its pairings than 802.15.4.

X

Who's Out There? Broadcast Messages

The first advantage to sessionless protocols like UDP and 802.15.4 is that they allow for broadcasting messages to everyone on the network at once. Although you don't want to do this all the time, because you'd flood the network with messages that not every device needs, it's a handy ability to have when you want to find out who else is on your network. You simply send out a broadcast message asking "Who's there?" and wait for replies. You could write your own methods for doing this, but most of the time you won't have to. Broadcast querying is such a useful technique that most manufacturers of network devices include it as part of their products' functionality. Lantronix uses a specific UDP message to query a subnet for any of their devices. Similarly, the XBee devices from Maxstream have a special broadcast query command.

Querying for Lantronix Devices Using UDP

All the Lantronix devices are preprogrammed to respond via UDP if they receive a particular UDP message. Knowing this, you can find out the IP address of any Lantronix device on your subnet by sending this message. They reserve a special port for status queries: port 30718. When you send UDP messages to that port, you get back a status report from the device. This is handy if you've got a few Lantronix devices on the network and need to know their addresses.

To test this, all you need is a Lantronix device that's powered and connected to your Ethernet or Wi-Fi network, and a program that sends UDP datagrams. The network query messages don't involve any communication over the devices' serial ports, so it doesn't matter what you've got connected to the serial port. You can reuse the pong client you built in Chapter 5 or the air quality meter from Chapter 4. In fact, it would work if you just provided power to the Lantronix device.

There's no way to send UDP messages using the Processing Network library, but there's a good free UDP library available from the Hypermedia Atelier at hypermedia.loeil org/processing/. You can also find it linked from the Libraries page of the main Processing site, at www.processing.org/reference/libraries/index.html. To use it, make a new directory called **udp/** in the **libraries/** subdirectory of your Processing application directory. Then unzip the contents of the download and drop them in the directory you created. After that, restart Processing and you're ready to use the UDP library.

Try It Here's a Processing sketch using that library that sends out a search string for Lantronix devices on a subnet, waits for responses, and then prints them. Run the program, make sure the applet window has focus, and press any key to send the UDP broadcast message.

```
/*
  Lantronix UDP Device Query
  Language: Processing

 Sends out a UDP broadcast packet to query a subnet for Lantronix
 serial-to-ethernet devices.
 Lantronix devices are programmed to respond to UDP messages
 received on port 30718.  If a Lantronix device receives the string
 0x00 0x00 0x00 0xF6, it responds with a UDP packet containing the
 status message on port 30718.

 When the program starts, press any key on the keyboard and watch
```

»

The response is stored in an array called inData[], and you can see in the code how that array breaks down. Byte 3, for example, is a byte that tells us what follows. If that byte's value is 0xF7, then the next 16 bytes contain the device's basic configuration, including its firmware version, checksum and device type. Following that, in bytes 24 to 30 of the array, is the device's MAC address. Because the MAC address is usually on a sticker on the side of the device, this is a handy way to find out who's who.

Continued from previous page.

```
 the message pane for responses.

 See the Lantronix integration guide from http://www.lantronix.com
 for the details.

 This program uses the Hypermedia UDP library, available from
 http://hypermedia.loeil.org/processing/.

*/

// import UDP library
import hypermedia.net.*;


UDP udp;                        // define the UDP object
int queryPort = 30718;          // the port number for the device query

void setup() {
  // create a new connection to listen for
  // UDP datagrams on query port;
  udp = new UDP(this, queryPort);

  // listen for incoming packets:
  udp.listen( true );
}

//process events
void draw() {
  // Twiddle your thumbs. Everything is event-generated.
}

/*
 send the query message when any key is pressed:
 */
void keyPressed() {
  byte[] queryMsg = new byte[4];
  queryMsg[0] = 0x00;
  queryMsg[1] = 0x00;
  queryMsg[2] = 0x00;
  // because 0xF6 (decimal value 246) is greater than 128
  // you have to explicitly convert it to a byte:
  queryMsg[3] = byte(0xF6);

  // send the message
  udp.send( queryMsg, "255.255.255.255", queryPort );
  println("UDP Query sent");
}

/*
```

The responses you get from the query message will look like this:

```
UDP Query sent
response from 192.168.1.128 on port 30718
Received response: F6


response from 192.168.1.47 on port 30718
Received response: F7
MAC Addr:  00 20 4A 8A 1E 48

response from 192.168.1.116 on port 30718
Received response: F7
MAC Addr:  00 20 4A 8F A1 6F

response from 192.168.1.236 on port 30718
Received response: F7
MAC Addr:  00 20 4A 66 A9 DD
```

NOTE: Notice that the first response is from the IP address of your computer. When you send a broadcast message, it comes back to you as well!

Continued from opposite page.

```
  listen for responses via UDP
 */
void receive( byte[] data, String ip, int port ) {
  String inString = new String(data);  // incoming data converted to string
  int[] intData = int(data);           // data converted to ints
  int i = 0;                           // counter

  // print the result:
  println( "response from "+ip+" on port "+port );

  // parse the response for the appropriate data.
  // if the fourth byte is <F7>, we got a status reply:
  print("Received response: ");
  println(hex(intData[3],2));
  if (intData[3] == 0xF7) {
    // MAC address of the sender is bytes 24 to 30 (the end):
    print("MAC Addr: ");
    for (i=24; i < intData.length; i++) {
      print(" " + hex(intData[i], 2));
    }
  }
  // print two blank lines to separate messages from multiple responders:
  print("\n\n");
}
```

Querying for XBee Radios Using 802.15.4 Broadcast Messages

Like the Lantronix devices, the XBee radios have a command for querying the air for any available radios. This is referred to as node discovery. When given the AT command ATND\r, the XBee radio sends out a broadcast message requesting that all other radios on the same personal area network (PAN) identify themselves. If a radio receives this message, it responds with its source address, serial number, received signal strength, and node identifier.

NOTE: To do node discovery, your radios must have version 10A1 of the XBee firmware or later. See the sidebar on upgrading the firmware on XBee radios for more details.

For the purposes of this exercise, you'll need at least two XBee radios connected to serial ports on your computer. The easiest way to do this is by using the USB-to-serial converter you've been using all along. The circuit from Figure 6-5 will work for this.

Once you've got the radios connected and working, open a serial terminal connection to one of them and issue the following command (you can use the XBee Terminal Processing program from Chapter 6 to communicate more easily with the XBee): +++ then wait for the radio to respond with OK. Then type (remember, \r means carriage return, so press Enter or Return instead of \r): ATND\r.

If there are other XBee radios on the same personal area network in range, the radio will respond after a few seconds with a string like this:

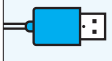
```
1234
13A200
400842A7
28
TIGOE1

5678
13A200
400842A9
```

```
1E
TIG0E3
```

Each grouping represents a different radio. The first number is the radio's source address (the number you get when you send it the command string `ATMY`). The second is the high word of the radio's serial number, the third is the low word. The fourth number is a measurement of the radio's received signal strength; in other words, it tells you how strong the radio signal of the query message was when it arrived at the receiving radio. The final line gives the radio's node identifier. This is a text string, up to 20 characters long, that you can enter into the radio to give it a name you can remember. You didn't use this function in Chapter 6, so your radios may not have node identifier strings. If you want to set the node identifier for further use, type: `ATNI myname, WR\r`

Replace myname with the name you want.

Broadcast messages can be useful for reasons other than for identification queries like the ones shown here, but they should be used sparingly, because they create more traffic than is necessary. In the next project, you'll use broadcast messages to reach all the other objects in a small, closed network.



Upgrading the Firmware on XBee Radios

To use the node discover and node identifier and some of the other XBee AT commands covered in this chapter, your XBee radios need to be upgraded to at least version 10A1. To check the firmware version of your radios, send the following command: `ATVR\r`. The radio will respond: 10A2. If the number is 10A1 or above (remember, it's in hexadecimal), you're good to go. If not, go to www.maxstream.net/support/downloads.php and download the X-CTU software. Bad news, Mac OS X users: it only runs on Windows (though you can run it under Parallels, if your Mac runs Parallels).

Before you can download software, you'll need to add a couple of connections between your serial port and your radio. Specifically, connect the DTR and RTS connections from the XBee (pins 9 and 16, respectively) to the same pins of your serial port. Figure 7-2 shows how do to this on the SparkFun FTDI board. Once you've made these connections, you're ready to run the software.

Once you've installed the software, launch it. On the PC Settings tab, you'll be able to select the serial port that your XBee radio is attached to. Pick a port, and leave the settings at their defaults. Click the Modem Configuration tab and you'll get to the tab where you can update the firmware. Click the Read button to read the current firmware on your radio. You'll get a screenful of the settings of your radio, similar to that in Figure 7-1. The firmware version is shown in the upper righthand corner. You can pull down that menu to see the latest versions available. Pick the latest one (anything after 10A1), then check the Always Update Firmware checkbox. Leave the Function Set menu choice set to XBEE 802.15.4. Then click the Write button. The software will download the new firmware to your radio, and you're ready to go. The X-CTU software is useful to keep around, because it also lets you change and record your radio's settings without having to use the AT commands.

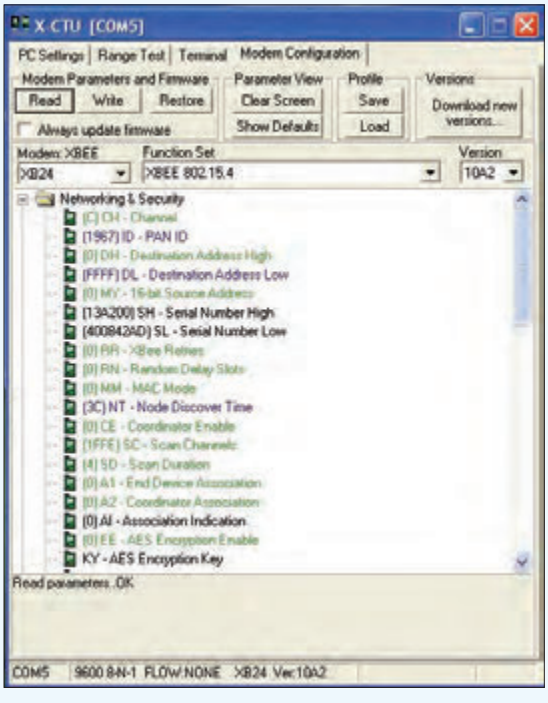


Figure 7-1. The X-CTU Modem Configuration tab.

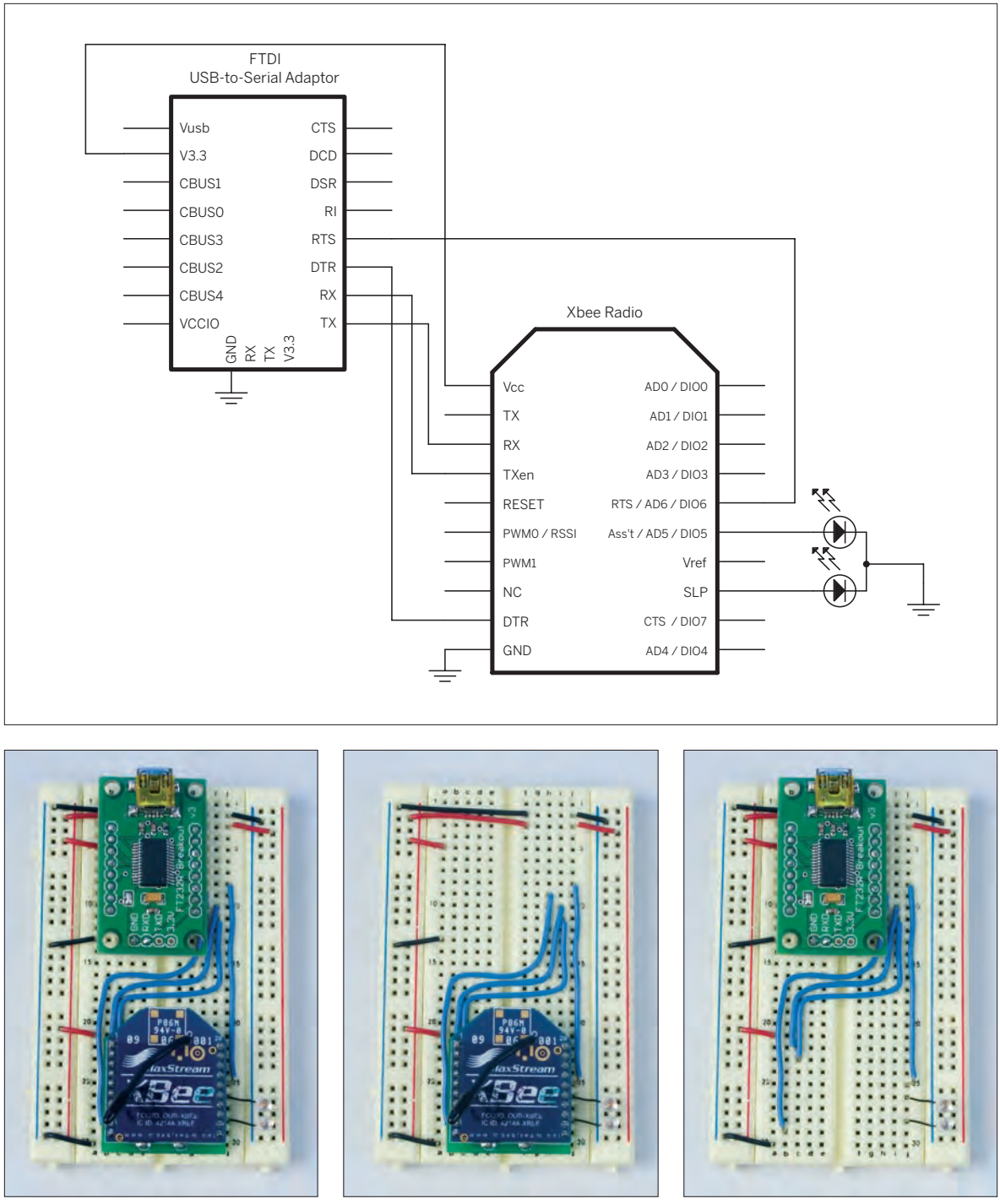


Figure 7-2. Connecting the DTR and RTS pins on the XBee radio to the FTDI USB-to-serial adaptor, including the wiring under the radio and the FTDI adaptor.

Project 12

Reporting Toxic Chemicals in the Shop

If you've got a workshop to take care of, you'll appreciate this project. You're going to attach a volatile organic compound (VOC) sensor to an XBee radio to sense the concentration of organic solvents in the air in your shop. All too often, when you're working in the shop by yourself, you become insensitive to the fumes of the chemicals you're working with. This project is an attempt to remedy that issue.

The sensor values are sent to two other radios: one is attached to an XPort, which is connected to the Internet. From there, a PHP script reads the data and stores it in a web document. The other radio is attached to a cymbal-playing toy monkey elsewhere in the house that makes an unholy racket when the organic solvent levels in the shop get high. That way, the rest of the family will know immediately if your shop is toxic. If you don't share my love of monkeys, anything that can be switched on from a transistor can be controlled by this circuit. Figure 7-4 shows the network for this project. And Figure 7-3 shows the completed elements of the project.

WARNING: This project is designed for demonstration purposes only. The sensor circuit hasn't been calibrated. It won't save your life; it'll just make you a bit more aware of the solvents in your environment. Don't rely on this circuit if you need an accurate measurement of the concentration of organic compounds. Check with Figaro Sensor (www.figarosensor.com) to learn how to build a properly calibrated sensor circuit.

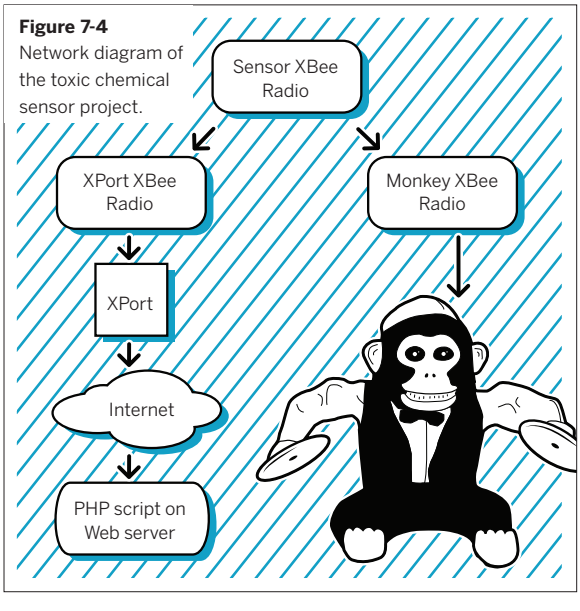


Figure 7-4
Network diagram of the toxic chemical sensor project.

» at right, above

Figure 7-3
The completed toxic sensor system: sensor, monkey, and network connection.

MATERIALS

» **1 USB-to-TTL serial adaptor** SparkFun's (www.sparkfun.com) BOB-00718 from Chapter 2 will do the job. If you use a USB-to-RS-232 adaptor such as a Keyspan or logear dongle, refer to Chapter 2 for the schematics to convert RS-232-to-5V TTL serial. You'll use this for configuring the radios only.

Sensor Circuit

» **1 solderless breadboard** such as Digi-Key (www.digikey.com) part number 438-1045-ND, or Jameco (www.jameco.com) part number 20601
» **1 MaxStream XBee OEM RF module** available from www.maxstream.net or www.gridconnect.com, part number GC-WLM-XB24-A
» **1 5V regulator** The LM7805 series (SparkFun part number COM-00107, Digi-Key part number LM7805CT-ND) work well.
» **1 3.3V regulator** The LD1117-33V (SparkFun part number COM-00526) or the MIC2940A-3.3WT (Digi-Key part number 576-1134-ND) work well.
» **1 2mm breakout board** The XBee modules listed here have pins spaced 2mm apart. To use them on a breadboard, you'll need a breakout board that shifts the spacing to 0.1 inches. You could solder wires on to every pin, or you could make or purchase a printed circuit board that shifts the pins. SparkFun's Breakout Board for XBee Module (BOB-08276) works.
» **2 rows of 0.1-inch header pins** as available from most electronics retailers
» **2 2mm female header rows** Samtec (www.samtec.com) part number MMS-110-01-L-SV. Samtec, like many part makers, supplies free samples of this part in small quantities. SparkFun sells these as part number PRT-08272.
» **1 1µF capacitor** Digi-Key part number P10312-ND
» **1 10µF capacitor** SparkFun part number COM-00523, Digi-Key part number P11212-ND
» **1 Figaro Sensors TGS2620 sensor** for the detection of solvent vapors. You can order this directly from Figaro (www.figarosensor.com or +1-847-832-1701).
» **2 LEDs**
» **1 4.7KΩ resistor**

Internet Connection Circuit

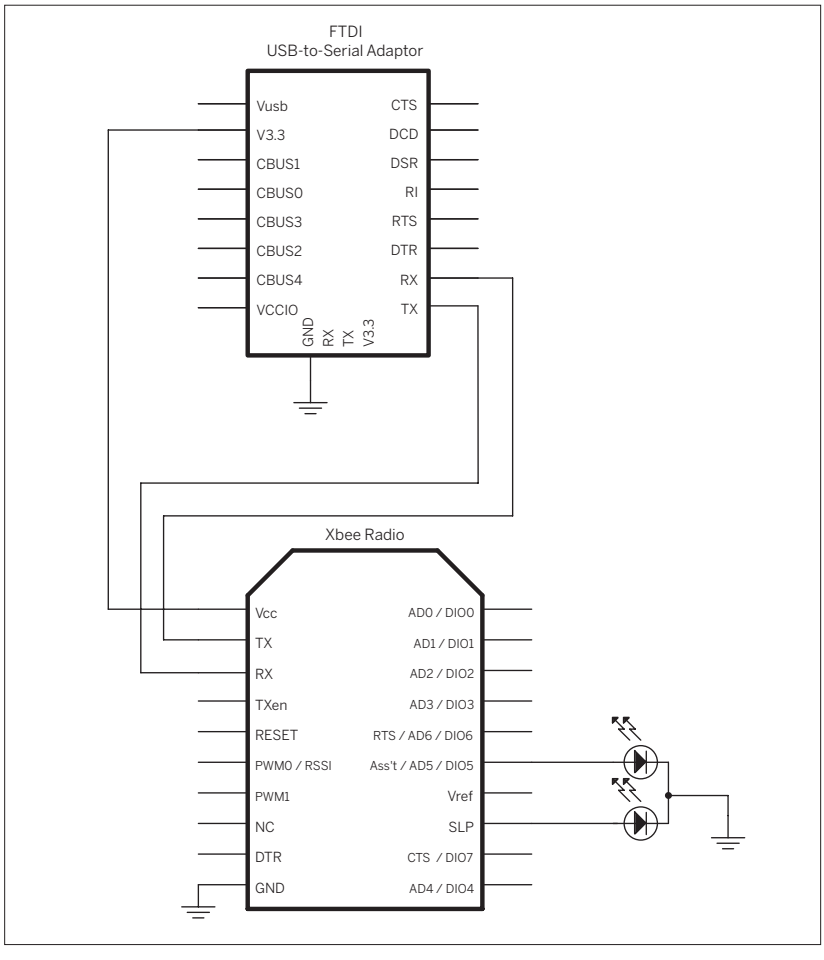
» **1 Lantronix embedded device server** Available from many vendors, including Symmetry Electronics (www.semiconductorstore.com) as part number CO-E1-11AA (Micro) or WM11A0002-01 (WiMicro), or XP1001001-03R (XPort). This example uses an XPort.
» **1 RJ45 breakout board** SparkFun part number BOB-00716 (needed only if you're using an XPort).
» **1 solderless breadboard** such Digi-Key part number 438-1045-ND, or Jameco part number 20601
» **1 MaxStream XBee OEM RF module** MaxStream part number GC-WLM-XB24-A
» **1 3.3V regulator** SparkFun part number COM-00526 or Digi-Key part number 576-1134-ND
» **1 2mm breakout board** SparkFun BOB-08276
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows** Samtec part number MMS-110-01-L-SV. SparkFun part number PRT-08272.
» **1 1µF capacitor** Digi-Key part number P10312-ND
» **1 10µF capacitor** SparkFun part number COM-00523, Digi-Key part number P11212-ND
» **2 LEDs**
» **1 momentary reset switch** SparkFun part number COM-00097, Digi-Key part number SW400-ND.

Cymbal Monkey Circuit

» **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
» **1 MaxStream XBee OEM RF module** MaxStream part number GC-WLM-XB24-A
» **1 cymbal monkey** The one used here is a Charlie Chimp, ordered from the Aboyd Company (aboyd.com), part number ABC 40-1006.

NOTE: If your Monkey uses a 3V power supply (such as 2 D batteries), you won't need the LD1117-33V regulator. Make sure that there's adequate amperage supplied for the radios. If you connect the circuit as shown and the radios behave erratically, the monkey's motor may be drawing all the power. If so, use a separate power supply for the radio circuit.

» **1 2mm breakout board** SparkFun BOB-08276
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows** Samtec part number MMS-110-01-L-SV. SparkFun part number PRT-08272.
» **2 LEDs**
» **1 10K trimmer potentiometer** SparkFun part number COM-00104, Digi-Key part number D4AA14-ND
» **1 TIP120 Darlington NPN transistor**. Digi-Key part number 497-2539-5-ND.
» **1 1N4004 power diode.** Digi-Key part number 1N4004-E3/54GICT-ND.
» **1 1KΩ resistor**
» **1 100µF capacitor.** SparkFun part number COM-00096, Digi-Key part number P10195-ND



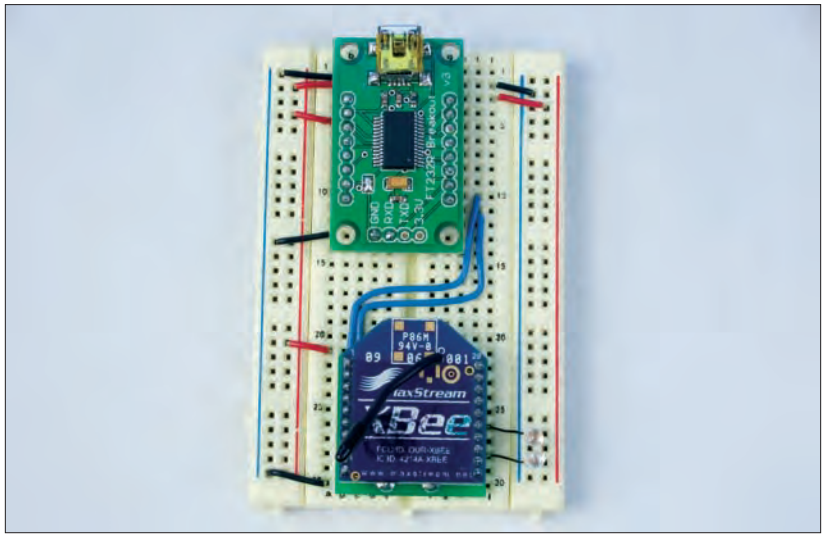


Figure 7-5
XBee radio wired to a serial-to-USB device.

You'll be building three separate circuits for this project, so the parts list is broken down for each one. Most of these items are available at retailers other than the ones listed here, if you can't find them at the places mentioned in the materials list.

Radio Settings

Solder the XBee breakout boards as you did in Chapter 6. Connect one of the radio breakout boards to the FTDI USB-to-serial adaptor, as shown in Figure 7-5. You'll use this circuit for configuring the radios only. Notice that the FTDI module is supplying 3.3V for the radio, so no regulator is needed. If you're using a different USB-to-serial adaptor, you must supply 3.3V for the radio.

You've got three radios: the sensor's radio, the monkey's radio, and the XPort's radio. You know from Chapter 5 that you can configure the radios' addresses, destination addresses, and Personal Area Network (PAN) IDs. In addition, you can also configure some of their behavior. For example, you can configure the digital and analog I/O pins to operate as inputs, outputs, or to turn off. You can also set them to be digital or analog inputs, or digital or pulse width modulation (PWM) outputs. You can even link an output pin's behavior to the signals it receives from another radio.

The sensor radio is the center of this project. You'll configure it to read an analog voltage on its first analog input (AD0, pin 20) and broadcast the value that it reads to all other radios on the same PAN. Its settings are as follows:

- ATMY01 – Sets the sensor radio's source address
- ATDLFFFF – Sets the destination address to broadcast to the whole PAN
- ATID1111 – Sets the Personal Area Network (PAN)
- ATD02 – Sets I/O pin 0 (D0) to act as an analog input
- ATIR64 – Sets the analog input sample rate to 100 milliseconds (0x64 hex)
- ATIT5 – Sets the radio to gather five samples before sending, so it will send every 500 milliseconds (5 samples x 100 milliseconds sample rate = 500 milliseconds)

The monkey radio will listen for messages on the PAN, and if any radio sends it a packet of data with an analog sensor reading formatted the way it expects, it will set the first pulse width modulation output (PWM0) to the value of the received data. In other words, the monkey radio's PWM0 output will be linked to the sensor radio's analog input. Its settings are as follows:

- ATMY02 – Sets the monkey radio's source address
- ATDL01 – Sets the destination address to send only to the sensor radio (address 01). Doesn't really matter, as this radio won't be sending.
- ATID1111 – Sets the Personal Area Network (PAN)
- ATP02 – Sets PWM pin 0 (P0) to act as a PWM output
- ATIU1 – Sets the radio to send any I/O data packets out the serial port. This is used for debugging purposes only; you won't actually attach anything to this radio's serial port in the final project.
- ATIA01 or ATIAFFFF – Sets the radio to set its PWM outputs using any I/O data packets received from address 01 (the sensor radio's address). If you set this parameter to FFFF, the radio sets its PWM outputs using data received from any radio on the PAN.

The XPort radio listens for messages on the PAN and sends them out its serial port to the XBee. This radio's settings are the simplest, as it's doing the least. Its settings are as follows:

- ATMY03 – Sets the XPort radio's source address
- ATDL01 – Sets the destination address to send only to the sensor radio (address 01). Again, doesn't matter, as this radio won't be sending.
- ATID1111 – Sets the Personal Area Network (PAN)
- ATIU1 – Sets the radio to send any I/O data packets out the serial port. This data will go to the attached XPort.

Here's a summary of all of the settings:

Sensor Radio	Monkey Radio	XPort Radio
MY = 01 DL = FFFF ID = 1111 D0 = 2 IR = 64 IT = 5	MY = 02 DL = 01 ID = 1111 P0 = 2 IU = 1 IA = 01 (or FFFF)	MY = 03 DL = 01 ID = 1111 IU = 1

NOTE: If you want to reset your XBee radios to the factory default settings before configuring for this project, send them the command `ATRE\r`

Make sure to save the configuration to each radio's memory by finishing your commands with `WR`. To set the whole configuration of these, you can do it line by line, or all at once. For example, to set the sensor radio, type:

```
+++
```

Then wait for the radio to respond with OK. Then type the following (the 0 in D02 is the number 0):

```
ATMY1, DLFFFF\r
ATID1111, D02, IR64\r
ATIT5, WR\r
```

For the monkey radio, the configuration is:

```
ATMY2, DL1\r
ATID1111, P02\r
ATIU1, IA1, WR\r
```

And for the XPort radio, it's:

```
ATMY3, DL1\r
ATID1111, IU1, WR\r
```

The Circuits

Once you've got the radios configured, set up the circuits for the sensor, the monkey, and the XPort. In all of these circuits, make sure to include the decoupling capacitors on either side of the voltage regulator. The XPort and the XBee radios tend to be unreliable without them.

The Sensor Circuit

The VOC sensor takes a 5V supply voltage, so you need a 5V regulator for it, a 3.3V regulator for the XBee, and a power supply that's at least 9V to supply voltage to the circuit. Figure 7-6 shows the circuit. The VOC sensor should output between 0 and 3.3V under the most likely shop conditions, but test it before you go too much further. Connect and power the circuit, but leave out the wire connecting the sensor's output to the XBee's analog input. Power up the circuit, and let it heat for a minute or two. The circuit takes time to warm up, because there's a heater element in the sensor. Measure the voltage between the sensor's output and ground. You should get about 1 volt if the air is free of VOCs. While still measuring the voltage, take a bottle of something that has an organic solvent (I used hand sanitizer, which has a lot of alcohol in it), and gently waft the fumes over the sensor. Be careful not to breathe it in yourself. You should get something considerably higher — up to 3 volts. If the voltage exceeds 3.3V, change the fixed resistor until you get results in a range below 3.3V, even when the solvent's fumes are high. Once you've got the sensor reading in an acceptable range, connect its output to the XBee's analog input pin, which is pin 20. Make sure to connect the XBee's voltage reference pin (pin 14) to 3.3 volts as well.

NOTE: Make sure to air out your workspace as soon as you've tested the sensor. You don't want to poison yourself making a poison sensor!

To test whether the XBee is reading the sensor correctly, connect its TX pin to the USB-to-serial adaptor's RX pin, its RX pin to the adaptor's TX pin, and connect their ground lines together. Then plug the adaptor into your computer and open a serial connection to it. Type `+++`, and wait for the OK. Then type `ATIS\r` (this command forces the XBee to read the analog inputs and return a series of values). You'll get a reply like this:

```
1
200
3FF
```

Don't worry about what the values are for now, as long as you're getting something. You'll see the actual values as the project develops later.

The Monkey Circuit

To control the monkey, disconnect the monkey's motor from its switch and connect the motor directly to the circuit shown in Figure 7-7. The monkey's battery pack supplies 3V, which is enough for the XBee radio, so you can power the whole radio circuit from the monkey. Connect leads from the battery pack's power and ground to the board. If your monkey runs on a different voltage, make sure to adapt the circuit accordingly, so that your radio circuit is getting at least 3V. Figure 7-8 shows the modifications in the monkey's innards. I used an old telephone cord to wire the monkey to the board, for convenience.

The cymbal monkey circuit takes the variable output that the radio received and turns it into an on-off switch. The PWM output from the XBee radio controls the base of a TIP120 transistor. The monkey itself has a motor built into it, which is controlled by a TIP120 Darlington transistor in this circuit. When the transistor's base goes high, the motor turns on. When it goes low, the motor turns off. The motor has physical inertia, however, so if the length of the pulse is short and the length of the pause between pulses is long, the motor doesn't turn. When the duty cycle of the pulse width (the ratio of the pulse and the pause) is high enough, the motor starts to turn.

To test this circuit, make sure that the sensor radio is working, and turn it on. When the sensor's value is low, the motor should be turned off, and when the sensor reads a high concentration of VOCs, the motor will turn on and the

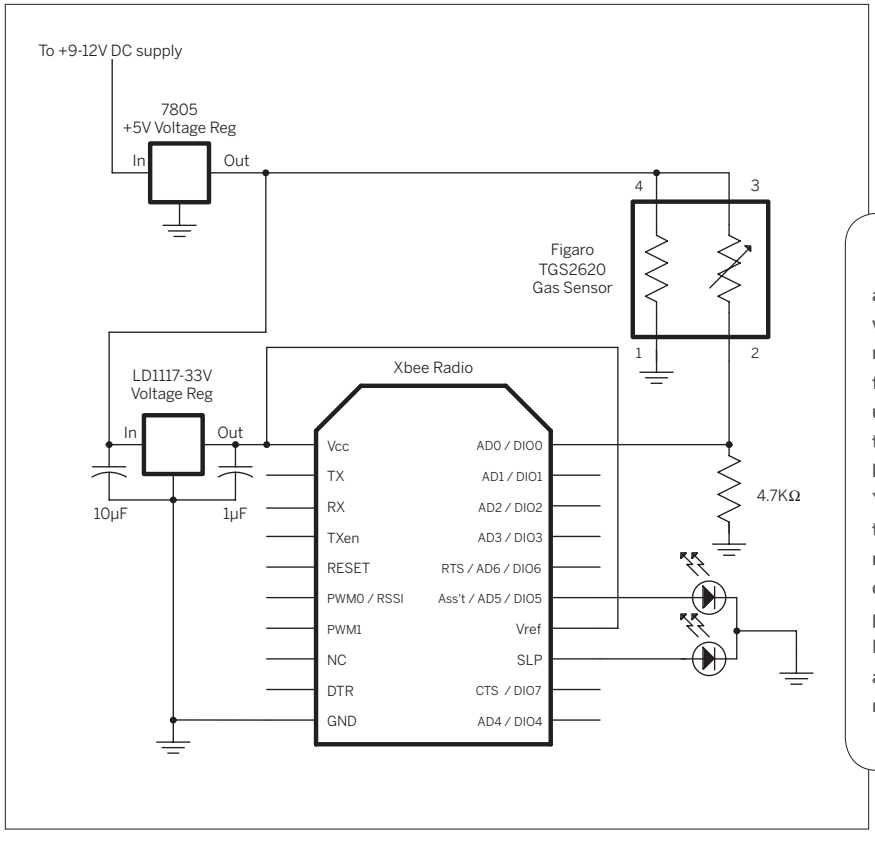
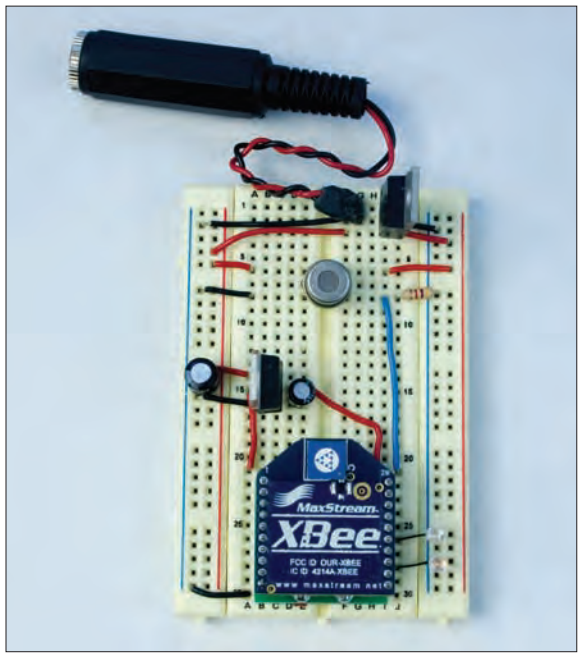
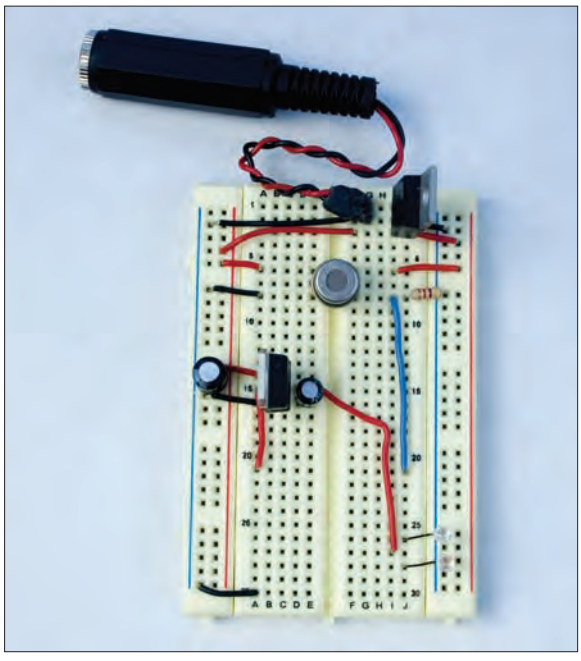
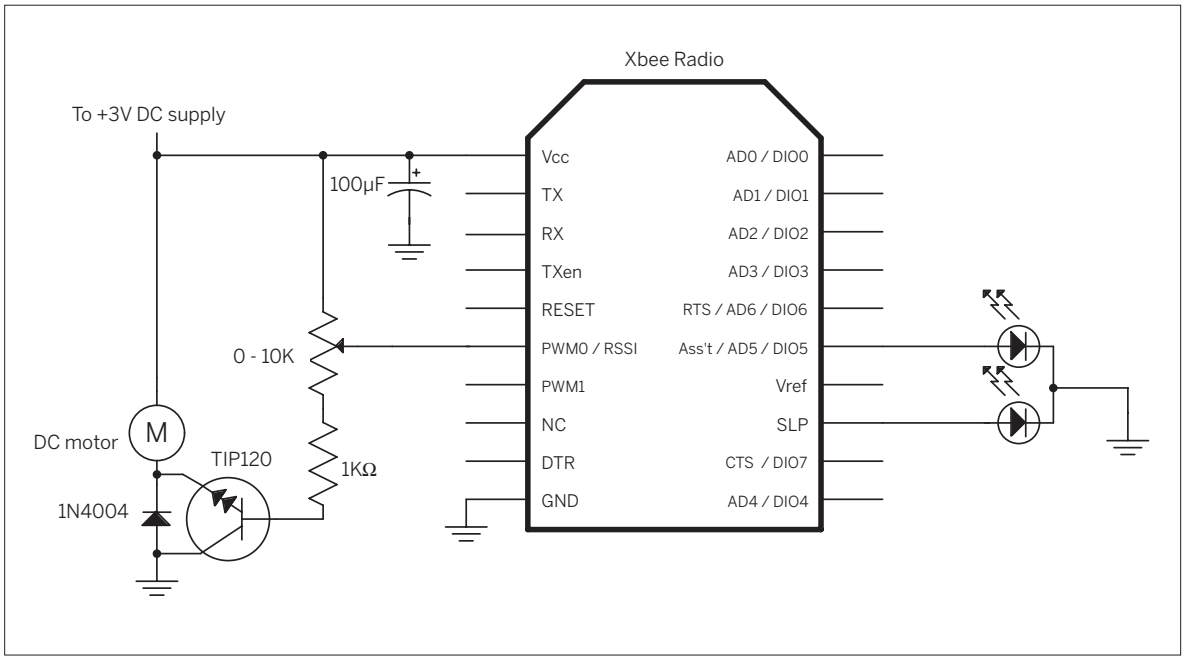


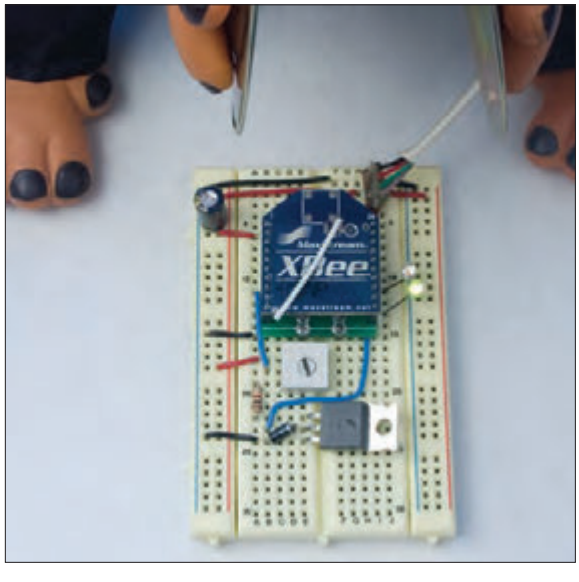
Figure 7-6
XBee radio connected to a Figaro Sensors VOC sensor. The detail photo shows the wiring underneath the XBee radio.

All the projects in this chapter are made using the LD1117-33V 3.3V voltage regulator. The pins on this regulator are configured differently from the pins on the other regulators used in the book. Be sure to check the data sheet for your regulator to be sure you have the pins correct. You should make a habit of checking this because the same catalog number may be supplied as slightly different parts. For example, SparkFun part number COM-00526 (Voltage Regulator—3.3V) can arrive either as an LD1117-33V or an LM7833, which do not have the same pin configuration.









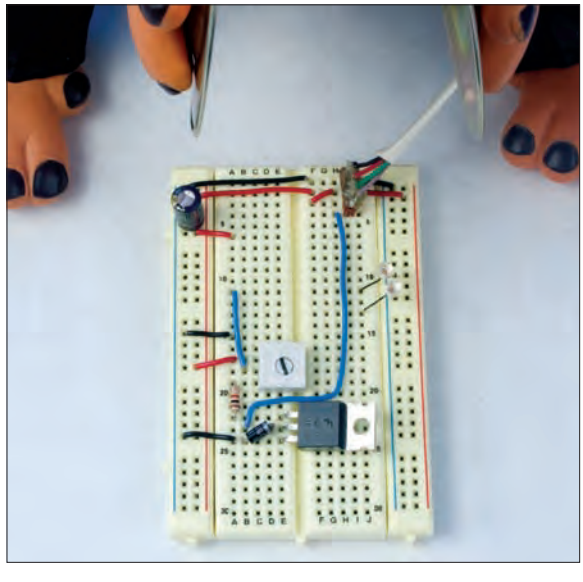
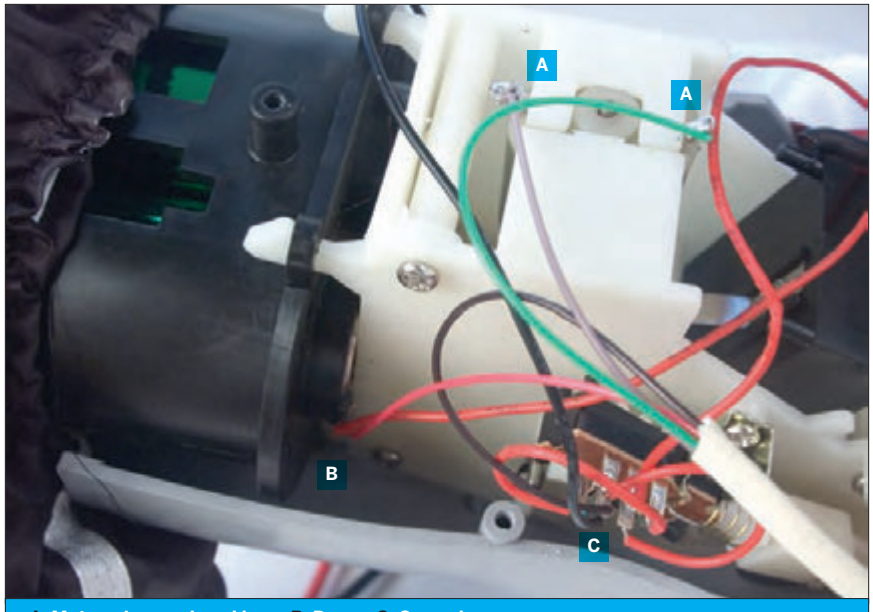


Figure 7-7
XBee radio connected to a Cymbal Monkey. The detail shows the circuit without the XBee, to reveal the wiring underneath.



A. Motor wires replaced here B. Power C. Ground

Figure 7-8
The insides of the monkey, showing the wiring modifications. Solder the power to the breadboard to the positive terminal of the battery. Solder the ground wire to the ground terminal. Cut the existing motor leads, and add new ones that connect to the breadboard.

monkey will play his cymbals in warning. Use the potentiometer to affect the motor's activation threshold. Start with the potentiometer set very high, then slowly turn it down until the motor turns off. At that point, expose the sensor to some alcohol. The motor should turn on again, and should go off when the air around the sensor is clear. If you're unsure that the motor circuit is working correctly, connect an LED from 3V to the collector of the transistor instead of the motor. It should grow brighter when the sensor reading is higher, and dimmer when the sensor reading is lower. The LED has no physical inertia like the motor does, so it turns on at a much lower duty cycle.

The XPort Circuit

The XPort, like the XBee radios, takes a 3.3V supply voltage, so you can run both from a 3.3V regulator, as shown in Figure 7-9. Before you connect this circuit, connect the XPort to your serial port using the circuit in Figure 7-10, and configure it as follows. See "Configuring the Micro" in Chapter 4 for detailed configuration instructions.

Server settings:

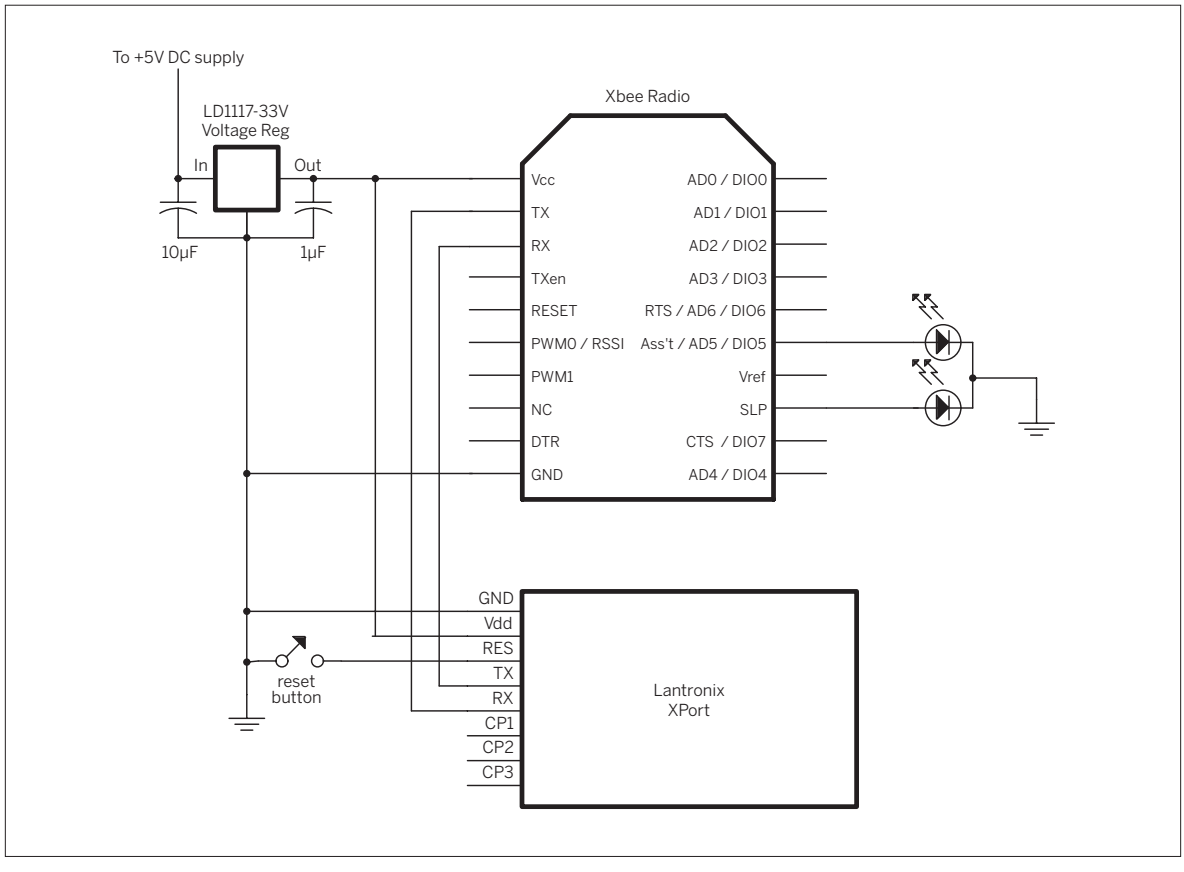
```
IP Address : as appropriate for your network
Set Gateway IP Address Y
Gateway IP addr as appropriate for your network
Netmask: Number of Bits for Host Part  8
```

Channel 1 settings:

```
Baudrate:  9600
I/F Mode: 4C
Flow: 00
Port No: 10001
ConnectMode: D4
Send '+++' in Modem Mode:  N
Auto increment source port:  N
Remote IP Address : 0.0.0.0
Remote Port  0
DisConnMode 00
FlushMode   00
DisConnTime 00:00
SendChar 1  00
SendChar 2  00
```

Figure 7-10 shows an XPort connected to an FTDI USB-to-serial adaptor. This circuit can be used to set the XPort's settings. Even though the FTDI adaptor can supply 3.3V, it's worthwhile to use a regulator, because the XPort performs inconsistently when powered directly from the FTDI adaptor's 3.3V supply. To test this circuit, connect your XPort to your local area network and open a telnet session to its IP address on port 10001. If you're on Linux or Mac OS X, make one change now. When you're logged in, type Control-] and you'll get a telnet prompt like this:

```
telnet>
```



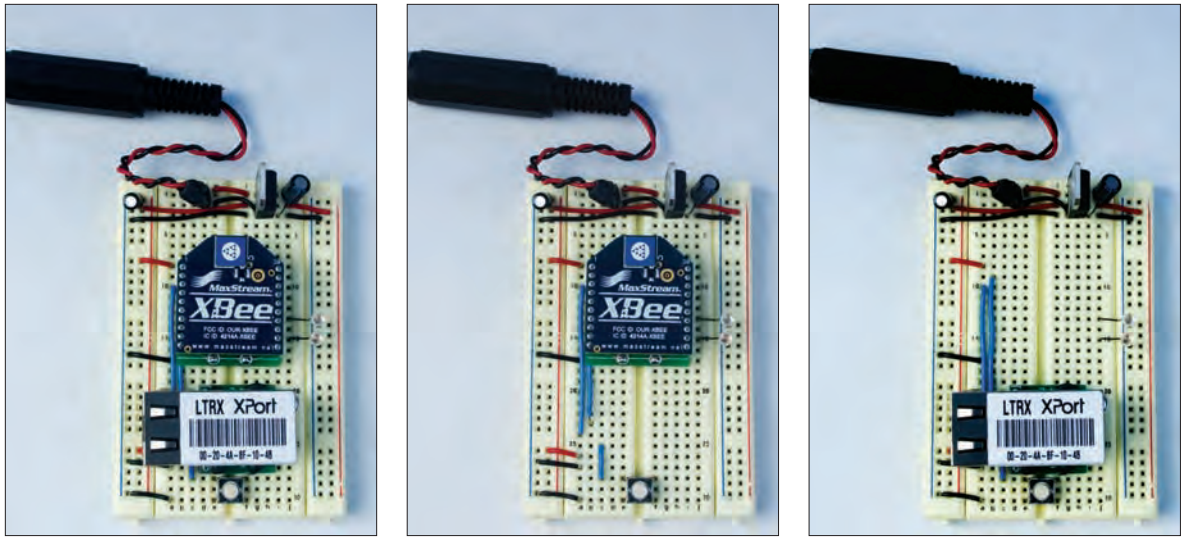
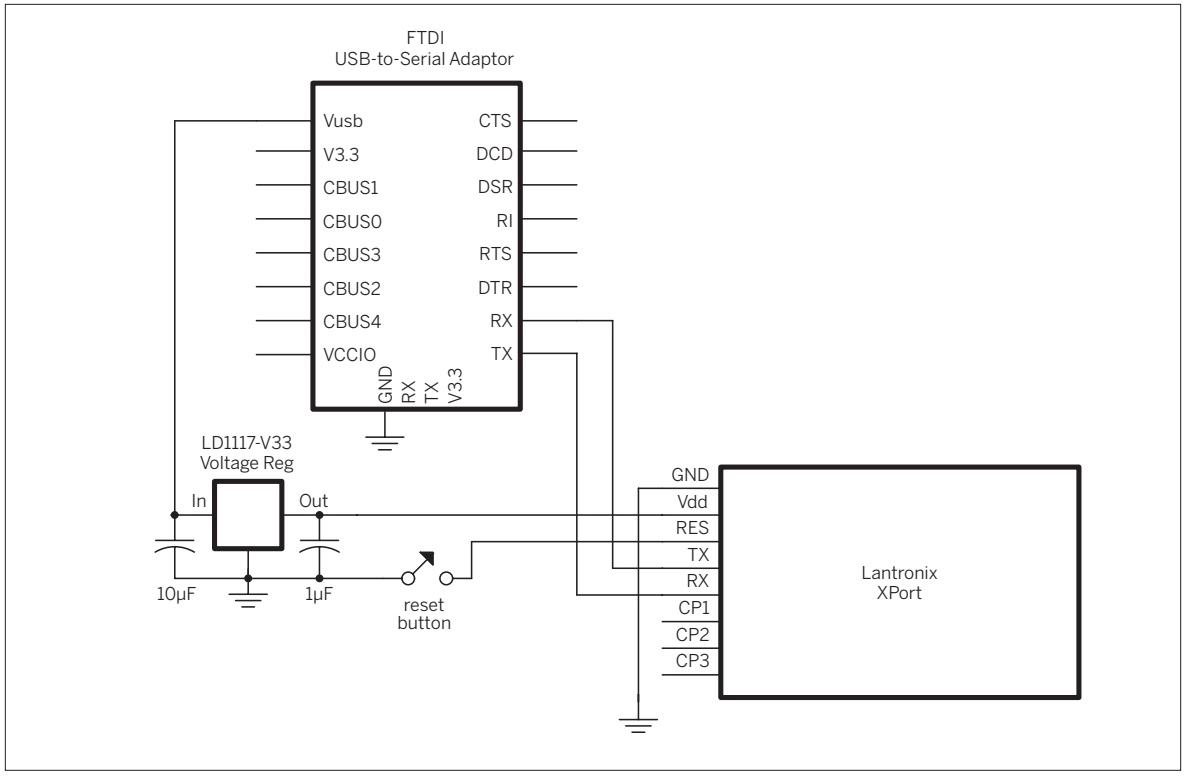
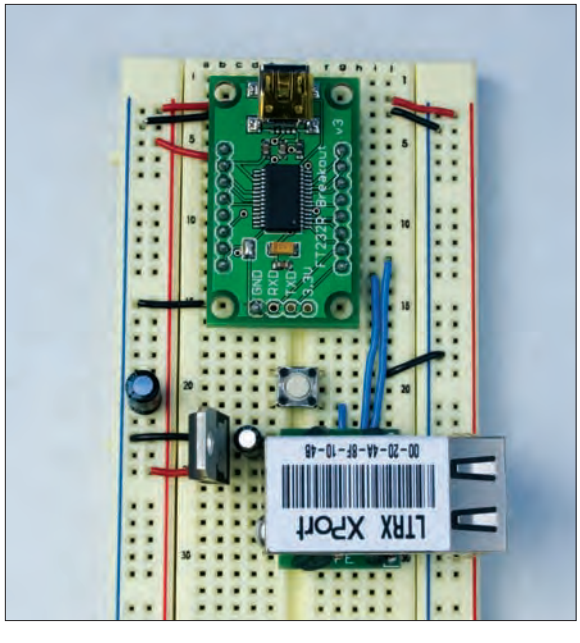


Figure 7-9
XBee radio connected to an XPort. The detail photos show the wiring under the XBee radio and the XPort.





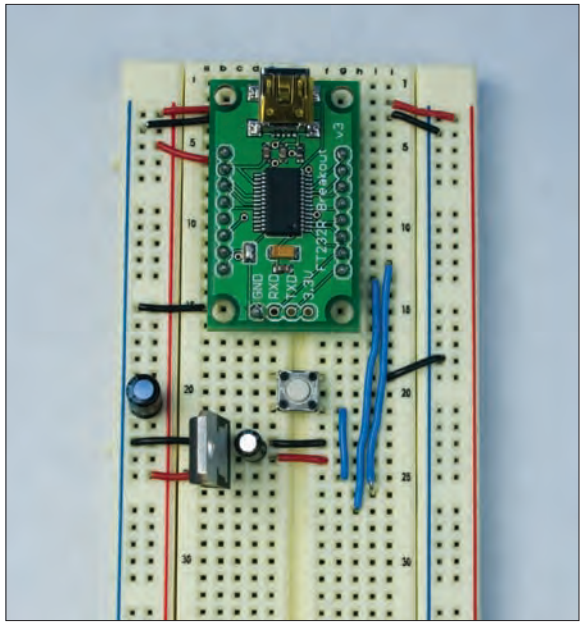


Figure 7-10
XPort connected to a USB-to-serial adaptor. The detail photo shows the wiring under the XPort.

Then type: `mode char\r`

This command puts telnet into character mode, meaning that it will send every character as you type it. You'll need to be in this mode in order to send the +++ string (by default, the Windows telnet program is in the correct mode).

Now try typing XBee commands to read the configuration. Start with the usual +++ and wait for the OK, then ATMY\r, ATDL\r and so forth. If you've wired the circuit correctly, you should get responses from the XBee just as if you were connected to its serial port — because you are!

Once you know that you've got serial data transmitting from the XPort to the XBee, power up the sensor radio while you're still logged in to the XPort. Once it starts up, it should be transmitting regularly, and you should see the data coming into your telnet window. It's being transmitted from the sensor's radio, then to the XPort's radio, on to the XPort, and to your window via the network. Likewise, if you plug the monkey radio in now, you should see the monkey clashing his cymbals as the data changes from the sensor. Now all of your hardware works.

The Server Code

Once you have all of the circuits working, it's time to get the data onto the Web. To do this, you're going to write a PHP script that logs into the XPort, retrieves the data, and displays the results. It will display a summary value telling you the sensor's average reading over 10 packets of data, and it will write that value to a file on the server, so you can see the sensor readings over time if you want.

So far, you've been able to rely on the XBee radios to do their work without having to understand their message protocol. Now it's time to interpret that protocol. The XBee radios format the readings from their analog-to-digital converters into a packet before they transmit. The format of the packet is explained in the MaxStream XBee 802.15.4 user's manual. It works like this:

- Byte 1: 0x7E, the start byte value.
- Byte 2-3: packet size, a 2-byte value. This depends on your other settings.
- Byte 4: API identifier value, a code that says what this response is.
- Byte 5-6: XBee sender's address.
- Byte 7: RSSI, Received Signal Strength Indicator.
- Byte 8: Broadcast options (not used here).
- Byte 9: Number of samples in the packet (you set it to 5 using the IT command shown earlier).
- Byte 10–11: Which I/O channels are currently being used. This example assumes only one analog channel, AD0, and no digital channels are in use.
- Byte 12–21: 10-bit values, each ADC sample from the sender. Set this to 5 using the IT command.

Because every packet starts with a constant value, 0x7E (that's decimal value 126), you can start your PHP program looking for that value.

If your PHP script is running on a different network than your XPort (such as on a web hosting company's server), you'll need to find a way to put your XPort on the Internet. See "Making a Device Visible to the Internet When It Has a Private IP Address," later in this chapter, for more information.

Connect It

The following program opens a socket to the XPort, reads bytes and puts them in an array until it's seen the value 0x7E ten times, then closes the socket. In other words, it attempts to read ten packets. Save this file to your server as **toxic_report.php**.

NOTE: In the following code, set $ip to the IP address of your XPort.

```
<?php
/*  toxic_report.php
    Socket connection string reader
    Language: PHP
    This program opens a socket connection to an XPort
    and reads bytes from the socket.
*/

// Global variables.  These can be used by any of the script's functions:
global $ip, $port, $packetsToRead, $timeStamp, $messageString;
$ip = "192.168.1.236";     // IP adddress to connect to.
                           // Change this to your XPort's IP address.
$port = 10001;             // port number of IP
```

»

Continued from opposite page.

```
$packetsToRead = 10;      // total number of packets to read
$packetCounter = 0;       // counter for packets when you're reading them
$bytes = array();         // array for bytes when you're reading them

// open a socket to the XPort:
$mySocket = fsockopen ($ip, $port, $errorno, $errorstr, 30);
if (!$mySocket) {
    //if the socket didn't open, return an error message
    return "Error $errorno: $errorstr<br>";
} else {
    // if the socket exists, read packets until you reach $packetsToRead:
    while ($packetCounter < $packetsToRead) {
        // read a character from the socket connection,
        // and convert it to a numeric value using ord(),
        $char = ord(fgetc($mySocket));

        // if you got a header byte, deal with the last array
        // of bytes first:
        if ($char == 0x7E) {
            // increment the packet counter:
            $packetCounter++;
        }
        // push the current byte onto the end of the byte array:
        array_push($bytes, $char);
    }
    // iterate over the array of bytes and print them out:
    foreach ($bytes as $thisByte) {
        // if the current byte = 0x7E, it starts a new packet;
        // print a break first, so you see each packet on a new line
        // in the browser:
        if ($thisByte == 0x7E) {
            echo "<br>";
        }
        echo "$thisByte ";
    }
    // close the socket:
    fclose ($mySocket);
}
?>
```

When you save this to your server and open the script in a browser, you'll get a result like this:

NOTE: If you're getting weird numbers, power off the sensor and Lantronix circuits, wait about a minute, and power them back on. Devices sometimes get confused.

```
201 1 201 1 200 1 197 91
126 0 18 131 0 1 43 0 5 2 0 1 197 1 196 1 198 1 198 1 197 106
126 0 18 131 0 1 43 0 5 2 0 1 197 1 193 1 193 1 192 1 192 125
126 0 18 131 0 1 44 0 5 2 0 1 194 1 194 1 193 1 190 1 190 130
126 0 18 131 0 1 43 0 5 2 0 1 189 1 189 1 191 1 190 1 190 143
126 0 18 131 0 1 43 0 5 2 0 1 190 1 186 1 186 1 186 1 188 156
126 0 18 131 0 1 43 0 5 2 0 1 187 1 187 1 186 1 183 1 183 166
126 0 18 131 0 1 43 0 5 2 0 1 182 1 182 1 184 1 183 1 183 178
126 0 18 131 0 1 43 0 5 2 0 1 181 1 180 1 179 1 179 1 182 191
126 0 18 131 0 1 43 0 5 2 0 1 181 1 181 1 180 1 178 1 177 195
126
```

Refine It It'd be handy to have each packet in its own array. To make that happen, add a new global array variable called $packets before you open the socket:

```
$packets = array();   // array to hold the arrays of bytes
```

» Then change the while() loop, adding code to push the byte array on to the packet array when a new header byte arrives, and then to empty the byte array to receive a new packet:

```
   while ($packetCounter < $packetsToRead) {
        // read a character from the socket connection,
        // and convert it to a numeric value using ord(),
        $char = ord(fgetc($mySocket));

        // if you got a header byte, deal with the last array
        // of bytes first:
        if ($char == 0x7E) {
            // push the last byte array onto the end of the packet array:
            array_push($packets, $bytes);
            // clear the byte array:
            $bytes = array();
            // increment the packet counter:
            $packetCounter++;
        }
        // push the current byte onto the end of the byte array:
        array_push($bytes, $char);
    }
```

Most of the lines have 22 bytes, corresponding to the packet format described earlier. You may wonder why the first line of the output shown above didn't have a full complement of bytes. It's simply because there's no way to know what byte the Lantronix XBee radio is receiving when the PHP script connects to it.

You want only full packets, so add a function to parse the array and read only full packets. This function also extracts the XBee address of the sender, and averages the ADC readings in bytes 12 to 21 of the packet.

» Add this at the end of the program (but before the closing ?>):

```
/*-------------------------------------*/

function parsePacket($whichPacket) {
    $adcStart = 11;                 // ADC reading starts at 12th byte
    $numSamples = $whichPacket[8];  // number of samples in the packet
    $total = 0;                     // sum of ADC readings for averaging

    // if you got all the bytes, find the average ADC reading:
    if( count($whichPacket) == 22) {
        // read the address -- it's a two-byte value, so you
        // add the two bytes as follows:
        $address = $whichPacket[5] + $whichPacket[4] * 256;

        // read $numSamples 10-bit analog values, two at a time,
        // because each reading is two bytes long:
        for ($i = 0; $i < $numSamples * 2;  $i=$i+2) {
            // 10-bit value = high byte * 256 + low byte:
            $thisSample = ($whichPacket[$i + $adcStart] * 256) +
              $whichPacket[($i + 1) + $adcStart];
            // add the result to the total for averaging later:
            $total = $total + $thisSample;
```

»

Continued from opposite page.

```
        }
        // average the result:
        $average = $total / $numSamples;
        return $average;
    } else {
        return -1;
    }
}
```

To call this routine, add another variable at the beginning of the program called $totalAverage:

```
$totalAverage = 0;  // average of sensor readings in each packet
```

Next, replace the for() loop that iterates over the byte array and prints the bytes with the following one that iterates over the array of packets:

```
    // iterate over the array of arrays and print them out:
    foreach ($packets as $thisPacket) {
        $packetAverage = parsePacket($thisPacket);
        echo "Average sensor reading in this packet: $packetAverage<br>";
    }
```

You should get a result like this:

```
Average sensor reading in this packet: -1
Average sensor reading in this packet: 368.2
Average sensor reading in this packet: 368.8
Average sensor reading in this packet: 368.2
Average sensor reading in this packet: 368
Average sensor reading in this packet: 368.6
Average sensor reading in this packet: 367.6
Average sensor reading in this packet: 366.6
Average sensor reading in this packet: 368.2
Average sensor reading in this packet: 367.2
```

When there's not a complete packet, parsePacket() returns –1, so you know if you've got a good reading or not.

Now you've got ten packets of data coming in, and you're averaging the sensor readings from each packet. To display an overall reading, you need a function to average the results from all of the packets. Here's a function to do it:

```
/*-------------------------------------*/

function averagePackets($whichArray) {
    $packetAverage = 0;        // average of all the sensor readings
    $validReadings = 0;        // number of valid readings
    $readingsTotal = 0;        // total of all readings, for averaging

  // iterate over the packet array:
    foreach ($whichArray as $thisPacket) {
        // parse each packet to get the average sensor reading:
        $thisSensorReading = parsePacket($thisPacket);
```

Continued from previous page.

```
        if ($thisSensorReading > 0 && $thisSensorReading < 1023) {
            // if the sensor reading is valid, add it to the total:
            $readingsTotal = $readingsTotal + $thisSensorReading;
            // increment the total number of valid readings:
            $validReadings++;
        }
    }
    if ($validReadings > 0) {
        // round the packet average to 2 decimal points:
        $packetAverage = round($readingsTotal / $validReadings, 2);
        return $packetAverage;
    } else {
        return -1;
    }

}
```

This function replaces the for() loop that you just added to print the results, so you can take that loop out now, and just write:

```
    // average the readings from all the packets to get a final
    // sensor reading:
    $totalAverage = averagePackets($packets);

    echo "Sensor Reading :" . $totalAverage;
```

Now you've got a web page that gives you a snapshot of the air quality in your shop. The page should read something like this:

Sensor Reading :349.98

But Wait! That's Not All!

Perhaps you'd like to set up several sensors, each attached to its own XBee/XPort combination, or perhaps you'd like to save the sensor data to a file so that you can see the levels over time. To do that, you need to add a form so you can change the variables from the Web, and you need to add a routine to write the sensor reading to a file. While you're at it, you need to check the time of the reading so that you can add a time stamp to the saved readings.

First, add the form at the end of the script. This is just HTML, and it comes at the very end, outside your closing PHP ?> tag:

```
<html>
    <head>
    </head>
    <body>
        <h2>
        <?=$messageString?>
        </h2>
        <hr>
        <form name="message" method="post" action="toxic_report.php">
            IP Address: <input type="text" name="ip" value="<?=$ip?>"
```

Continued from opposite page.

```
                    size="15" maxlength="15">
          Port: <input type="text" name="port" value="<?=$port?>"
                    size="5" maxlength="5">  <br>
          Number of readings to take: <input type="text"
            name="packetsToRead" value="<?=$packetsToRead?>" size="6">
          <input type="submit" value="Send It">
        </form>
    </body>
</html>
```

You can see a few PHP tags in the HTML. These add the PHP variables to the HTML. Next, you need some code in the script to read the form. Add the following near the beginning of the program, after the global variables, but before you open the socket to the XPort:

```
//if a filled textbox was submitted, get the values:
if ((isset($_POST["ip"])) &&
    (isset($_POST["port"])) &&
    (isset($_POST["packetsToRead"]))) {
        $ip = $_POST["ip"];
        $port = $_POST["port"];
        $packetsToRead = $_POST["packetsToRead"];
}
```

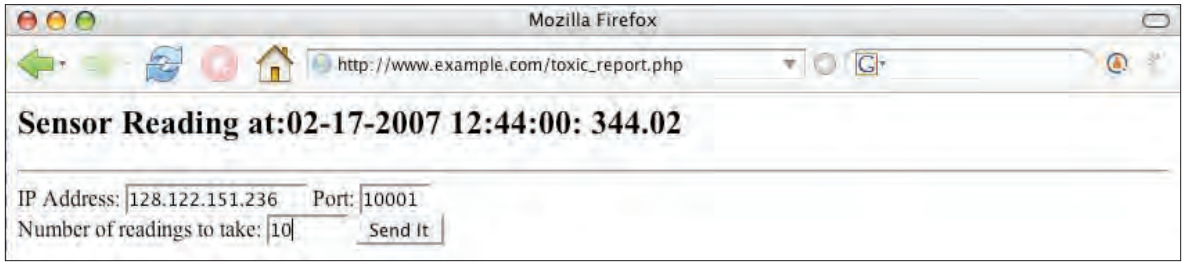
Add two new variables to the beginning of the program, one to get the time, and one to make the HTML code simpler to write:

```
// $messageString is used to return messages for printing in the HTML:
$messageString = "No Sensor Reading Taken";
// Get the time and date:
$timeStamp =  $date = date("m-d-Y H:i:s");
```

To use these two variables, change the lines that average and print the sensor reading like so. You'll get a page that looks like Figure 7-11.

```
    // average the readings from all the packets to get a
    // final sensor reading:
    $totalAverage = averagePackets($packets);

    // update the message for the HTML:
    $messageString =
      "Sensor Reading at: ". $timeStamp . ": " . $totalAverage;
```

Figure 7-11
The final web page of the toxic report.



Now you can add some code to save the sensor data to a server. Add one more function to do this:

```
/*-------------------------------------*/

function writeToFile($whichReading) {
    global $timeStamp, $messageString;

    // combine the reading and the timestamp:
    $logData = "$timeStamp $whichReading\n";
    $myFile = "datalog.txt";    // name of the file to write to:

    // check to see whether the file exists and is writable:
    if (is_writable($myFile)) {
        // try to write to the file:
       if (!($fh = fopen($myFile, "a"))) {
            $messageString = "Couldn't open file $myFile";
        } else {
            // if you could open the file but not write to it, say so:
            if (!fwrite($fh, $logData)) {
                $messageString = "Couldn't write to $myFile";
            }
        }
    } else {
        //if it's not writeable:
       $messageString = "The file $myFile is not writable";
    }
}
```

Finally, just before the socket closing at the end of the main script, add the following to call this function:

You can find a complete listing in Appendix C.

```
    // if you got a good reading, write it to the datalog file:
    if ($totalAverage > 0) {
        writeToFile($totalAverage);
    }
```

To enable the PHP script to write to the data log file, you'll need to create the file on your server first. Make a blank text file called **datalog.txt** in the same directory as the PHP script (at the Linux or Mac OS X terminal, you can use the command touch datalog.txt). Change its permissions so that it's readable and writable by others. From the command line of a Linux or Mac OS X system, you'd type:

```
chmod o+rw datalog.txt
```

If you're creating the file using a GUI-based program, get info on the file and set the permissions that way. Figure 7-12 shows the Get Info window from BBEdit, which is similar to many others. Once you've made this file and viewed the web page a few times, open the datalog.txt file. You'll see something like this:

```
02-17-2007 13:01:38 338.98
02-17-2007 13:01:44 338.93
02-17-2007 13:09:57 338.31
02-17-2007 13:10:03 338.2
02-17-2007 13:10:09 338.62
```

Now that you've got the data, you can work with it in interesting ways. For example, you could write another PHP script to read the sensor regularly and graph the results. X

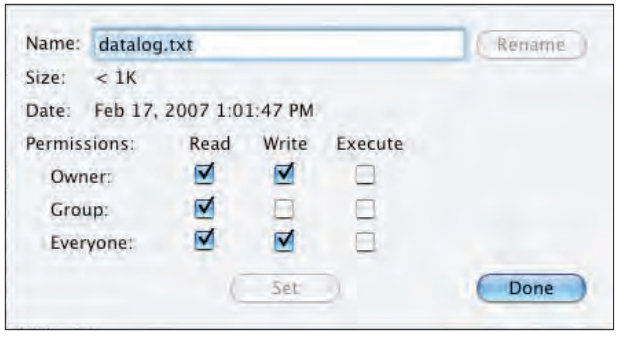
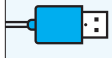


Figure 7-12
Setting the read-write permissions for a file from a GUI-based program.



Making a Private IP Device Visible to the Internet

Up until now, all of the Internet-related projects in this book have either worked only on a local subnet, or only sent data outbound and waited for a reply. This is the first project in which your Lantronix device needs to be visible to the Net at large (or at least to the PHP script). If it's connected to your home router and has a private IP address, however, that won't be the case. To get around this, you need to arrange for one of your router's ports to forward incoming messages and connection requests to your XPort.

To do this, open your router's administrator interface and look for controls for "port forwarding" or "port mapping."

The interface will vary depending on the make and model of your router, but the settings generally go by one of these names. It's easiest if the forwarded port on the router is the same as the open port on the XPort, so configure it so that port 10001 on your router connects to port 10001 on the XPort. Once you've done this, any incoming requests to connect to your router's public IP address on that port will be forwarded to the XPort's private IP address on the same port. Figures 7-13 and 7-14 show the settings on a Linksys wireless router and an Apple Airport Express router. On the Linksys router, Port Forwarding can be found under the Advanced tab.

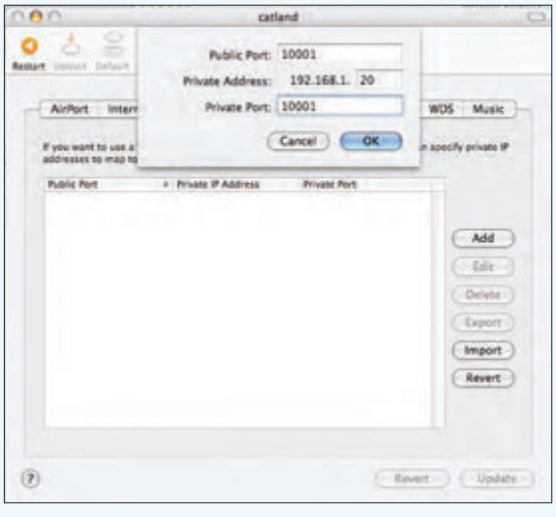


Figure 7-13. Port mapping tab on an Apple Airport Express router.

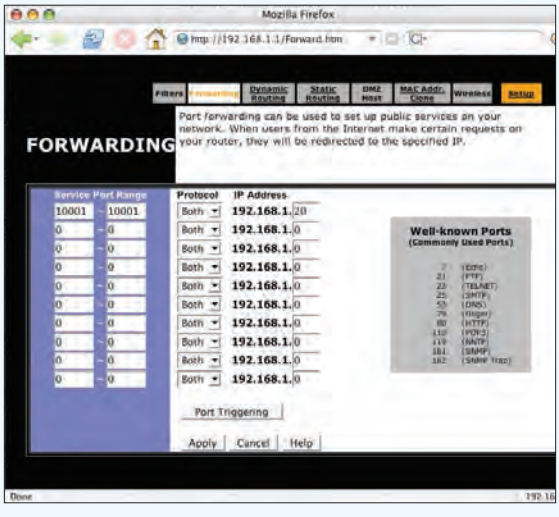


Figure 7-14. Port forwarding on a Linksys wireless router.

Directed Messages

The more common way to use sessionless protocols is to send *directed messages* to the object to which you want to speak. You saw this in action already in Chapter 6, when you programmed your microcontrollers to speak to each other using the XBee radios. Each radio had a source address (which you read and set using the *ATMY* command) and a destination address (which you read and set using the *ATDL* command). One radio's destination was another's source, and vice versa. Though there were only two radios in that example, you could have included many more radios, and decided which one to address by changing the destination address on the fly.

Sending UDP Datagrams To and From a Lantronix Device

So far, you've used the Lantronix devices to communicate via TCP, but they can also send and receive UDP packets. To do this, you have to set the connectMode appropriately (see "Configuring the Micro" in Chapter 4 for configuration instructions) and set an address to which the datagrams will be sent. You can also control when datagrams are sent; for example, the default is 12 milliseconds after a serial byte is received, but you can change the time delay, or you can set the device to send after it receives a specific string of characters.

To send directed UDP packets, you have to set the connect Mode to 0xCC, which sets your device to accept any incoming UDP packets. It also allows you to send UDP packets to the address and port number that you set for the remote IP address. Once you've set the connectMode to 0xCC, set the Datagram Type to 01. With these settings, your XPort, Micro, or WiPort will send only to the remote IP specified in your configuration. Here's a summary of the appropriate settings:

```
Baudrate: 9600
I/F Mode: 4C
Flow: 00
Port No: 10001
ConnectMode: CC
Datagram Type: 01
Remote IP Address : fill in the address of your personal computer
Remote Port : 10002
Pack Cntrl : 00
SendChar 1 : 00
```

Whatever bytes you send into the device's serial port are sent out as a datagram. There is a limit on the delay between bytes, but as long as you're sending all your bytes at once, you should be fine. If you have a problem, you can control it by changing the way datagrams are sent. You can set the device to send only after it gets a specific two-byte sequence, like \n\r. To do this, change the channel 1 configuration as follows:

```
packControl : 30
SendChar 1 : 0A (that's 10, or linefeed in ASCII)
SendChar 2 : 0D (13, or carriage return in ASCII)
```

Then make sure all of your messages end in a linefeed and a carriage return, and they'll always get through intact. For more on this, see the XPort, WiPort, or Micro's User's Guide, available from www.lantronix.com.

The next sketch demonstrates how UDP works using Processing and an XPort. In it, the sketch and the XPort form a loop, as shown in Figure 7-15: when you type a u, the sketch sends UDP packets to the XPort's Ethernet connection, and listens on its serial port.

When you type an s, it sends via the XPort's serial port, and listens for UDP messages sent by the XPort.

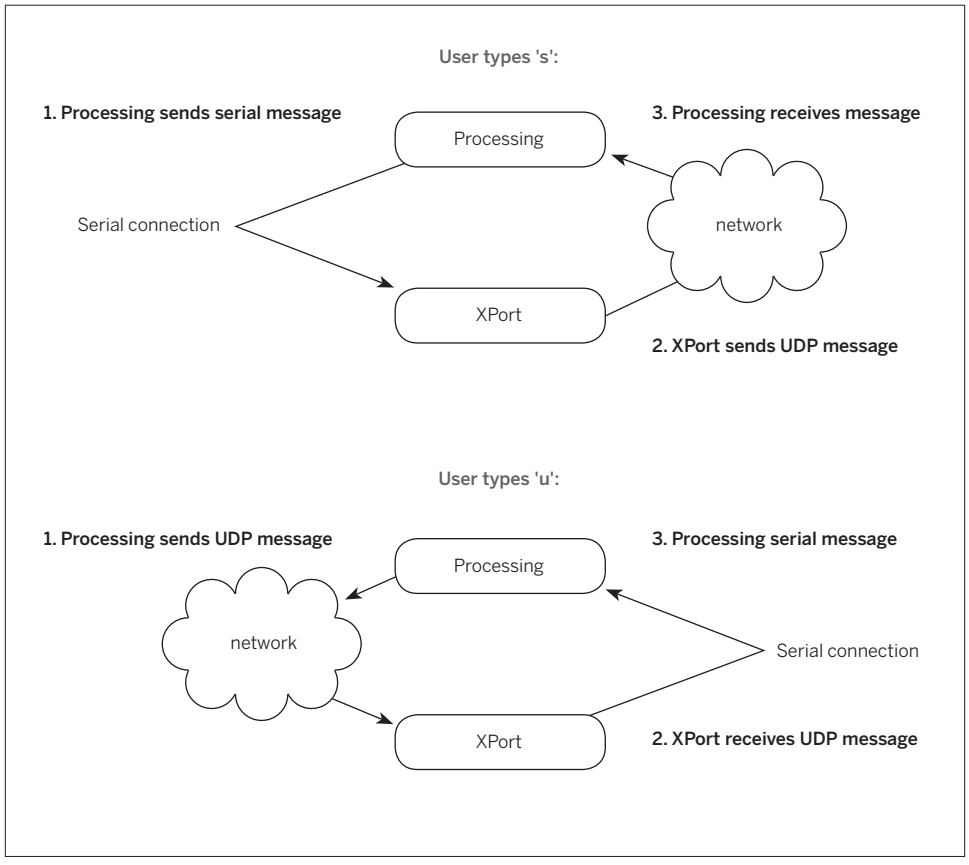


Figure 7-15
Message flow in the Processing UDP testing program.

Test It Here is the Processing sketch to test UDP sending and receiving. To use it, connect an XPort, WiPort, or Micro to a USB-to-serial adaptor as shown in Figure 7-10. Then run this program:

```
/*
 Lantronix UDP Tester
 language: Processing

 Sends and receives UDP messages from  Lantronix
 serial-to-Ethernet devices.

 Sends a serial message to a Lantronix device connected to the
 serial port when you type "s".

 Sends a UDP message to the Lantronix device when you type "u".

 Listens for both UDP and serial messages and prints them out.

 */

// import UDP library
import hypermedia.net.*;
// import serial library:
import processing.serial.*;
```

Continued from previous page.

```
UDP udp;                          // define the UDP object
int queryPort = 10002;            // the port number for the device query
Serial myPort;

String xportIP = "192.168.1.20";   // fill in your XPort's IP here
int xportPort = 10001;            // the XPort's receive port
String inString = "";             // incoming serial string

void setup() {
  // create a new connection to listen for
  // UDP datagrams on query port;
  udp = new UDP(this, queryPort);

  // listen for incoming packets:
  udp.listen( true );

  println(Serial.list());
  // make sure the serial port chosen here is the one attached
  // to your XPort:
  myPort = new Serial(this, Serial.list()[0], 9600);
}

//process events
void draw() {
  // a nice blue background:
  background(0,0,255);
}

/*
 send messages when s or u key is pressed:
 */
void keyPressed() {
  switch (key) {
  case 'u':
    udp.send("Hello UDP!\r\n", xportIP, xportPort);
    break;
  case 's':
    String messageString = "Hello Serial!";
    for (int c = 0; c < messageString.length(); c++) {
      myPort.write(messageString.charAt(c));
    }
    break;
  }
}
```

You'll need to change this number.

Continued from opposite page.

```
/*
  listen for UDP responses
 */
void receive( byte[] data, String ip, int port ) {
  String inString = new String(data); // incoming data converted to string
  println( "received "+inString +"  from "+ip+" on port "+port );

  // print two blank lines to separate messages from multiple responders:
  print("\n\n");
}

/*
  listen for serial responses
 */
void serialEvent(Serial myPort) {
  // read any incoming bytes from the serial port and print them:
  char inChar = char(myPort.read());

  // if you get a linefeed, the string is ended; print it:
  if (inChar == '\n') {
    println("received " + inString + "  in the serial port\r\n");
    // empty the string for the next message:
    inString = "";
  }
  else {
    // add the latest byte to inString:
    inString += inChar;
  }
}
```

Project 13

Relaying Solar Cell Data Wirelessly

In this project, you'll relay data from a solar cell via two XBee radios and an XPort to a Processing sketch that graphs the result. This project is almost identical to the previous one in terms of hardware, but instead of using broadcast messages, you'll relay the data from the first to the second to the third using directed messages. In addition, the XPort uses directed UDP datagrams to send messages to the Processing program.

This project comes from Gilad Lotan and Angela Pablo (Figure 7-16), students at the Interactive Telecommunications Program (ITP) at New York University. The ITP is on the fourth floor of a twelve-story building in Manhattan, and maintains an 80-watt solar panel on the roof of the building. The students wanted to watch how much useful energy the cell receives each day. Because it's used to charge a 12-volt battery, it's useful only when the output voltage is higher than 12V. In order to monitor the cell's output voltage on the fourth floor, Gilad and Angela (advised by a third student, Robert Faludi), arranged three XBee radios to relay the signal down the building's stairwell from the roof to the fourth floor. From there, the data went over the local network via an XPort, and on to an SQL database. This example, based on their work, uses a smaller solar cell from SparkFun and a Processing program to graph the data instead of an SQL database.

There are three radios in this project: one attached to the solar cell, one relay radio standing on its own, and one attached to the XPort. Figure 7-17 shows the network.

Radio Settings

The radio settings are similar to the settings for the previous project. The only difference is in the destination addresses. You won't be using broadcast addresses this time. Instead, the solar cell radio (address = 1) will send to the relay radio (address = 2), and that radio will send to the XPort radio (address = 3). Instead of forming a broadcast network, they form a chain, extending the distance the message travels. Their settings are shown in the table above. Here are the command strings to set them. For the solar cell radio:

Sensor Radio	Relay Radio	XPort Radio
MY = 01 DL = 02 ID = 1111 D0 = 2 IR = 0x64 IT = 5	MY = 02 DL = 03 ID = 1111 P0 = 2 IU = 1 IA = 01 (or 0xFFFF)	MY = 03 DL = 01 ID = 1111 IU = 1

```
ATMY1, DL02\r
ATID1111, D02, IR64\r
ATIT5, WR\r
```

For the relay radio:

```
ATMY2, DL03\r
ATID1111, P02\r
ATIU1, IA1, WR\r
```

And for the XPort radio:

```
ATMY3, DL01\r
ATID1111, IU1, WR\r
```

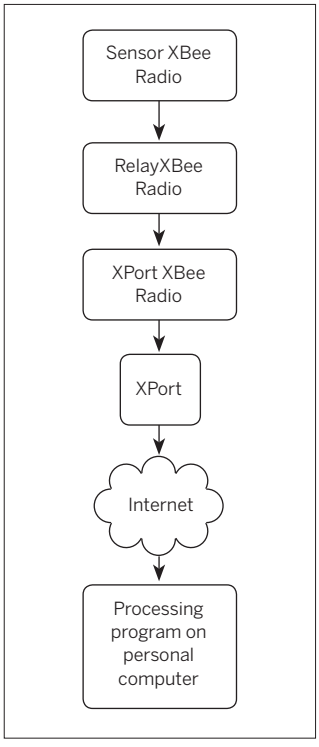
The Circuits

The solar cell circuit runs off the solar cell itself, because the cell can produce the voltage and amperage in daylight needed to power the radio. The LD1117-33V regulator can take up to 15V input, and the solar panel's maximum output is 12V, so you're safe there. The MAX8212 IC is a voltage trigger. When the input voltage on its threshold pin goes above a level determined by the resistors attached to the circuit, the output pin switches from high to low. This change turns on the 2N3906 transistor. The transistor then allows voltage and current from the solar cell to power the regulator. When the solar cell isn't producing enough voltage, the radio will simply turn off. It's okay if the radio doesn't transmit when the cell is dark, because there's nothing worth reporting then. The two resistors attached to the XBee's AD0 pin form a voltage divider that drops the voltage from the solar cell proportionally to something within the 3.3V range of the radio's analog-to-digital converter. The 4700µF capacitors store the charge

▲ **Figure 7-16**
ITP students Angela Pablo and Gilad Lotan with the solar battery pack and XBee monitor radio.

Mesh Networking

The XBee radios can be configured as a mesh network, using the ZigBee protocol. In a mesh network, some radios function as routers, similar to how the relay radio works in this project. Routers can not only relay messages, but can also store and forward them when the radios at the end node are not on. This provides the whole network with net power saving, as the end nodes can be turned off most of the time. At the time of this writing, MaxStream's implementation of the ZigBee protocol was not fully finished, so this simpler solution was used. MaxStream recently announced a second generation of the XBee radios, which uses a different chipset and implements the ZigBee protocol better than the original did. For more information, see www.maxstream.net.



◀ **Figure 7-17**
Network diagram for the solar project.

from the solar cell like batteries, to keep the radio's supply constant. Figure 7-18 shows the circuit.

The XPort radio circuit is identical to the one used in the last project. Build it as shown in Figure 7-9.

MATERIALS

» **1 USB-to-TTL serial adaptor** You'll use this for testing only, just as you did in the last project.

Solar cell Circuit

» **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
» **1 MaxStream XBee OEM RF module** part number GC-WLM-XB24-A
» **1 3.3V regulator** The LD1117-33V (SparkFun part number COM-00526) or the MIC2940A-3.3WT (Digi-Key part number 576-1134-ND) work well.
» **1 2mm breakout board** The XBee modules listed here have pins spaced 2mm apart. To use them on a breadboard, you'll need a breakout board that shifts the spacing to 0.1 inches. SparkFun's Breakout Board for XBee Module (BOB-08276) does the trick.
» **2 rows of 0.1-inch header pins** as available from most electronics retailers.
» **2 2mm female header rows** Samtec part number MMS-110-01-L-SV. Samtec, like many part makers, supplies free samples of this part in small quantities. SparkFun sells these as part number PRT-08272.
» **1 1µF capacitor** Digi-Key part number P10312-ND
» **1 10µF capacitor** SparkFun part number COM-00523, Digi-Key part number P11212-ND
» **3 4700µF electrolytic capacitors** Digi-Key part number 493-1088-ND. Other vendors carry these, too.
» **1 MAX8212 voltage monitor.** You can order free samples from Maxim (www.maxim-ic.com) or order it from Digi-Key, part number MAX8212CPA+-ND.
» **1 10kΩ resistor**
» **3 100kΩ resistors**
» **1 4.7kΩ resistor**
» **1 1kΩ resistor**
» **1 2N3906 PNP-type transistor** such as Digi-Key part number 2N3906D26ZCT-ND, or SparkFun part number COM-00522
» **2 LEDs**
» **1 solar cell** SparkFun part number PRT-07840 works at an acceptable voltage, and can produce enough current on its own to power the radio.

XPort radio circuit

This is identical to the radio circuit in the previous project.

» **1 Lantronix embedded device server** Available from many vendors, including Symmetry part number CO-E1-11AA (Micro), WM11A0002-01 (WiMicro), or XP1001001-03R (XPort). This example uses an XPort.
» **1 RJ45 breakout board** SparkFun part number BOB-00716 (needed only if you're using an XPort)
» **1 solderless readboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
» **1 MaxStream XBee OEM RF module** part number GC-WLM-XB24-A
» **1 3.3V regulator** The LD1117-33V (SparkFun part number COM-00526) and the MIC2940A-3.3WT (Digi-Key part number 576-1134-ND) work well.
» **1 2mm breakout board** SparkFun's Breakout Board for XBee Module (BOB-08276)
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows** Samtec part number MMS-110-01-L-SV. SparkFun sells these as part number PRT-08272.
» **1 1µF capacitor** Digi-Key part number P10312-ND
» **1 10µF capacitor** SparkFun part number COM-00523, or Digi-Key part number P11212-ND
» **2 LEDs**
» **1 reset switch** Any momentary switch such as SparkFun's COM-00097 or Digi-Key's SW400-ND.

Relay radio circuit

This circuit is just an XBee radio by itself, powered by a 9V battery.

» **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
» **1 Maxstream XBee OEM RF module** available from http://www.maxstream.net, or http://www.gridconnect.com, part number GC-WLM-XB24-A
» **1 3.3V regulator.** The LD1117-33V (SparkFun part number COM-00526) or the MIC2940A-3.3WT (Digi-Key part no. 576-1134-ND) will work well.
» **1 2mm breakout board.** SparkFun's Breakout Board for XBee Module (BOB-08276)
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows** Samtec part number MMS-110-01-L-SV. SparkFun sells these as part number PRT-08272.
» **2 LEDs**
» **2 1µF capacitors** Digi-Key part number P10312-ND
» **1 10µF capacitor** SparkFun part number COM-00523, or Digi-Key part number P11212-ND
» **1 9V battery clip**
» **1 9V battery** You can use 3 or 4 AA batteries as well, if you have a battery holder for them.

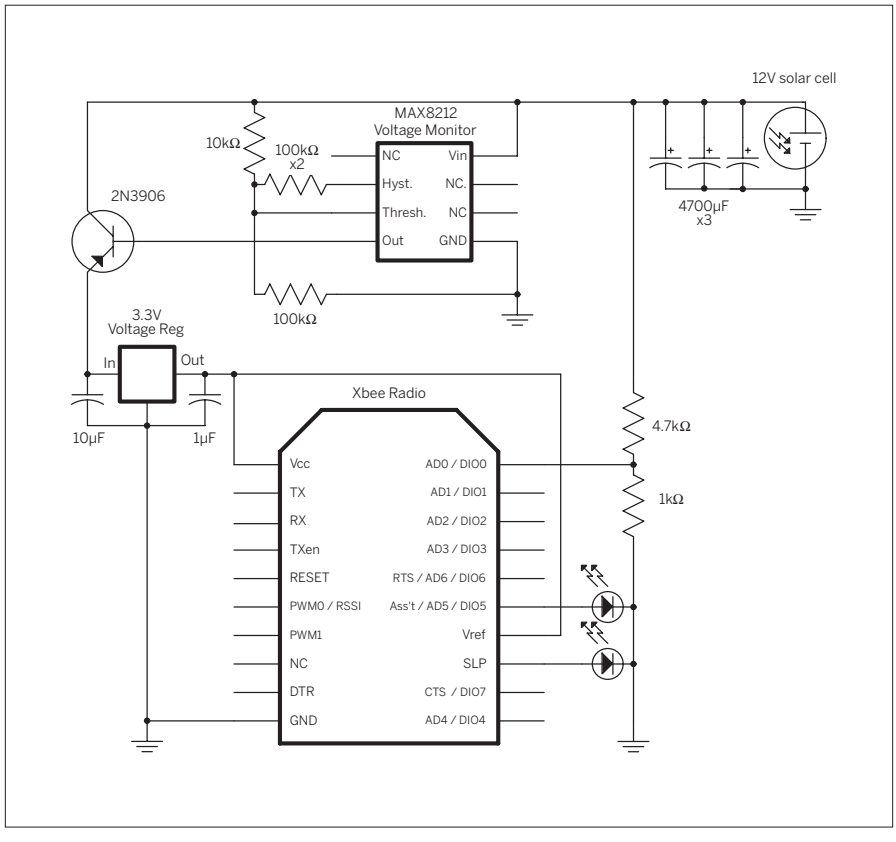
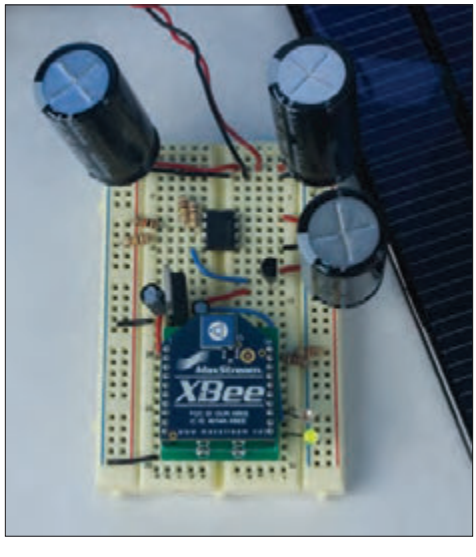
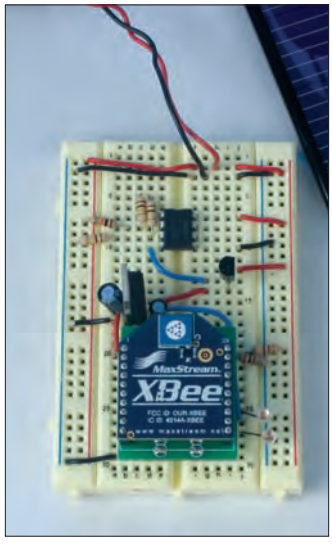
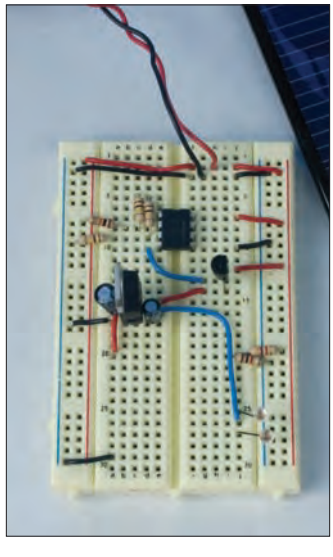
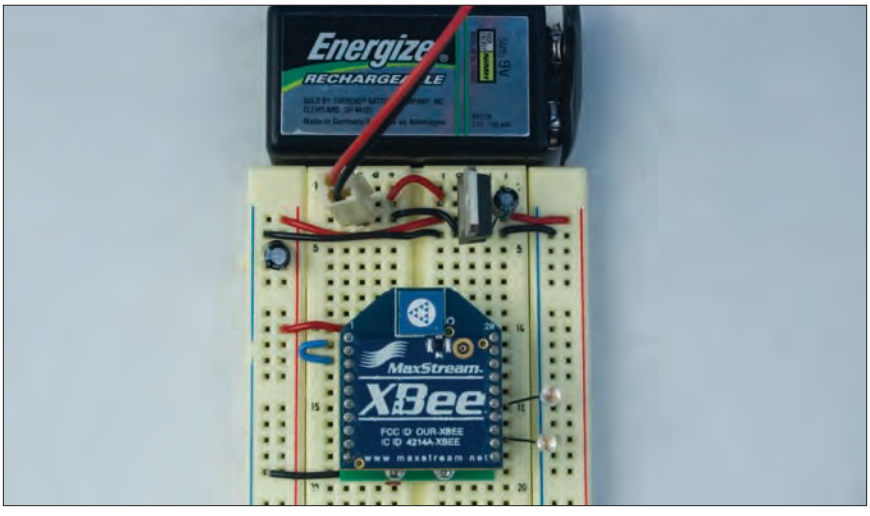


Figure 7-18
XBee radio attached to a solar cell. The detail photos show the circuit without the 4700µF capacitors and without the XBee, to reveal the components and wires beneath.









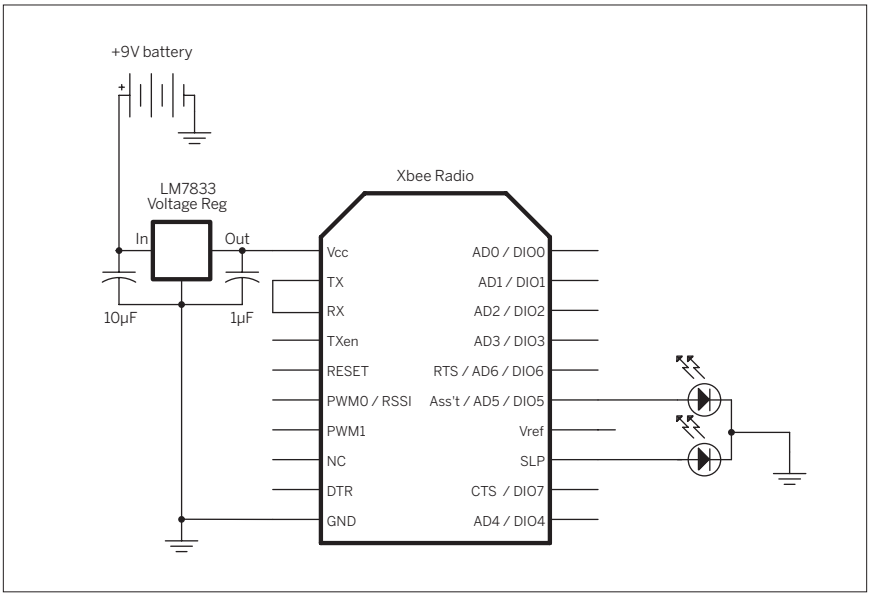


Figure 7-19
The XBee radio relay circuit.

The relay radio circuit is very simple. It's just a radio on a battery with its transmit pin and receive pin connected together. This is how it will relay the messages. Any incoming messages will get sent out the serial transmit pin, then back into the receive pin, where they will be sent out again as transmissions. Figure 7-19 shows the circuit.

Once you've got the radios configured and working, you need to configure the XPort so that it can pass the messages received from its radio on to the Processing program. The configuration is identical to that shown earlier in the "Sending UDP Datagrams to and from a Lantronix Device" section. The connect Mode is 0xCC, the datagramType is 01, the remote IP address is the address of your personal computer, and the remote port number is 10002. Set SendChar 1 and 2 both to 00.

The Graphing Program

Now that all the hardware is ready, it's time to write a Processing sketch to graph the data. The beginning of the program looks a lot like the UDP tester program shown earlier.

First you need to import the UDP library, initialize it, and write a method to listen for incoming datagrams:

This program will print out strings of numbers that look a lot like the initial ones from the PHP program in the VOC sensor project. That's because the datagrams the program is receiving are the same protocol — the XBee protocol for sending analog readings.

```
/*  XBee Packet Reader and Graphing Program
     Reads a packet from an XBee radio via UDP and parses it.
     Graphs the results over time.
    language: Processing

     Reads a packet from an XBee radio
 */

import hypermedia.net.*;
import processing.serial.*;

UDP udp;                          // define the UDP object
int queryPort = 10002;            // the port number for the device query

void setup() {
  // create a new connection to listen for
  // UDP datagrams on query port:
  udp = new UDP(this, queryPort);

  // listen for incoming packets:
  udp.listen( true );
}

void draw() {
    // nothing happens here.
}

/*
  listen for UDP responses
 */
void receive( byte[] data, String ip, int port ) {
  int[] inString = int(data);    // incoming data converted to string
  print(inString);
  println();
}
```

The next thing to do is to add a method to interpret the protocol. Not surprisingly, this looks a lot like the parsePacket() function from the PHP program in the previous project. Add this method to the end of your program.

To call it, replace the print() and println() statements in the receive() method with this:

```
parseData(inString);
```

```
void parseData(int[] thisPacket) {
  int adcStart = 11;                      // ADC reading starts at byte 12
  int numSamples = thisPacket[8];         // number of samples in packet
  int[] adcValues = new int[numSamples]; // array to hold the 5 readings
  int total = 0;                          // sum of all the ADC readings
  int rssi = 0;                           // the received signal strength

  // read the address -- a two-byte value, so you
  // add the two bytes as follows:
  int address = thisPacket[5] + thisPacket[4] * 256;

  // read the received signal strength:
  rssi = thisPacket[6];
```

Continued from previous page.

```
  // read <numSamples> 10-bit analog values, two at a time
  // because each reading is two bytes long:
  for (int i = 0; i < numSamples * 2;  i=i+2) {
    // 10-bit value = high byte * 256 + low byte:
    int thisSample = (thisPacket[i + adcStart] * 256) +
      thisPacket[(i + 1) + adcStart];
    // put the result in one of 5 bytes:
    adcValues[i/2] = thisSample;
    // add the result to the total for averaging later:
    total = total + thisSample;
  }
  // average the result:
  int average = total / numSamples;
  print("Average reading:" + average + "\t");
  // print the received signal strength:
  println("Signal Strength:" + rssi);
}
```

» Now that you've got the average reading printing out, add some code to graph the result. For this, you'll need a new global variable before the setup() method that keeps track of where you are horizontally on the graph:

```
int hPos = 0;           // horizontal position on the graph
```

» You'll also need to add a line at the beginning of the setup() method to size the window:

```
  // set the window size:
  size(400,300);
```

» Now add a new method, drawGraph(), to the end of the program:

```
/*
  update the graph
 */
void drawGraph(int graphValue) {
  // draw the line:
  stroke(0,255,0);
  line(hPos, height, hPos, height - graphValue);
  // at the edge of the screen, go back to the beginning:
  if (hPos >= width) {
    hPos = 0;
    //wipe the screen:
    background(0);
  }
  else {
    // increment the horizontal position to draw the next line:
    hPos++;
  }
}
```

Call this from the parseData() method, replacing the println() statement that prints out the average, as well as the println() statement that prints out the signal strength (rssi), like so:

```
// draw a line on the graph:
drawGraph(average/4);
```

Now when you run the program, it should draw a graph of the sensor readings, updating every time it gets a new datagram.

» Finally, add some code to add a time stamp. This task requires one new global variable before the setup() to set the size of a font to draw text on the screen:

```
int fontSize = 14;            // size of the text font
```

» Then add two lines to the setup() method to initialize the font, The first line picks a font from the list of available system fonts, and the second initializes the font (I didn't like the first font in my system's list, so I went with the second — choose your own as you see fit):

```
  // create a font with the second font available to the system:
  PFont myFont = createFont(PFont.list()[1], fontSize);
  textFont(myFont);
```

» Add two methods to the end of the program, eraseTime() and drawTime(). The latter draws the date and time, and the former draws a black block over the previous date and time:

```
/*
  Draw a black block over the previous date and time strings
 */

void eraseTime(int xPos, int yPos) {
  // use a rect to block out the previous time, rather than
  // redrawing the whole screen, which would mess up the graph:
  noStroke();
  fill(0);
  rect(xPos,yPos, 120, 80);
  // change the fill color for the text:
  fill(0,255,0);
}

/*
  print the date and the time
 */
void drawTime(int xPos, int yPos) {
  // set up an array to get the names of the months
  // from their numeric values:
  String[] months = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
    "Sep", "Oct", "Nov", "Dec"            };

  String date = "";          // string to hold the date
  String time = "";          // string to hold the time

  // format the date string:
  date += day();
  date += " ";
  date += months[month() -1];
  date += " ";
  date += year();

  // format the time string:
  time += hour();
```

You're going to call these methods from a few different places in the program. The first is at the end of the setup() method, to show the initial time:

```
// show the initial time and date:
background(0);
eraseTime(hPos, 0);
drawTime(hPos, 0);
```

The next is at the end of the parseData() method, to draw the time of the current line:

```
// draw a line on the graph:
drawGraph(average/4);
eraseTime (hPos - 1, fontSize * 2);
drawTime(hPos, fontSize * 2);
```

Continued from previous page.

```
  time += ":";
  if (minute() < 10) {
    time += "0";
    time += minute();
  }
  else {
    time +=minute();
  }
  time += ":";
  if (second() < 10) {
    time += "0";
    time += second();
  }
  else {
    time +=second();
  }

  // print both strings:
  text(date, xPos, yPos + fontSize);
  text(time, xPos, yPos + (2 * fontSize));
}
```

Finally, call these methods from the drawGraph() method, in the part where you reset the whole graph. This is inside the if() statement that checks to see where the horizontal position is. This way, each time the graph reaches the edge of the window, it erases the whole screen, and updates the initial time:

```
  if (hPos >= width) {
    hPos = 0;
    //wipe the screen:
    background(0);
    // wipe the old date and time, and draw the new:
    eraseTime(hPos, 0);
    drawTime(hPos, 0);
  }
```

That's the whole program. When it's running, it should look like Figure 7-20.

To view the program in its entirety, see Appendix C.

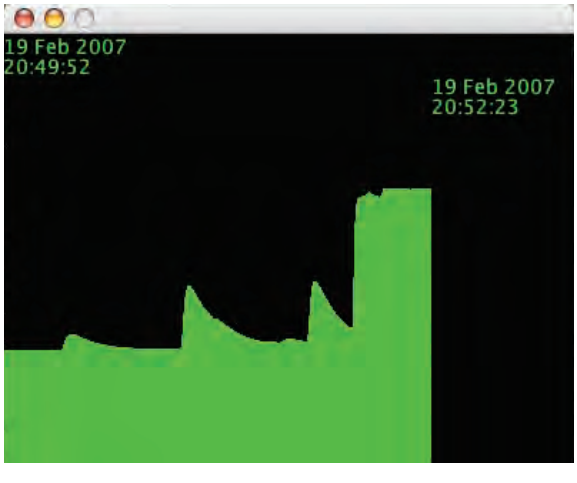


Figure 7-20
The output of the solar graph program. These sensor values were faked with a flashlight! Your actual values may differ.

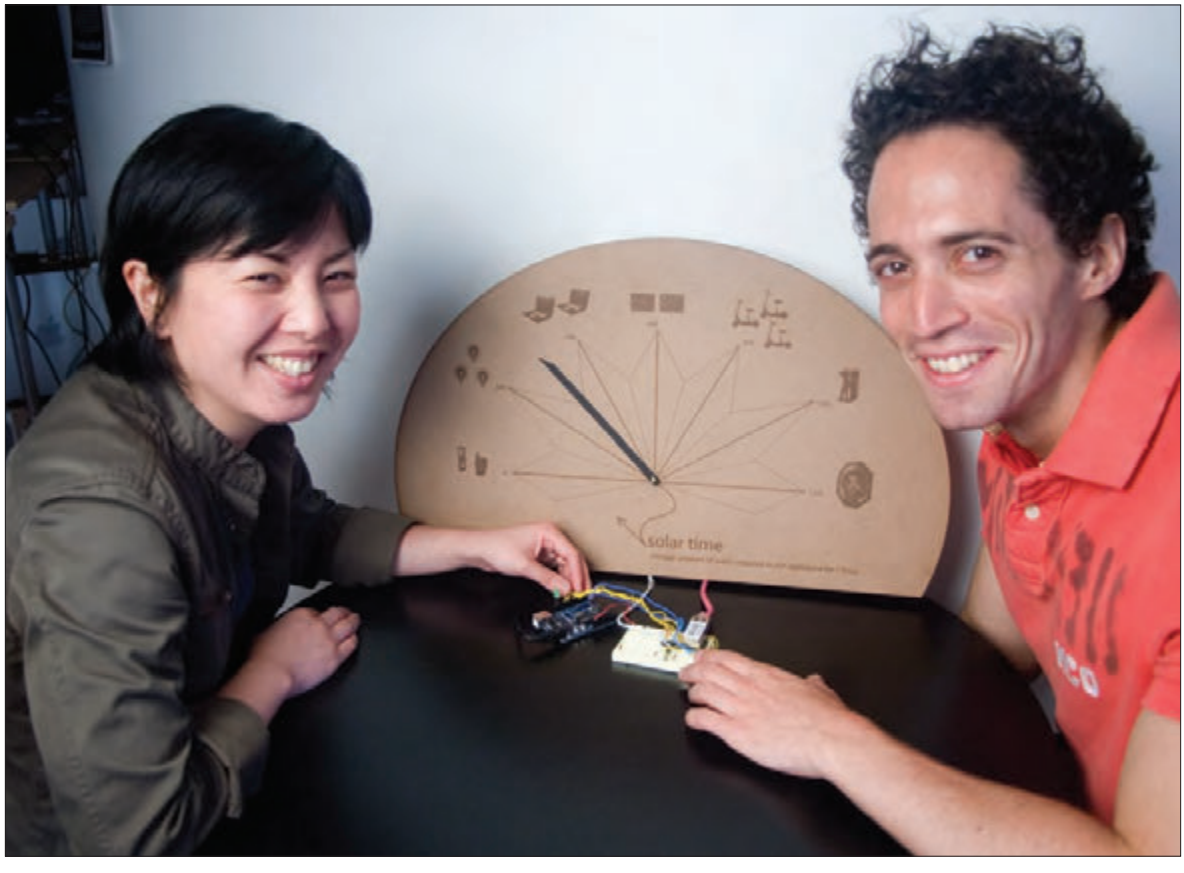
Conclusion

Sessionless networks can be really handy when you're just passing short messages around and don't need a lot of acknowledgment. They involve a lot less work, because you don't have to maintain the connection. They also give you a lot more freedom in how many devices you want to address at once.

As you can see by comparing the two projects in this chapter, there's not a lot of work to be done to switch from directed messages and broadcast messages when you're making a sessionless network. It's best to default to directed messages when you can, in order to reduce the traffic for those devices that don't need to get every message.

Now that you've got a good grasp of both session-based and sessionless networks, the next chapters switch direction slightly to talk about two other activities connecting networks to the physical world: location and identification.

X



The solar energy display, by Gilad Lotan and Angela Pablo.

8

MAKE: PROJECTS

How to Locate (Almost) Anything

By now, you've got a pretty good sense of how to make things talk to each other over networks. You've learned about packets, sockets, datagrams, clients, servers, and all sorts of protocols. Now that you know how to talk, the last two chapters deal with two common questions: where am I, and who am I talking to? Location technologies and identification technologies share some important properties. As a result, it's not uncommon to confuse the two, and to think that a location technology can be used to identify a person or an object, and vice versa. These are two different tasks in the physical world, however, and often in the network environment as well. Systems for determining physical location aren't always very good at determining identity, and identification systems don't do a good job of determining precise location. Likewise, knowing who's talking on a network doesn't always help you to know where the speaker is. In the examples that follow, you'll see methods for determining location and identification in both physical and network environments.

Address 2007 by Mouna Andraos and Sonali Sridhar

This necklace contains a GPS module. When activated, it displays the distance between the necklace and your home location. *Photo by J. Nordberg.*

Network Location and Physical Location

Locating things is one of the most common tasks people want to achieve with sensor systems. Once you understand the wide range of things that sensors can detect, it's natural to get excited about the freedom this affords. All of a sudden, you don't have to be confined to a chair to interact with computers. You're free to dance, run, jump — and it's still possible for a computer to read your action and respond in some way.

The downside of this freedom is the perception that in a networked world, you can be located anywhere. Ubiquitous surveillance cameras and systems like Wireless E911, a system for locating mobile phones on a network, make it seem as if anyone or anything can be located anywhere and at any time, whether you want to be located or not. The reality of location technologies lies somewhere in between these extremes.

Locating things on a network is different than locating things in physical space. As soon as a device is connected to a network, you can get a general idea of its location using a variety of means, from address lookup to measuring its signal strength, but that doesn't mean that you know its physical location. You just know its relationship to other nodes of the network. You might know that a cell phone is closest to a given cell transmitter tower, or that a computer is connected to a particular Wi-Fi access point. You can use that information along with other data to build up a picture of the person using the device. If you know that the cell transmitter tower is less than a kilometer from you, then you'd know that the person with the cell phone is going to reach you soon, and you can act appropriately in response. For many network applications, you don't need to know physical location as much as you need to know relationship to other nodes in the network.

Step 1: Ask a Person

People are really good at locating things. At the physical level, we have a variety of senses to throw at the problem and a brain that's wonderful at matching patterns of shapes and determining distances from different sensory clues. At the behavioral level, we've got thousands of patterns of behavior that make it easier to determine why you might be looking for something. Computer systems don't have these same advantages, so when you're designing an interactive system to locate things or people, the best tool you have to work with, and the first one you should consider, is the person for whom you're making your system.

Getting a good location starts with cultural and behavioral cues. If you want to know where you are, ask another person near you. In an instant, she's going to sum up all kinds of things, like your appearance, your behavior, the setting you're both in, the things you're carrying, and more, in order to give you a reasonably accurate and contextually relevant answer. No amount of technology can do that, because the connection between where we are and why we want to know is always an abstract thing. As a result, the best thing you can do when you're designing a locating system is to harness the connection-making talents of the person who will be using that system. Providing him with cues as to where to position himself when he should take action, and what actions he can take, helps eliminate the need for a lot of technology. Asking him to tell your system where things are, or to position them so that the system can easily find them, makes for a more effective system.

For example, imagine you're making an interactive space that responds to the movements of its viewers. This is popular among interactive artists, and often they begin by imagining a "body-as-cursor" project, in which the viewer is imagined as a body moving around in the space of the gallery. Some sort of tracking system is needed to determine his position and report it back in two dimensions, like the position of a cursor on a computer screen.

What's missing here is the reason why the viewer might be moving in the first place. If you start by defining what the viewer's doing, and give him cues as to what you expect him to do at each step, you can narrow down the space in which you need to track him. Perhaps you only need to know when he's approaching one of several sculptures in the space, so that you can trigger the sculpture to move in response. If you think of the sculptures as nodes in a network, the task gets easier. Instead of tracking the

viewer in an undefined two-dimensional space, now all you have to do is to determine his proximity to one of several points in the room. Instead of building a tracking system, you can now just place a proximity sensor near each object and look up which he's near, and read how near he is. You're using a combination of spatial organization and technology to simplify the task. You can make your job even easier by giving him visual, auditory, and behavioral cues to interact appropriately. He's no longer passive; he's now an active participant in the work.

Or take a different example: let's say you're designing a mobile phone city guide application for tourists that relies on knowing the phone's position relative to nearby cell towers to determine the phone's position. What do you do when you can't get a reliable signal from the cell towers? Perhaps you ask the tourist to input the address she's at, or the postal code she's in, or some other nearby cue. Then your program can combine that data with the location based on the last reliable signal it received, and determine a better result. In these cases, and in all location-based systems, it's important to incorporate human talents in the system to make it better.

Step 2: Know the Environment

Before you can determine where you are, you need to determine your environment. For any location, there are several ways to describe it. For example, you could describe a street corner in terms of its address, its latitude and longitude, its postal code, or the businesses nearby. Which of these coordinates you choose depends in part on the technology you have on hand to determine it. If you're making the mobile city guide described earlier, you might use several different ones: the nearest cell transmitter ID, the street address, and the nearby businesses might all work to define the location. In this case, as in many, your job in designing the system is to figure out how to relate one system of coordinates to another in order to give some meaningful information.

Mapping places to coordinate systems is a lot of work, so most map databases are incomplete. Geocoding allows you to look up the latitude and longitude of most any U.S. street address. It doesn't work everywhere in the U.S., and it doesn't work most places outside the U.S. because the data hasn't been gathered and put in the public domain for everywhere. Geocoding depends on having an accurate database of names mapped to locations. If you don't agree on the names, you're out of luck. The Virtual Terrain Project (www.vterrain.org) has a good list of geocoding resources for the US and international locations at www.vterrain.org/Culture/geocoding.html. Geocoder.net offers a free US-based lookup at www.geocoder.net, and Worldkit offers an extended version that also looks up international cities, at www.worldkit.org/geocoder.

Street addresses are the most common coordinates that are mapped to latitude and longitude, but there are other systems that'd be useful to have physical coordinates for as well. For example, mobile phone cell transmitters all have physical locations. It would be handy to have a database of physical coordinates for those towers. However, cell towers are privately owned by mobile telephone carriers, so detailed data about the tower locations is proprietary, and the data is not in the public domain. Projects such as CellSpotting (www.cellspotting.com) attempt to map cell towers by using GPS-equipped mobile phones running custom software. As there are many different mobile phone operating systems, even developing the software to do the job is a huge challenge. Open source cell geocoding is still in its infancy, so finding a complete database is difficult.

IP addresses don't map exactly to physical addresses, because computers can move. Nevertheless, there are several geocoding databases for IP addresses. These work on the assumption that routers don't move a lot, so if you know the physical location of a router, then the devices gaining access to the Net through that router can't be too far away. The accuracy of IP geocoding is limited, but it can help you determine a general area of the world, and sometimes even a neighborhood or city block, where a device on the Internet is located. Of course, IP lookup doesn't work on private IP addresses. In the next chapter, you'll see an example that combines network identity and geocoding.

You can develop your own database relating physical locations to cultural or network locations, if the amount of information you need is small, or you have a large group of people to do the job. But generally, it's better to rely on existing infrastructures when you can.

Step 3: Acquire and Refine

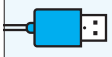
Once you know where you're going to look, there are two tasks that you have to do continually: acquire a new position, and refine the position's accuracy. Acquisition gives a rough position. Acquisition usually starts by identifying which device on a network is the center of activity. In the interactive installation example described earlier,

you could acquire a new position by determining that the viewer tripped a sensor near one of the objects in the room. Once you know roughly where he is, you can refine the position by measuring his distance with the proximity sensor attached to the object.

Refining doesn't have to mean getting a more accurate physical position. When you have a rough idea of where something's happening, you need to know about the activity at that location in order to provide a response. In the interactive installation example, you may never need to know the viewer's physical coordinates in feet and inches (or meters and centimeters). When you know which object he's close to in the room, and whether he's close enough to relate to it, you can make that object respond. You might be changing the graphics on an LCD display when he walks close to it, or activating an animatronic sculpture as he walks by. In both cases, you don't need to know the precise distance; you just need to know he's close enough to pay attention. Sometimes distance ranging sensors are used as motion detectors to define general zones of activity rather than to measure distance.

Determining proximity doesn't always give you enough information to take action. Refining can also involve determining the orientation of one object relative to another. If you're giving directions from one location to another, you need to know which way you're oriented. It's also valuable information when two people or objects are close to each other. You don't want to activate the animatronic sculpture if the viewer has his back to the thing!

X



35 Ways to Find Your Location

At the 2004 O'Reilly Emerging Technology Conference, Interaction designer and writer Chris Heathcote gave an excellent presentation on cultural and technological solutions to finding things entitled *35 Ways to Find Your Location*. He outlined a number of important factors to keep in mind before you choose tools to do the job. He pointed out that the best way to locate someone or something involves a combination of technological methods and interpretation of cultural and behavioral cues. His list is a handy tool for inspiring solutions when you need to develop a system to find locations. A few of the more popular techniques that Chris listed are:

- **Assume: the Earth. Or a smaller domain, but assume that's the largest space you have to look in.**
- **Use the time.**
- **Ask someone.**
- **Association: who or what are you near?**
- **Proximity to phone boxes, public transport stops, and utility markings.**
- **Use a map.**
- **Which cell phone operators are available?**

  Public phone operators?
- **Phone number syntax?**
- **Newspapers available?**
- **Language being spoken?**
- **Post codes/ZIP codes**
- **Street names.**
- **Street corners/intersections.**
- **Street numbers.**
- **Business names.**
- **Mobile phone location, through triangulation or trilateration.**
- **Triangulation and trilateration on other radio infrastructures, like TV, radio, and public Wi-Fi.**
- **GPS, assisted GPS, WAAS, and other GPS enhancements.**
- **Landmarks and "littlemarks."**
- **Dead reckoning.**

Determining Distance

Electronic locating systems like GPS, mobile phone location, and sonar seem magical at first, because there's no visible evidence as to how they work. When you break the job down into its components, it becomes relatively straightforward. Most physical location systems are based on the same principle. They determine distance from several known and fixed locations by measuring the energy of an electromagnetic or acoustic wave coming from the object to be located. Then they combine those measurements to determine a position in two or three dimensions.

For example, a GPS receiver determines its position on the surface of the planet by measuring the strength of received radio signals from several geosynchronous satellites. Similarly, mobile phone location systems measure the signal strength of the phone at several cell towers. Sonar and infrared ranging sensors work by sending out an acoustic signal (sonar) or an infrared signal (IR rangers) and measuring the strength of that signal when it's reflected off the target.

Distance ranging techniques can be classified as active or passive. In active systems, the target has a radio, light, or acoustic source on it, and the receiver just listens for the signal from the target. In passive systems, the target doesn't have to have any technology on board. The receiver emits a signal, and the signal bounces off the target. Mobile phone location is active, because it relies on the phone sending out a radio signal. Sonar and infrared ranging are passive, because the sensor has to emit a signal in order to measure the reflection. GPS is an active locating technology, because although the receiver doesn't emit a signal, it has an electronic receiver onboard to receive satellites' signals.

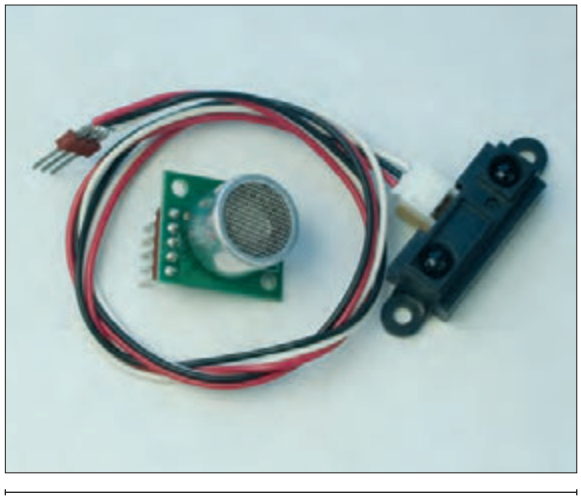


Figure 8-1
Devantech SRF02 and Sharp GP2D12 sensors. The Devantech sensor can read a range from 15 cm to 6 m. The Sharp sensor can read a range from 10 cm to 80cm.

Sometimes distance ranging is used for acquiring a position, and other times it's used for refining it. In the following examples, the passive distance rangers deliver a measurement of physical distance, but the radio ranging tell you only when another radio is in transmission range of your radio, and whether it's near or far within the range.

Passive Distance Ranging

Ultrasonic rangers like the Devantech SRF02 and infrared rangers like the Sharp GP2D12, shown in Figure 8-1, are examples of distance rangers. The Devantech sensor sends an ultrasonic signal out and listens for an echo; it's basically a sonar device. The Sharp sensor sends out an infrared light beam, and senses the reflection of that beam. These sensors only work in a short range. The Sharp sensor can read about 10 cm to 80 cm, and the Devantech sensor reads from about 15 cm to 6.4 m; these are useful only for very local measurements. Passive sensors like these are handy, though, when you want to measure the distance of a person in a limited space, and you don't want to have to put any hardware on the person. They're also handy when you're building moving objects that need to know their proximity to other objects in the same space as they move.

Project 14

Infrared Distance Ranger Example

The Sharp GP2*xx* series of infrared ranging sensors give a decent measurement of short-range distance by bouncing an infrared light signal off the target and measuring the returned brightness. They're very simple to use. Shown in Figure 8-2 is a circuit for a Sharp GP2D12 IR ranger, which can detect an object in front of it within about 10 cm to 80 cm. The sensor requires 5V power, and outputs an analog voltage from 0 to 5V, depending on the distance to the nearest object in its sensing area.

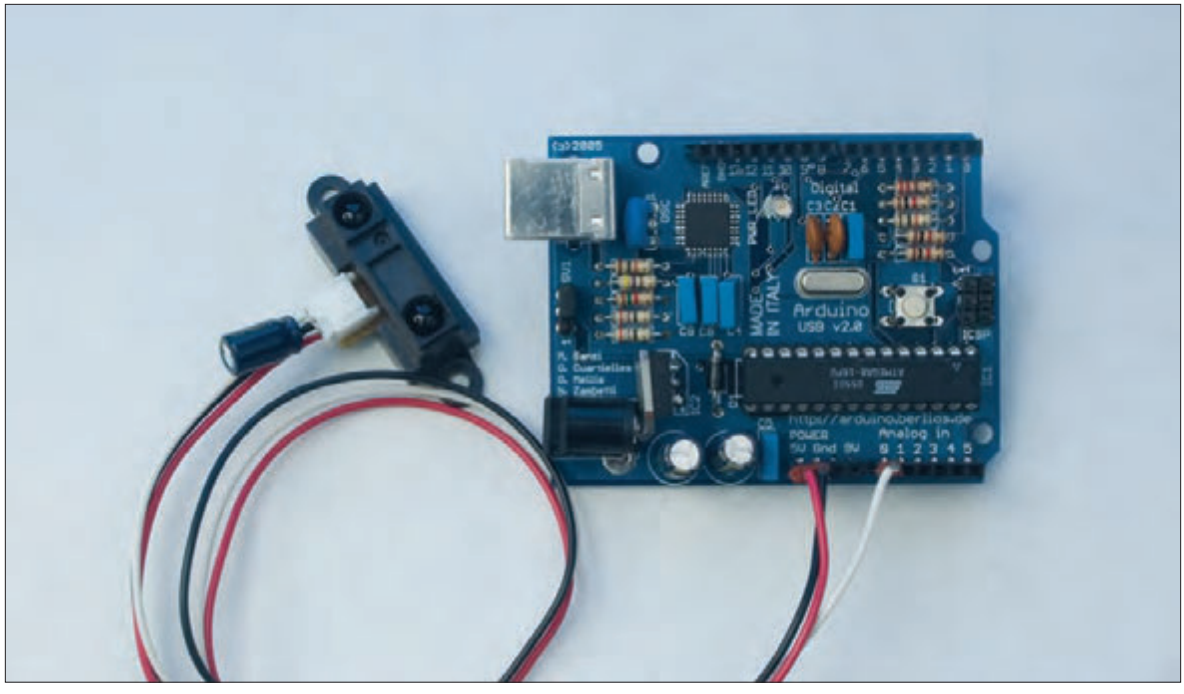
MATERIALS

- **1 solderless breadboard** such as Digi-Key (www.digikey.com) part number 438-1045-ND, or Jameco (www.jameco.com) part number 20601. The breadboard is optional; as shown in the photo, you can assemble this without it.
- **1 Arduino module** or other microcontroller
- **1 Sharp GP2D12 IR ranger** Acroname (www.acroname.com) part number R48-IR12; SparkFun (www.sparkfun.com) part number SEN-00242 for the Sharp GP2Y0A21YK, a similar model; Trossen Robotics (www.trossenrobotics.com) part number S-10-GP2D12; Digi-Key part number 425-2469-ND. It's best to buy the connector and cable needed with the sensor, as they are difficult to make. Acroname sells these as part number R47-JSTCON-2, Trossen Robotics as part number S-10-GP2D12C.
- **1 10µF capacitor**

» The Sharp sensors' outputs are not linear, so if you want to get a linear range, you need to make a graph of the voltage over distance, and do some math. Fortunately, the good folks at Acroname Robotics have done the math for you. For the details, see www.acroname.com/robotics/info/articles/irlinear/irlinear.html.

The next program reads the sensor and outputs the distance measured in centimeters. The conversion formula gives only an approximation, but it's accurate enough for general purposes.

```
/*
  Sharp GP2D12 IR ranger reader
  language: Wiring/Arduino

  Reads the value from a Sharp GP2D12 IR ranger and sends it
  out serially.
*/

int sensorPin = 0;     // Analog input pin
int sensorValue = 0;   // value read from the pot

void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  sensorValue = analogRead(sensorPin); // read the pot value

  // the sensor actually gives results that aren't linear.
  // this formula converts the results to a linear range.
  int range = (6787 / (sensorValue - 3)) - 4;

  Serial.println(range, DEC);    // print the sensor value
  delay(10);                     // wait 10 milliseconds
                                 // before the next loop
}
```



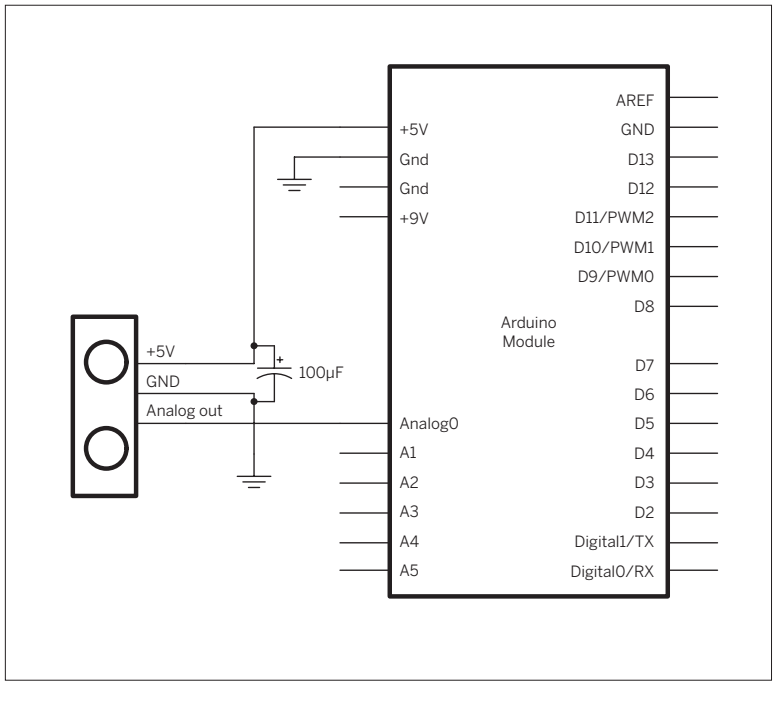


Figure 8-2
The Sharp GP2D12 IR ranger attached to a microcontroller. The capacitor attached to the body of the sensor smoothes out fluctuations due to the sensor's current load.

Project 15

Ultrasonic Distance Ranger Example

The Devantech SRF*xx* ultrasonic sensors measure distance using a similar method to the Sharp sensors, but have a greater sensing range. They send an ultrasonic signal out and wait for the echo to return, and measure the distance based on the time required for the echo to return. These sensors require 5V power, and return their results via an I2C synchronous serial protocol. The SRF02 sensors and other SRF*xx* sensors like the SRF08 and SRF10, which use the same protocol, are available from the sites www.acroname.com and robot-electronics.co.uk. The MaxBotix EZ1 and LV-EZ1 (available from www.sparkfun.com) are ultrasonic rangers that are similar to the Devantech ones, but that use TTL serial, pulsewidth output, or analog voltage output instead of I2C.

MATERIALS

- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
- **1 Arduino module** or other microcontroller
- **1 Devantech SRF02 ultrasonic ranger** Acroname Robotics part number R287-SRF02
- **1 100µF capacitor** SparkFun part number COM-00096, or Digi-Key part number P10195-ND

I2C is comparable to RS-232 or USB, in that it doesn't define the application — just the way that data is sent. Every I2C device uses two wires to send and receive data: a serial clock pin, called the SCL pin, that the microcontroller pulses at a regular interval, and a serial data pin, called the SDA pin, over which data is transmitted. For each serial clock pulse, a bit of data is sent or received. When the clock changes from low to high (known as the rising edge of the clock), a bit of data is transferred from the microcontroller to the I2C device. When the clock changes from high to low (known as the falling edge of the clock), a bit of data is transferred from the I2C device to the microcontroller.

The Arduino module uses analog pin 4 as the SDA pin, and analog pin 5 as the SCL pin. Figure 8-3 shows the SRF02 sensor connected to an Arduino.

Distance rangers have a limited conical field of sensitivity, so they're not great for determining location over a large two-dimensional area. The Devantech SRF02 sensor, for example, has a cone-shaped field of sensitivity that's about 55 degrees wide and 6 meters from the sensor to the edge of the range. In order to use it to cover a room, you'd need to use several of them and get creative about how you arrange them. Figure 8-4 shows one way to cover a 4m x 4m space using five of the SRF02 rangers. In this case, you'd need to make sure that no two of the sensors were operating at the same instant, because their signals would interfere with each other. The sensors would have to be activated one after another in sequence. Because each one takes up to 36 milliseconds to return a result, you'd need up to 180 milliseconds to make a complete scan of the space.

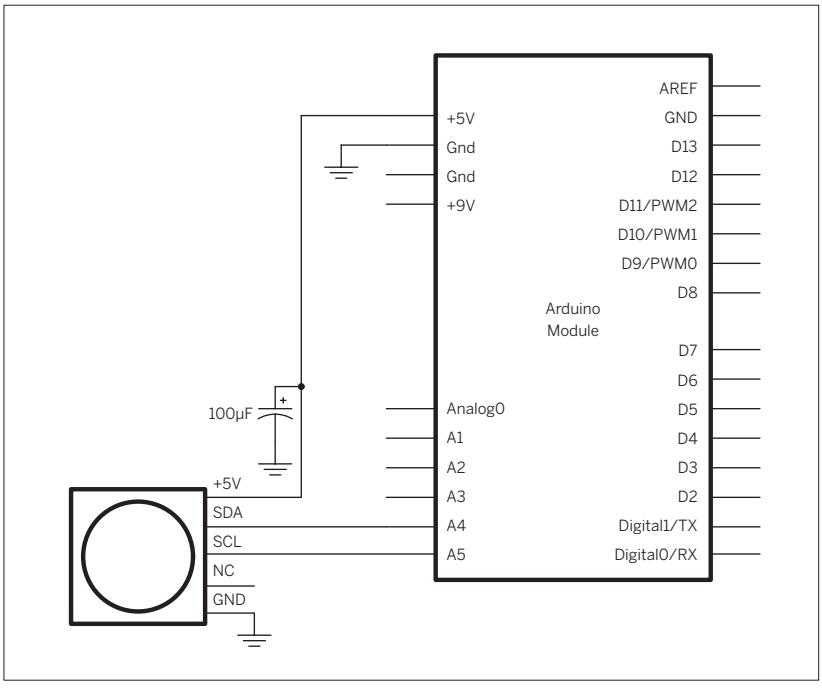
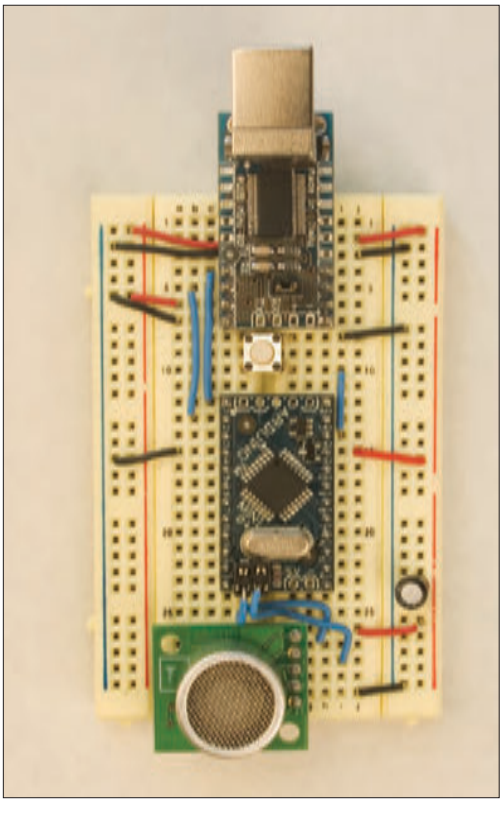
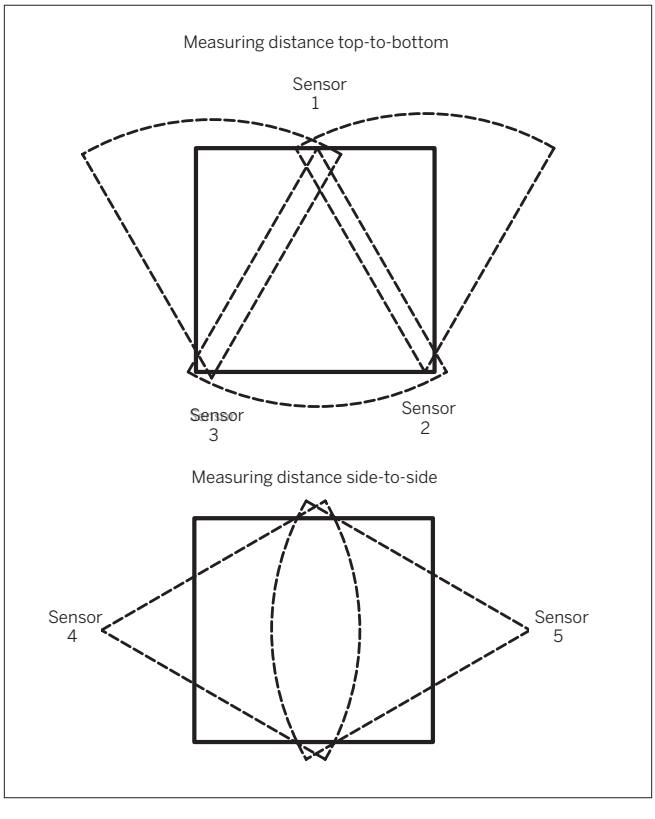


Figure 8-3
SRF02 ultrasonic sensor connected to an an Arduino Mini 04 module. Female headers have been soldered to the Arduino Mini's analog 4-7 holes to make them easier to use.

» Bottom right

Figure 8-4
Measuring distance in two dimensions using ultrasonic distance rangers. The square in each drawing is a 4m × 4m floor plan of a room. In order to cover the whole of a rectangular space, you need several sensors placed around the side of the room.





Try It This sketch sends commands to the SRF02 sensor to take distance readings and return them to the microcontroller.

NOTE: If you have trouble compiling this sketch, you might need to delete the compiled version of the Wire library, and let Arduino recompile it the next time you compile your program. The file you need to delete is called **Wire.o**, and it's in a subdirectory of the Arduino application directory: **lib/targets/libraries/Wire/**. In general, for any library you want to recompile, you can delete the object file (the **.o** file), and Arduino will recompile it.

```
/*
  SRF02 sensor reader
 language: Wiring/Arduino

 Reads data from a Devantech SRF02 ultrasonic sensor.
 Should also work for the SRF08 and SRF10 sensors as well.

 Sensor connections:
 SDA - Analog pin 4
 SCL - Analog pin 5

 */
// include Wire library to read and write I2C commands:
#include <Wire.h>

// the commands needed for the SRF sensors:
#define sensorAddress 0x70
#define readInches 0x50
// use these as alternatives if you want centimeters or microseconds:
#define readCentimeters 0x51
#define readMicroseconds 0x52
// this is the memory register in the sensor that contains the result:
#define resultRegister 0x02


void setup()
{
  // start the I2C bus
  Wire.begin();
  // open the serial port:
  Serial.begin(9600);
}

void loop()
{
  // send the command to read the result in inches:
  sendCommand(sensorAddress, readInches);
  // wait at least 70 milliseconds for a result:
  delay(70);
  // set the register that you want to read the result from:
  setRegister(sensorAddress, resultRegister);

  // read the result:
  int sensorReading = readData(sensorAddress, 2);
  // print it:
  Serial.print("distance: ");
  Serial.print(sensorReading);
  Serial.println(" inches");
  // wait before next reading:
  delay(70);
}
```

Continued from opposite page.

```
/*
  SendCommand() sends commands in the format that the SRF sensors expect
 */
void sendCommand (int address, int command) {
  // start I2C transmission:
  Wire.beginTransmission(address);
  // send command:
  Wire.send(0x00);
  Wire.send(command);
  // end I2C transmission:
  Wire.endTransmission();
}
/*
  setRegister() tells the SRF sensor to change the address
  pointer position
 */
void setRegister(int address, int thisRegister) {
  // start I2C transmission:
  Wire.beginTransmission(address);
  // send address to read from:
  Wire.send(thisRegister);
  // end I2C transmission:
  Wire.endTransmission();
}
/*
readData() returns a result from the SRF sensor
 */
int readData(int address, int numBytes) {
  int result = 0;        // the result is two bytes long

  // send I2C request for data:
  Wire.requestFrom(address, numBytes);
  // wait for two bytes to return:
  while (Wire.available() < 2 )   {
    // wait for result
  }
  // read the two bytes, and combine them into one int:
  result = Wire.receive() * 256;
  result = result + Wire.receive();
  // return the result:
  return result;
}
```

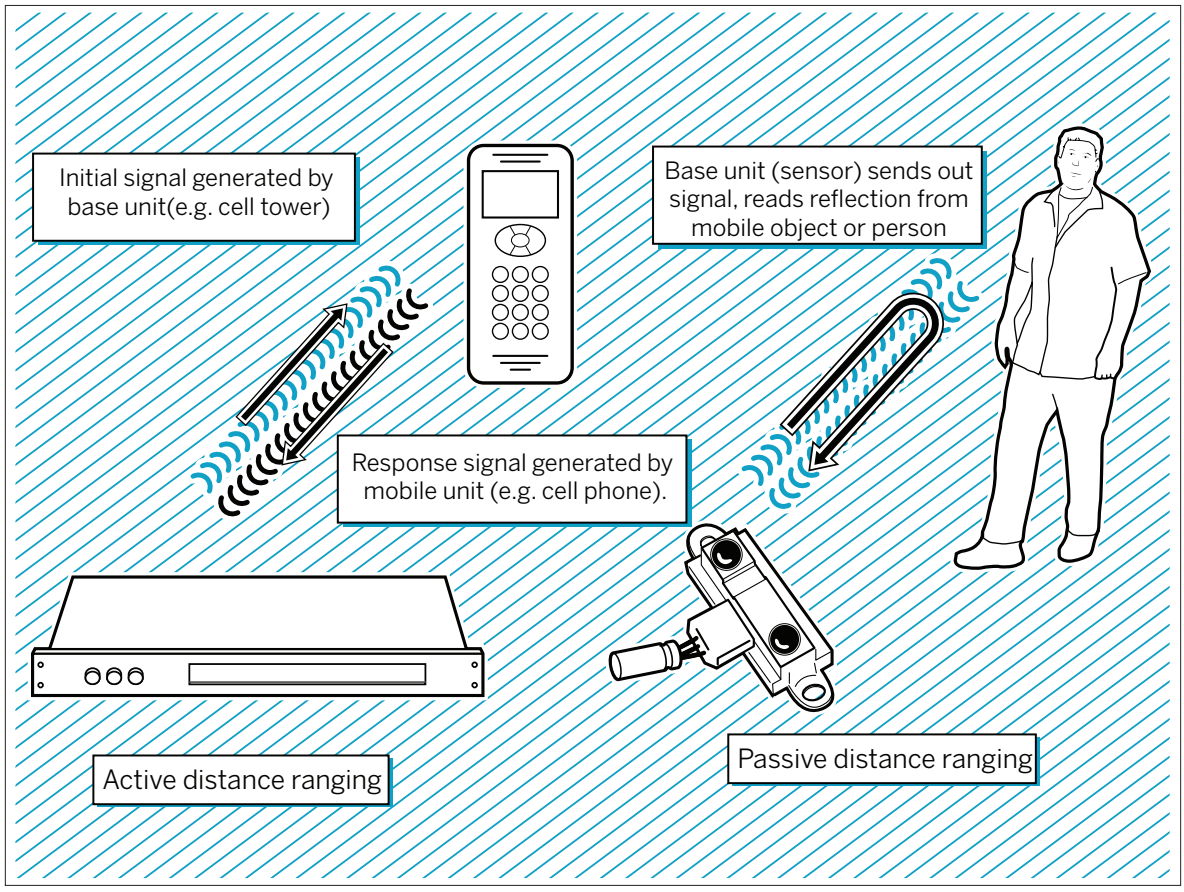
Active Distance Ranging

The ultrasonic and infrared rangers in the preceding sections are passive distance sensing systems. Mobile phones and the Global Positioning System measure longer distances by using ranging as well. These systems include a radio beacon (the cell tower or GPS satellite) and a radio receiver (the phone or GPS receiver). The receiver determines its distance from the beacon based on the received signal from the beacon. These systems can measure much greater distances, on an urban or global scale. The disadvantage of active distance ranging is that you must have both a sending device and receiving device. You can't measure a person's distance from somewhere using active distance ranging unless you attach a receiver to the person.

GPS and cellular location systems don't actually give you the distance from their radio beacons, just the relative signal strength of the radio signal. Bluetooth, 802.15.4, ZigBee, and Wi-Fi radios all provide data about signal strength as well. In order to relate this to distance, you need to be able to calculate that distance as a function of signal strength. The main function of a GPS receiver is to calculate distances to the GPS satellites based on signal strength and determine a position using those distances. The other radio systems mentioned here don't do those calculations for you.

In many applications, though, you don't need to know the distance — you just need to know how relatively near or far one person or object is to another. For example, if you're making a pet door lock that opens in response to the pet, you could imagine a Bluetooth beacon on the pet's collar, and a receiver on the door lock. When the signal strength from the pet's collar is strong enough, the door lock opens. In this case, and in others like it, there's no need to know the actual distance.

X



Project 16

Reading Received Signal Strength Using XBee Radios

In the last chapter, you saw the received signal strength, but you didn't do anything with it. The Processing code that read the solar cell's voltage output parsed the XBee packet for the received signal strength (RSSI). Here's a simpler variation on it that just reads the signal strength. To test it, you can use the same radio settings from the solar cell project and attach the receiving XBee radio to a USB-to-serial adaptor. See Figure 7-5 for the receiving circuit, and Figure 7-6 (the VOC sensor circuit) or Figure 7-18 (the solar cell circuit) for circuits that work well as transmitters.

» Run this Processing sketch to connect to the receiver radio via the USB-to-serial device. When you run this program, you'll get a graphing bar like that in Figure 8-5.

```
/*  XBee Signal Strength Reader
    Language: Processing

 Reads a packet from an XBee radio and parses it.  The packet
 should be 22 bytes long. It should be made up of the following:
 byte 1:     0x7E, the start byte value
 byte 2-3:   packet size, a 2-byte value  (not used here)
 byte 4:     API identifier value, a code that says what this response
             is (not used here)
 byte 5-6:   Sender's address
 byte 7:     RSSI, Received Signal Strength Indicator (not used here)
 byte 8:     Broadcast options (not used here)
 byte 9:     Number of samples to follow
 byte 10-11: Active channels indicator (not used here)
 byte 12-21: 5 10-bit values, each ADC samples from the sender

*/
import processing.serial.*;

Serial XBee ;                   // input serial port from the XBee Radio
int[] packet = new int[22];     // with 5 samples, the XBee packet is
                                // 22 bytes long
int byteCounter;                // keeps track of where you are in
                                // the packet
int rssi = 0;                   // received signal strength
int address = 0;                // the sending XBee 's address

Serial myPort;                  // The serial port

int fontSize = 18;              // size of the text on the screen
int lastReading = 0;            // value of the previous incoming byte

void setup () {
  size(400, 300);        // window size
```

»

Continued from previous page.

```
  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], fontSize);
  textFont(myFont);

  // get a list of the serial ports:
  println(Serial.list());
  // open the serial port attached to your XBee radio:
  XBee = new Serial(this, Serial.list()[0], 9600);
}

void draw() {
  // if you have new data and it's valid (>0), graph it:
  if ((rssi > 0 ) && (rssi != lastReading)) {
    // set the background:
    background(0);
    // set the bar height and width:
    int rectHeight = rssi;
    int rectWidth = 50;
    // draw the rect:
    stroke(23, 127, 255);
    fill (23, 127, 255);
    rect(width/2 - rectWidth, height-rectHeight, rectWidth, height);
    // write the number:
    text("XBee Radio Signal Strength test", 10, 20);
    text("From: " + hex(address), 10, 40);

    text ("RSSI: -" + rssi + " dBm", 10, 60);
    // save the current byte for next read:
    lastReading = rssi;
  }
}

void serialEvent(Serial XBee ) {
  // read a byte from the port:
  int thisByte = XBee .read();
  // if the byte = 0x7E, the value of a start byte, you have
  // a new packet:
  if (thisByte == 0x7E) {   // start byte
    // parse the previous packet if there's data:
    if (packet[2] > 0) {
      parseData(packet);
    }
    // reset the byte counter:
    byteCounter = 0;
  }
  // put the current byte into the packet at the current position:
  packet[byteCounter] = thisByte;
  //  increment the byte counter:
  byteCounter++;
}
```

Continued from opposite page.

```
/*
 Once you've got a packet, you need to extract the useful data.
 This method gets the address of the sender and RSSI.
 */
void parseData(int[] thisPacket) {

  // read the address -- a two-byte value, so you
  // add the two bytes as follows:
  address = thisPacket[5] + thisPacket[4] * 256;
  // get RSSI:
  rssi = thisPacket[6];
}
```

Radio signal strength is measured in decibel-milliwatts (dBm). You might wonder why the signal reads –65dBm. How can the signal strength be negative? The relationship between milliwatts of power and dBm is logarithmic. To get the dBm, take the log of the milliwatts. So, for example, if you receive 1 milliwatt of signal strength, you've got log 1 dBm. Log 1 = 0, so 1 mW = 0 dBm. When the power drops below 1 mW, the dBm drops below 0, like so: 0.5 mW = (log 0.5) dBm or –3.01 dBm. 0.25mW = (log 0.25) dBm, or –6.02 dBm. If logarithms confuse you, just remember that 0 dBm is the maximum transmission power, which means that signal strength is going to start at 0 dBm and go down from there. The minimum signal that the XBee radios can receive is –92 dBm. Bluetooth radios and Wi-Fi radios typically have a similar range of sensitivity. In a perfect world with no obstructions to create errors, relationship between signal strength and distance would be a logarithmic curve.

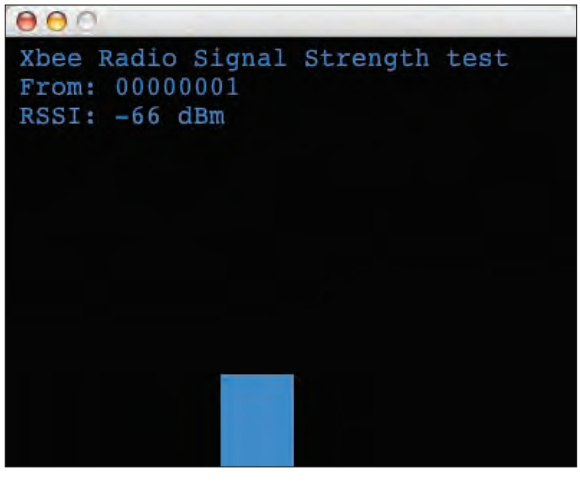


Figure 8-5
Output of the XBee RSSI test program.

Project 17

Reading Received Signal Strength Using Bluetooth Radios

The BlueRadios modules used in Chapters 2 and 6 can also give you an RSSI reading. To see this, the radio needs to be connected to another Bluetooth radio. The simplest way to see this is to pair your radio with your laptop as shown in Project #3 in Chapter 2.

Once you've done this, open a serial connection to the radio via Bluetooth, not via the radio's hardware serial interface. Once you're connected, drop out of data mode into command mode by typing the following (\r indicates that you should type a carriage return; press Enter or Return when you see \r):

```
+++AT\r
```

You'll get an OK prompt from the radio. Next type:

```
ATRSSI\r
```

The radio will respond like so:

```
OK
+0.6
```

As you move the radio closer to or further from your computer, type the ATRSSI command again. You'll see the signal strength change just as it did in the XBee example in the preceding project.

The Multipath Effect

The biggest source of error in distance ranging is what's called the multipath effect (see Figure 8-6). When electromagnetic waves radiate, they bounce off things. Your phone may receive multiple signals from a nearby cell tower if you're positioned near a large obstacle such as a building. The reflected waves off the building create "phantom" signals that look to the receiver as real as the original signal. This issue makes it impossible for the receiver to calculate the distance from the beacon accurately and causes degradation in the signal quality of mobile phone reception, as well as errors in locating the phones. For GPS receivers, multipath results in a much wider range of possible locations, as the error means that you can't calculate the position as accurately. It is possible to filter for the reflected signals, but not all radios incorporate such filtering.

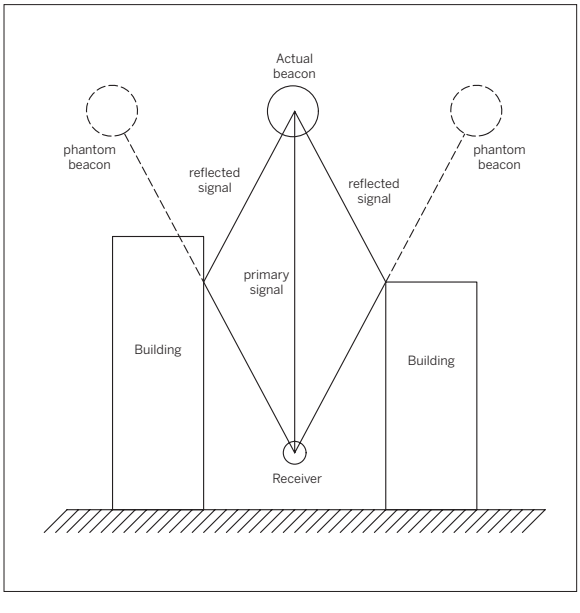


Figure 8-6
Multipath. Reflected radio waves create phantom beacons that the receiver can't tell from the real beacon, causing errors in calculating the distance based on signal strength.

Determining Position Through Trilateration

Distance ranging tells you how far away an object is from your measuring point in one dimension, but it doesn't define the whole position. The distance between your position and the target object determines a circle around your position (or a sphere, if you're measuring in three dimensions). Your object could be anywhere on that circle.

In order to locate it within a two- or three-dimensional space, though, you need to know more than distance. The most common way to do this is by measuring the distance from at least three points. This method is called trilateration. If you measure the object's distance from two points, you get two possible places it could be on a plane, as shown in Figure 8-7. When you add a third circle, you have one distinct point on the plane where your object could be. A similar method, triangulation, uses two known points and calculates the position using the distance between the two known points and the angles of the triangle formed by those points and the position you want to know.

The Global Positioning System uses trilateration to determine an object's position. GPS uses a network of geosynchronous satellites circling the globe. The position of each satellite can be calculated at any given moment. Each one is broadcasting its position, and GPS receivers pick up that broadcast. When a receiver has at least three satellites, it can determine a rough position. Most receivers use at least six satellite signals to calculate their position, in order to correct any errors. Cell phone location systems like Wireless E911 calculate a phone's approximate position in a similar fashion, by measuring the distance from multiple cell towers based on the time difference of arrival of signals from those towers.

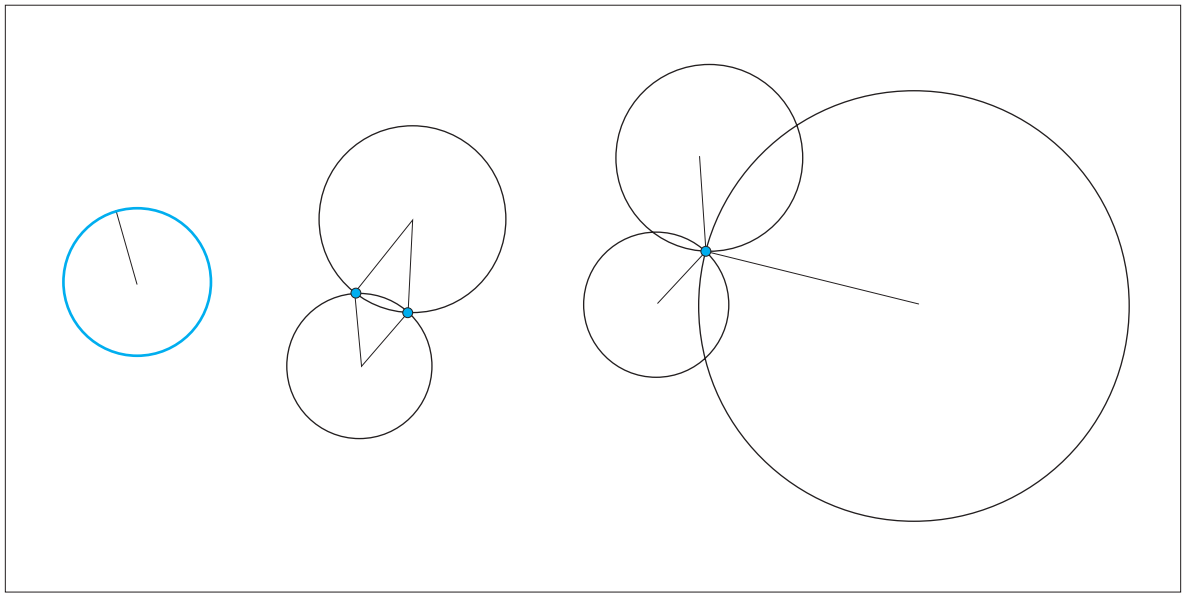


Figure 8-7
Trilateration on a two-dimensional plane. Knowing the distance from one point defines a circle of possible locations. Knowing the distance from two points narrows it to two possible points. Knowing the distance from three points determines a single point on the plane.

Project 18

Reading the GPS Serial Protocol

The good news is that if you're using GPS, you never have to do trilateration or triangulation calculations for yourself. GPS receivers do the work for you. They then give you the position in terms of latitude and longitude. There are several data protocols for GPS receivers, but the most widely used is the NMEA 0183 protocol established by the National Marine Electronics Association in the United States. Just about all receivers on the market output NMEA 0183, and usually one or two other protocols as well.

MATERIALS

- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
- **1 GPS receiver module** SparkFun part number GPS-00465 or Parallax (www.parallax.com) part number 28146 (the Parallax module, developed by Grand Design Studio, comes mounted on a breadboard-ready breakout board)
- **1 BlueSMiRF Bluetooth Modem module** from SparkFun
- **1 5V voltage regulator** Digi-Key part number LM7805CT-ND; Jameco part number 51262; SparkFun part number COM-00107

NMEA 0183 is a serial protocol that operates at 4800 bits per second, 8 data bits, no parity, 1 stop bit (4800-8-N-1). Most receivers send this data using either RS-232 or TTL serial levels. The receiver used for this example, an EM-406A SiRF III receiver available from SparkFun, sends NMEA data at 5V TTL levels. Figure 8-8 shows the module connected to BlueSMiRF radio, for testing. The data is sent back over Bluetooth to a personal computer running Processing. You can pair the BlueSMiRF to your personal computer using the instructions from Project #2 in Chapter 2, if it's not already paired.

Once your personal computer recognizes the Bluetooth radio, open a serial connection to it at 9600 bits per second. You're opening a serial connection over Bluetooth, not over USB as you've done with most of the projects in this book. The GPS module operates at 4800 bits per second, so you need to reset the Bluetooth radio's serial data rate in order to get data from the GPS module through it. If you do see any data, it'll be illegible, because the radio's serial connection to the GPS module is set to 9600 bps, and the GPS module's sending at 4800 bps. To change this, type: `+++\r` This command takes the radio out of data mode and puts it in command mode. The radio will respond: `OK`

Next, type:

```
ATSW20,20,0,0,1\r
ATMD\r
```

» This changes the data rate to 4800 bits per second

» This puts the radio back into data mode.

After that, you should see data in the NMEA protocol, like what you see here, below:

```
$GPGGA,155123.000,4043.8432,N,07359.7653,W,1,05,1.7,49.7,M,-34.2,M,,0000*5F
$GPGSA,A,3,10,24,29,02,26,,,,,,,,6.2,1.7,6.0*3C
$GPGSV,3,1,12,10,62,038,38,29,57,160,38,24,52,311,32,06,52,273,29*77
$GPGSV,3,2,12,26,43,175,39,02,40,106,38,07,40,294,36,15,30,301,*77
$GPGSV,3,3,12,21,14,298,,08,10,078,,27,10,051,,04,04,112,*7B
$GPRMC,155123.000,A,4043.8432,N,07359.7653,W,0.15,83.25,200407,,*28
$GPGGA,155124.000,4043.8432,N,07359.7653,W,1,05,1.7,49.8,M,-34.2,M,,0000*57
$GPGSA,A,3,10,24,29,02,26,,,,,,,,6.2,1.7,6.0*3C
$GPRMC,155124.000,A,4043.8432,N,07359.7653,W,0.15,79.50,200407,,*28
$GPGGA,155125.000,4043.8432,N,07359.7654,W,1,05,1.7,49.7,M,-34.2,M,,0000*5E
$GPGSA,A,3,10,24,29,02,26,,,,,,,,6.2,1.7,6.0*3C
$GPRMC,155125.000,A,4043.8432,N,07359.7654,W,0.10,11.88,200407,,*20
```

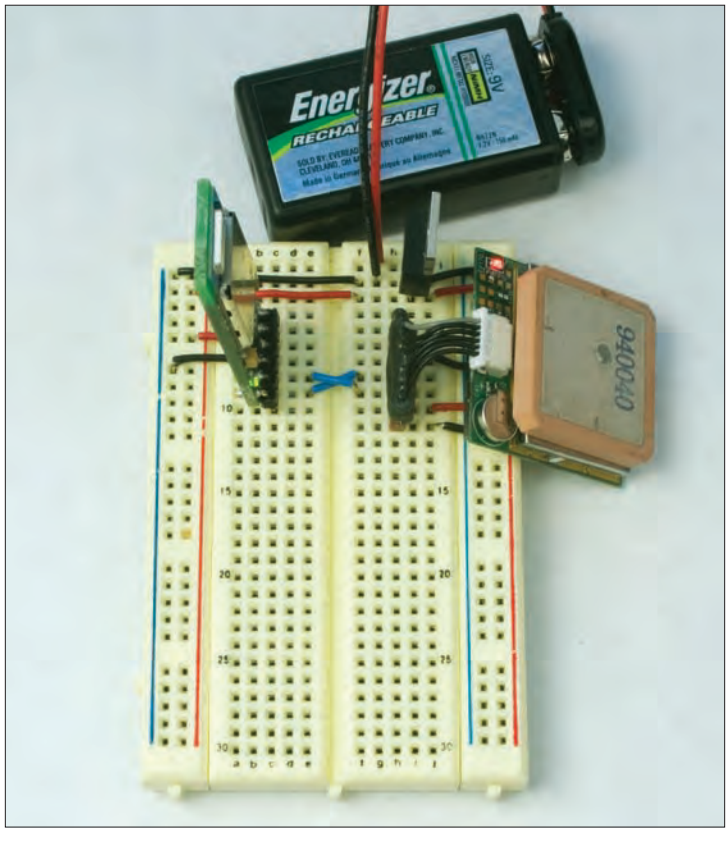
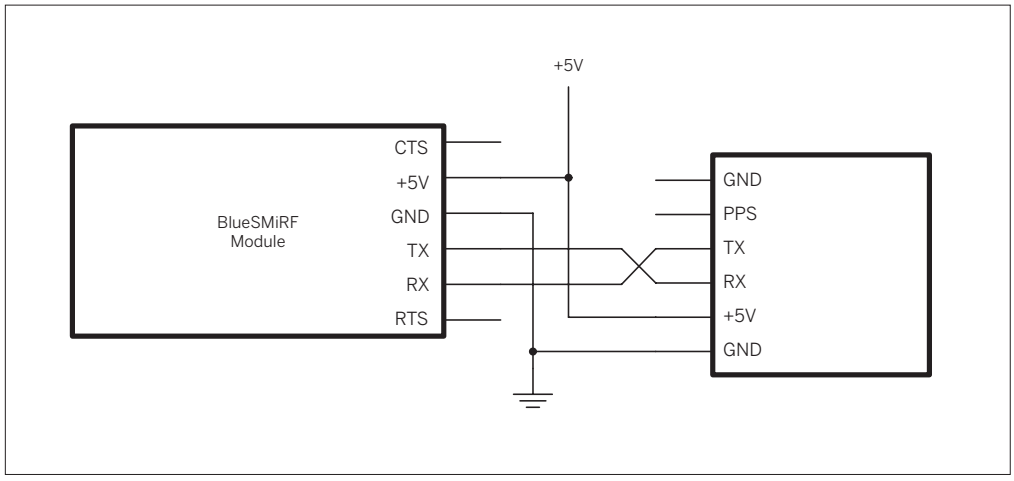


Figure 8-8
EM-406A GPS receiver attached to a Bluetooth radio. In order to get a real GPS signal, you'll have to go outside, so wireless data and a battery power source are handy.



There are several different types of sentences within the NMEA protocol. Each sentence serves a different function. Some tell you your position, some tell you about the satellites in view of the receiver, some deliver information about your course heading, and so on. Each sentence begins with a dollar sign ($) followed by five letters that identify the type of sentence. After that come each of the parameters of the sentence, separated by commas. An asterisk comes after the parameters, then a checksum, then a carriage return and a linefeed.

» Take a look at the $GPRMC sentence as an example:

```
$GPRMC,155125.000,A,4043.8432,N,07359.7654,W,0.10,11.88,200407,,*20
```

RMC stands for Recommended Minimum specifiC global navigation system satellite data. It gives the basic information almost any application might need. This sentence contains the information shown in the table.

Message identifier	$GPRMC
Time	155125.000 or 15:51:25 GMT
Status of the data (valid or not valid)	A = valid data (V = not valid)
Latitude	4043.8432 or 40°43.8432'
North/South Indicator	N = North (S = South)
Longitude	07359.7654 or 73°59.7654'
East/West indicator	W = West (E = East)
Speed over ground	0.10 knots
Course over ground	11.88° from north
Date	200407 or April 20, 2007
Magnetic Variation	none
Mode	none
Checksum (there is no comma before the checksum; magnetic variation would appear to the left of that final comma, and mode would appear to the right)	*20

Using the NMEA protocol in a program is just a matter of deciding which sentence gives you the information you need, reading the data in serially, and converting the data into values you can use. In most cases, the RMC sentence gives you all the data you need about position.

The Processing sketch shown next reads NMEA serial data in and parses out the time, date, latitude, longitude, and heading. It draws an arrow on the screen to indicate heading. The output looks like Figure 8-9. (Be sure to use the correct serial port when opening myPort; you may need to modify the code before you can run it.)

NOTE: Extra credit: Figure out where I was when I wrote this chapter.

Figure 8-9
The output of the Processing GPS parser.

Find It First, set up your global variables as usual:

```
/*
 GPS parser
 language: Processing

 This program takes in NMEA 0183 serial data and
 parses out the date, time, latitude, and longitude
 using the GPRMC sentence.
*/

// import the serial library:
import processing.serial.*;

Serial myPort;           // the serial port
float latitude = 0.0;    // the latitude reading in degrees
String northSouth;       // north or south?
float longitude = 0.0;   // the longitude reading in degrees
String eastWest;         // east or west?
float heading = 0.0;     // the heading in degrees

int hrs, mins, secs;     // time units
int thisDay,  thisMonth, thisYear;
```

» The setup() method sets the window size, defines the font and the drawing parameters, and opens the serial port.

```
void setup() {
  size(300, 300);        // window size

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 14);
  textFont(myFont);

  // settings for drawing arrow:
  noStroke();
  smooth();

  // list all the available serial ports:
  println(Serial.list());

  // I know that the first port in the serial list on my mac
  // is always my Keyspan adaptor, so I open Serial.list()[0].
  // Open whatever port you're using.
  myPort = new Serial(this, Serial.list()[0], 4800);

  // read bytes into a buffer until you get a carriage
  // return (ASCII 13):
  myPort.bufferUntil('\r');
}
```

The draw() method prints the readings in the window, and calls another method, drawArrow() to draw the arrow and circle.

```
void draw() {
  background(0);
  // make the text white:
  fill(255);

  // print the date and time from the GPS sentence:
  text(thisMonth+ "/"+ thisDay+ "/"+ thisYear , 50, 30);
  text(hrs+ ":"+ mins+ ":"+ secs + " GMT  ", 50, 50);
  // print the position from the GPS sentence:
  text(latitude + " " + northSouth + ", " +longitude +" "+ eastWest,
       50, 70);
  text("heading " + heading + " degrees", 50,90);

  // draw an arrow using the heading:
  drawArrow(heading);
}
```

The serialEvent() method gets any incoming data as usual, and passes it off to a method called parseString(). That method splits the incoming string into all the parts of the GPS sentence. If it's a GPRMC sentence, it passes it to a method, getRMC() to handle it. If you were writing a more universal parser, you'd write similar methods for each type of sentence.

```
void serialEvent(Serial myPort) {
  // read the serial buffer:
  String myString = myPort.readStringUntil('\n');

  // if you got any bytes other than the linefeed, parse it:
  if (myString != null) {
    parseString(myString);
  }
}

void parseString (String serialString) {
  // split the string at the commas:
  String items[] = (split(serialString, ','));

  // if the first item in the sentence is the identifier, parse the rest
  if (items[0].equals("$GPRMC")) {
    // get time, date, position, course, and speed
    getRMC(items);
  }
}
```

The getRMC() method converts the latitude, longitude, and other numerical parts of the sentence into numbers.

```
void getRMC(String[] data) {
  // move the items from the string into the variables:
  int time = int(data[1]);
  // first two digits of the time are hours:
  hrs = time/10000;
  // second two digits of the time are minutes:
  mins = (time%10000)/100;
  // last two digits of the time are seconds:
  secs = (time%100);
```

Continued from opposite page.

```
  // if you have a valid reading, parse the rest of it:
  if (data[2].equals("A")) {
    latitude = float(data[3])/100.0;
    northSouth = data[4];
    longitude = float(data[5])/100.0;
    eastWest = data[6];
    heading = float(data[8]);
    int date = int(data[9]);
    // last two digits of the date are year.  Add the century too:
    thisYear = date%100 + 2000;
    // second two digits of the date are month:
    thisMonth =  (date%10000)/100;
    // first two digits of the date are day:
    thisDay = date/10000;
  }
}
```

drawArrow() is called by the draw() method. It draws the arrow and the circle.

```
void drawArrow(float angle) {
  // move whatever you draw next so that (0,0) is centered on the screen:
  translate(width/2, height/2);

  // draw a circle in light blue:
  fill(80,200,230);
  ellipse(0,0,50,50);
  // make the arrow black:
  fill(0);
  // rotate using the heading:
  rotate(radians(angle));

  // draw the arrow.  center of the arrow is at (0,0):
  triangle(-10, 0, 0, -20, 10, 0);
  rect(-2,0, 4,20);
}
```

Determining Orientation

People have an innate ability to determine their orientation relative to the world around them, but objects don't. So orientation sensors are usually used for refining the position of objects rather than of people. In this section, you'll see two types of orientation sensors: a digital compass for determining heading relative to the Earth's magnetic field, and an accelerometer for determining orientation relative to the Earth's gravitational field. Using these two sensors, you can determine which way is north and which way is up.

Project 19

Determining Heading Using a Digital Compass

Calculating heading can be done using a compass if you are in a space that doesn't have a lot of magnetic interference. There are many digital compasses on the market. These acquire a heading by measuring the change in the earth's magnetic field along two axes, just as an analog compass does. Like analog compasses, they are subject to interference from other magnetic fields, including those generated by strong electrical induction.

MATERIALS

- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
- **1 Arduino module** or other microcontroller
- **2 4.7kΩ resistors**
- **1 digital compass, Devantech model CMPS03** available from Acroname Robotics, part number R117-COMPASS, or Robot Electronics CMPS03

This example uses a digital compass from Devantech, model CMPS03. It's available from www.acroname.com, and www.robot-electronics.co.uk. It measures its orientation using two magnetic field sensors placed at right angles to each other, and reports the results via two interfaces: a changing pulse width corresponding to the heading, or synchronous serial data sent over an I2C connection. You'll recognize some of the I2C methods used here from the SRF02 distance ranger example shown earlier. Both are made by the same company, and both use similar protocols.

The interface to the microcontroller is similar to that of the distance ranger. The Arduino module uses analog pin 4 as the SDA pin, and analog pin 5 as the SCL pin. Figure 8-10 shows the compass connected to an Arduino.

The compass operates on 5V. Its pins are as follows:

1. +5V
2. SCL – I2C serial clock
3. SDA – I2C serial data
4. PWM – Pulsewidth output. Pulsewidth is from 1 millisecond to 36.99 milliseconds, and each degree of compass heading is 100 microseconds of pulsewidth.
5. No connection.
6. Calibrate.
7. 50/60Hz select — Take this pin high or leave unconnected for 60Hz operation, take it low for 50Hz.
8. No connection
9. Ground

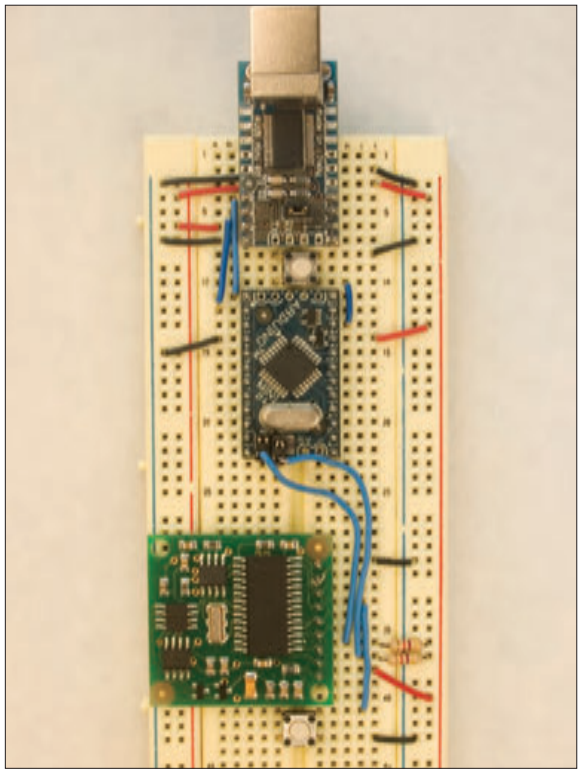
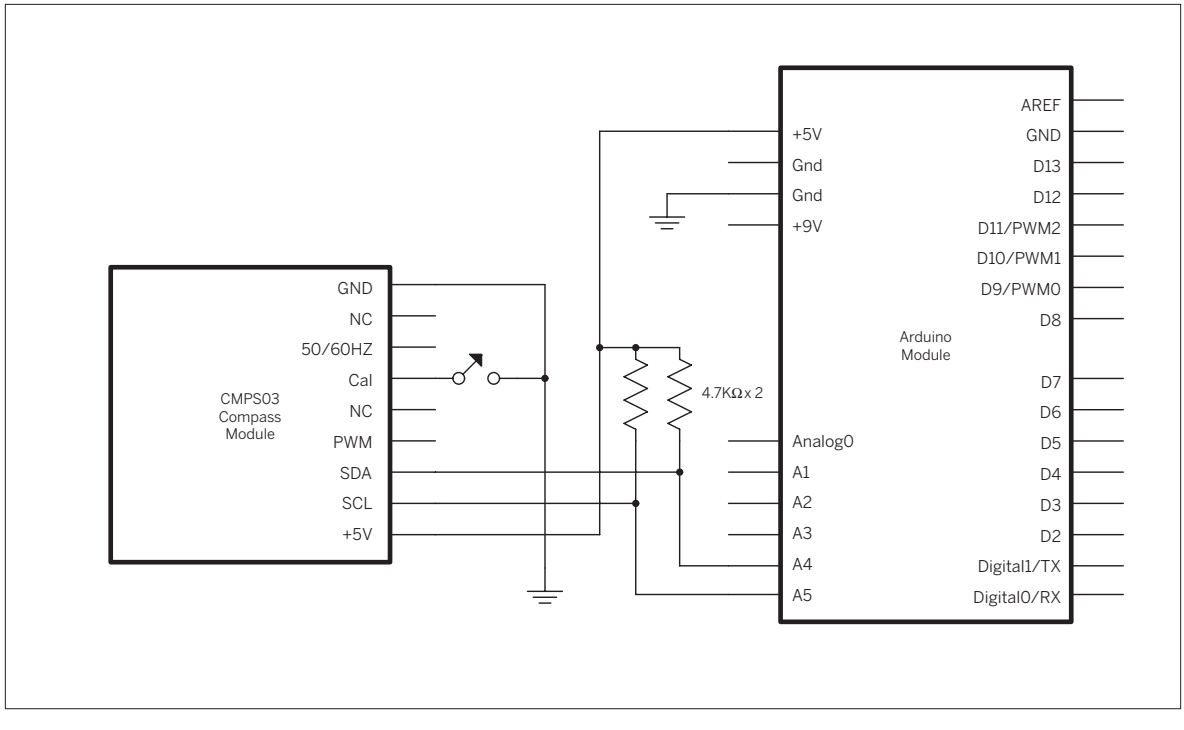


Figure 8-10
Devantech CMPS03 compass connected to an Arduino Mini 04 module. Female headers have been soldered to the Arduino Mini's analog 4-7 holes to make them easier to use. Note that the SDA and SCL lines need to have 4.7KΩ pull-up resistors connecting them to 5V.

NOTE: To calibrate the compass, take pin 6 low and rotate the compass slowly through 360 degrees on a flat, level surface. When calibrating the compass, you need to know the cardinal directions precisely. Get a magnetic needle compass and check properly. You should calibrate away from lots of electronic equipment and sources of magnetic energy (except the Earth). For example, in my office, needle compasses tend to point west-southwest, so I calibrate outside, powering the whole Arduino circuit from a battery. For more on calibrating the CMPS03, see www.robot-electronics.co.uk/htm/cmps_cal.shtml.



Try It This program uses the Wire library to communicate via I2C with the compass. There are no global variables, but before the setup() method, you have to include the library and define a couple of constants that are operational codes for the compass module.

```
/*
  CMPS03 compass reader
 language: Wiring/Arduino

 Reads data from a Devantech CMPS03 compass sensor.

 Sensor connections:
 SDA - Analog pin 4
 SCL - Analog pin 5
 */

// include Wire library to read and write I2C commands:
#include <Wire.h>

// the commands needed for the SRF sensors:
#define sensorAddress 0x60
// this is the memory register in the sensor that contains the result:
#define resultRegister 0x02
```

» The setup() method initializes the Wire and Serial libraries.

```
void setup()
{
  Wire.begin();        // start the I2C bus
  Serial.begin(9600);  // open the serial port
}
```

» The main loop calls a method called setRegister() to read from the compass' registers. Then it prints what it read. The compass needs about 68 milliseconds between readings, so the loop delays after the readings.

```
void loop()
{
  // send the command to read the result in inches:
  setRegister(sensorAddress, resultRegister);
  // read the result:
  int bearing = readData(sensorAddress, 2);
  // print it:
  Serial.print("bearing: ");
  Serial.print(bearing/10);
  Serial.println(" degrees");
  delay(70);  // wait before next reading
}
```

» The compass has memory registers and function registers. The compass heading is stored in a memory register. To read it, you send an initial transmission of the address of the register you want to read from, then you request however many bytes you want to read.

```
/*
  setRegister() tells the SRF sensor to change the address
  pointer position
 */
void setRegister(int address, int thisRegister) {
  Wire.beginTransmission(address);  // start I2C transmission
  // send address to read from:
  Wire.send(thisRegister);
  Wire.endTransmission();           // end I2C transmission
}
```

readData() makes the request for two bytes of data from the compass' memory register, waits until those bytes have been returned, reads them, and returns the result as a single integer.

```
/*
readData() returns a result from the SRF sensor
 */

int readData(int address, int numBytes) {
  int result = 0;        // the result is two bytes long

  // send I2C request for data:
  Wire.requestFrom(address, numBytes);
  // wait for two bytes to return:
  while (Wire.available() < 2 )   {
    // wait for result
  }
  // read the two bytes, and combine them into one int:
  result = Wire.receive() * 256;
  result = result + Wire.receive();
  // return the result:
  return result;
}
```

Project 20

Determining Attitude Using an Accelerometer

Compass heading is an excellent way to determine orientation if you're level with the Earth, but sometimes you need to know how you're tilted. In navigation terms, this is called your attitude, and there are two major aspects to it: roll and pitch. Roll refers to how you're tilted side-to-side. Pitch refers how you're tilted front-to-back. If you've ever used an analog compass, you know how important it is to control your roll and pitch in order to get an accurate reading. Rotation on the axis perpendicular to the horizon is called yaw, and for our purposes, it's easiest to measure with a compass.

MATERIALS

- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
- **1 Arduino module** or other microcontroller
- **1 Analog Devices ADXL320 accelerometer** SparkFun sells a module with this accelerometer mounted on a breakout board, part number SEN-00847

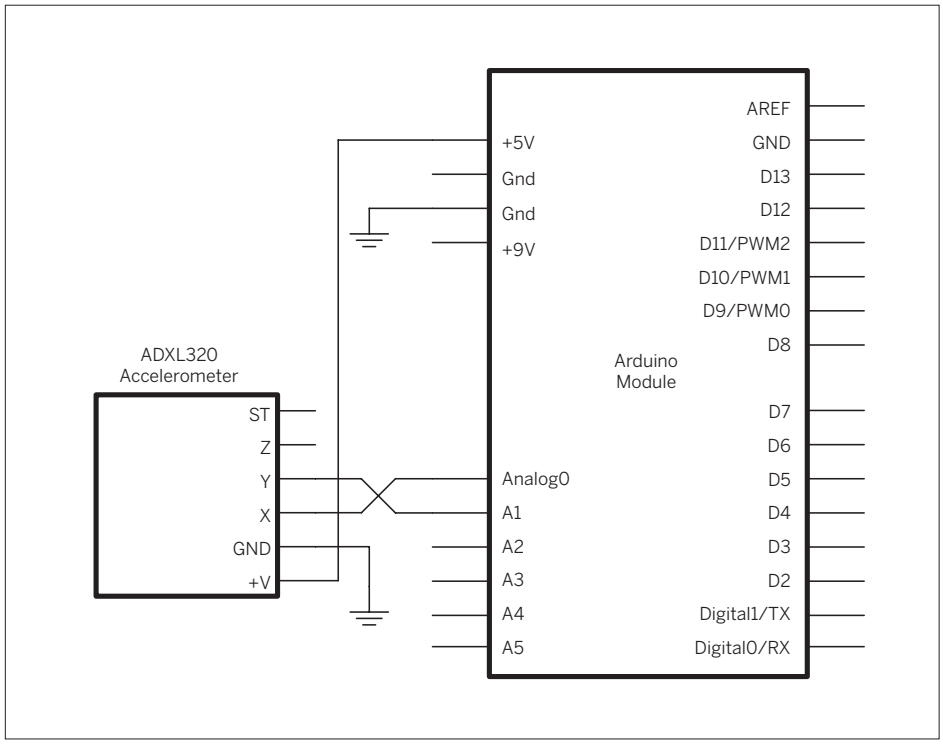
Measuring roll and pitch is relatively easy to do using an accelerometer. You used one of these already in Chapter 5, in the seesaw ping pong client. Accelerometers measure changing acceleration. At the center of an accelerometer is a tiny mass that's free to swing in one, two, or three dimensions. As the accelerometer tilts relative to the earth, the gravitational force exerted on the mass changes. Because force equals mass times acceleration, and because the mass of the accelerometer is constant, the change is read as a changing acceleration.

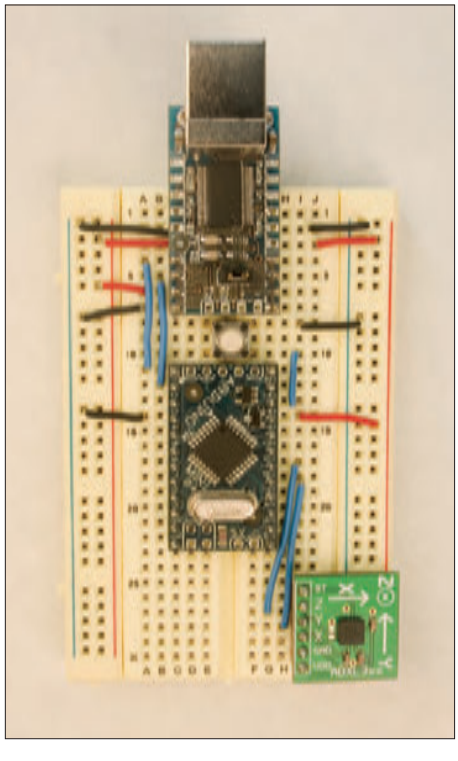
In the following example, you'll use an accelerometer to control the pitch and roll of a disk onscreen in Processing. The numeric values from the sensor are written on the disk as it tilts.

» Connect the accelerometer to the Arduino as shown in Figure 8-11. Then program it using the following code:

```
/*
 Accelerometer reader
 language: Wiring/Arduino

 Reads 2 axes of an accelerometer and sends the
 values out the serial port
 */

int accelerometer[2];     // variable to hold the accelerometer values

void setup() {
  // open serial port:
  Serial.begin(9600);
  // send out some initial data:
  Serial.println("0,0,");
}
```

»





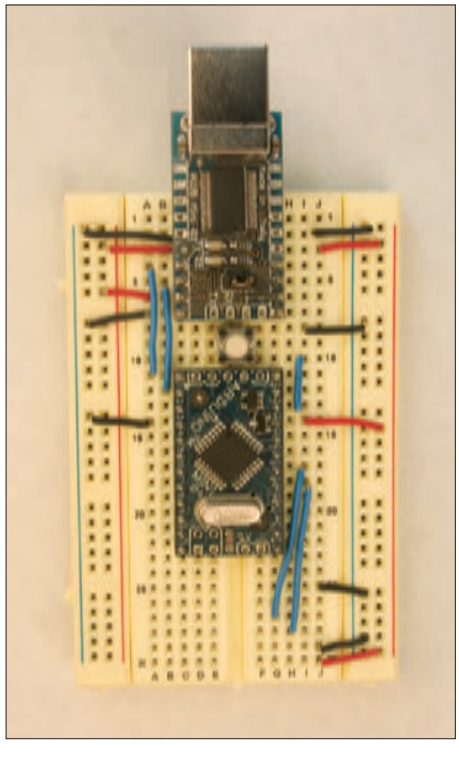


Figure 8-11
ADXL320 accelerometer connected to an Arduino Mini 04 module. The detail photo shows the board with the accelerometer removed to show the wiring underneath.

Continued from previous page.

```
void loop() {
  // read 2 channels of the accelerometer:
  for (int i = 0; i < 2; i++) {
    accelerometer[i] = analogRead(i);
    // delay to allow analog-to-digital converter to settle:
    delay(10);
  }

  // if there's serial data in, print sensor values out:
  if (Serial.available() > 0) {
    // read incoming data to clear serial input buffer:
    int inByte = Serial.read();
    for (int i = 0; i <  2; i++) {
      // values as ASCII strings:
      Serial.print(accelerometer[i], DEC);
      // print commas between values:
      Serial.print(",");
    }
    // print \r and \n after values are sent:
    Serial.println();
  }
}
```

Connect It This sketch uses a call-and-response serial method, so you won't see any data coming out unless you send it a character serially. You can test it using your favorite serial terminal program. With the serial port open in the serial terminal program, type any key to make the microcontroller send a response. Once you've got it working, run the following sketch in Processing, making sure that the serial port opened by the program matches the one to which your microcontroller is connected:

```
/*
  accelerometer tilt
  language: Processing

  Takes the values from an accelerometer
  and uses it to set the attitudeof a disk on the screen.
 */

import processing.serial.*;      // import the serial lib

int graphPosition = 0;           // horizontal position of the graph
int[] vals = new int[2];         // raw values from the sensor
int[] maximum = new int[2];      // maximum value sensed
int[] minimum = new int[2];      // minimum value sensed
int[] range = new int[2];        // total range sensed
float[] attitude = new float[2]; // the tilt values
float position;                  // position to translate to

Serial myPort;                   // the serial port
boolean madeContact = false;     // whether there's been serial data sent in

void setup () {
  // draw the window:
  size(400, 400, P3D);

  // set the background color:
```

Continued from opposite page.

```
  background(0);
  // set the maximum and minimum values:
  for (int i = 0; i < 2; i++) {
    maximum[i] = 600;
    minimum[i] = 200;
    // calculate the total current range:
    range[i] = maximum[i] - minimum[i];
  }
  position = width/2;  // calculate position

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 18);
  textFont(myFont);

  // list all the available serial ports:
  println(Serial.list());
  // Open whatever port you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);
  // generate a serial event only when you get a return char:
  myPort.bufferUntil('\r');

  // set the fill color:
  fill(90,250,250);
}
```

The draw() method just refreshes the screen in the window, as usual. It calls a method, setAttitude(), to calculate the tilt of the plane. Then it calls a method called tilt() to actually tilt the plane.

```
void draw () {
  // clear the screen:
  background(0);
  // print the values:
  text(vals[0] + " " + vals[1], -30, 10);

  // if you've never gotten a string from the microcontroller,
  // keep sending carriage returns to prompt for one:
  if (madeContact == false) {
    myPort.write('\r');
  }
  setAttitude(); // set the attitude
  tilt();        // draw the plane
}
```

The 3D system in Processing works on rotations from zero to 2*PI. setAttitude() converts the accelerometer readings into that range, so the values can be used to set the tilt of the plane.

```
void setAttitude() {
  for (int i = 0; i < 2; i++) {
    // calculate the current attitude as a percentage of 2*PI,
    // based on the current range:
    attitude[i] = (2*PI) * float(vals[i] - minimum[i]) /float(range[i]);
  }
}
```

» The tilt() method uses Processing's translate() and rotate() methods to move and rotate the plane of the disc to correspond with the accelerometer's movement.

```
void tilt() {
  // translate from origin to center:
  translate(position, position, position);

  // X is front-to-back:
  rotateX(-attitude[1]);
  // Y is left-to-right:
  rotateY(-attitude[0] - PI/2);

  // set the fill color:
  fill(90,250,250);
  // draw the rect:
  ellipse(0, 0, width/4, width/4);
  // change the fill color:
  fill(0);
  // draw some text so you can tell front from back:
  // print the values:
  text(vals[0] + " " + vals[1], -30, 10,1);

}
```

» The serialEvent() method reads all the incoming serial bytes and parses them as comma-separated ASCII values just as you did in Monski pong in Chapter 2.

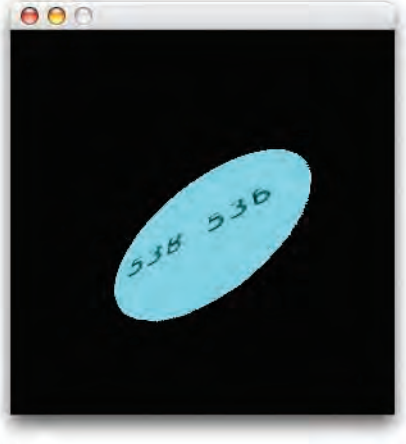


Figure 8-12
The output of the Processing accelerometer sketch.

```
// The serialEvent method is run automatically by the Processing applet
// whenever the buffer reaches the byte value set in the bufferUntil()
// method in the setup():

void serialEvent(Serial myPort) {
  // if serialEvent occurs at all, contact with the microcontroller
  // has been made:
  madeContact = true;
  // read the serial buffer:
  String myString = myPort.readStringUntil('\n');

  // if you got any bytes other than the linefeed:
  if (myString != null) {
    myString = trim(myString);
    // split the string at the commas
    //and convert the sections into integers:
    int sensors[] = int(split(myString, ','));
    // if you received all the sensor strings, use them:
    if (sensors.length >= 2) {
      vals[0] = sensors[0];
      vals[1] = sensors[1];

      // send out the serial port to ask for data:
      myPort.write('\r');
    }
  }
}
```

Conclusion

When you start to develop projects that use location systems, you usually find that less is more. It's not unusual to start a project thinking you need to know position, distance, and orientation, then pare away systems as you develop the project. The physical limitations of the things you build and the spaces you build them in solve many problems for you.

This effect, combined with your users' innate ability to locate and orient themselves, makes your job much easier. Before you start to solve all problems in code or electronics, put yourself physically in the place you're building for, and do what you intend your users to do. You'll learn a lot about your project, and save yourself time, aggravation, and money.

The examples in this chapter are all focused on a solitary person or object. As soon as you introduce multiple participants, location and identification become more tightly connected, because you need to know whose signal is coming from a given location, or what location a given speaker is at. In the next chapter, you'll see methods crossing the line from physical identity to network identity.

X

9

MAKE: PROJECTS

Identification

In the previous chapters, you've assumed that identity equals address. Once you knew a device's address on the network, you started talking. Think about how disastrous this would be if you used this formula in everyday life: you pick up the phone, dial a number, and just start talking. What if you dialed the wrong number? What if someone other than the person you expected answers the phone?

Networked objects mark the boundaries of networks, but not of the communications that travel across them. We use these devices to send messages to other people. The network identity of the device and the physical identity of the person are two different things. Physical identity generally equates to presence (is it near me?) or address (where is it?), but also takes into consideration network capabilities of the device and the state it's in when you contact it. In this chapter, you'll learn some methods for giving physical objects network identities. You'll also learn ways that devices on a network can learn each other's capabilities though the messages they send and the protocols they use.

Sniff, a toy for sight-impaired children, by Sara Johansson.
The dog's nose contains an RFID reader. When he detects RFID-tagged objects, he gives sound and tactile feedback, a unique response for each object. Designed by Sara Johansson, a student in the Tangible Interaction course at the Oslo School of Architecture and Design, under the instruction of tutors Timo Arnall and Mosse Sjaastad. *Photo courtesy of Sara Johansson.*

Physical Identification

The process of identifying physical objects is such a fundamental part of our experience that we seldom think about how we do it. We use our senses, of course: we look at, feel, pick up, shake and listen to, smell, and taste objects until we have a reference for them — then we give them a label. The whole process relies on some pretty sophisticated work by our brains and bodies, and anyone who's ever dabbled in computer vision or artificial intelligence in general can tell you that teaching a computer to recognize physical objects is no small feat. Just as it's easier to determine location by having a human being narrow it down for you, it's easier to distinguish objects computationally if you can limit the field, and if you can label the important objects.

Just as we identify things using information from our senses, so do computers. They can identify physical objects only by using information from their sensors. Two of the best-known digital identification techniques are optical recognition and radio frequency identification, or RFID. Optical recognition can take many forms, from video color tracking and shape recognition to the ubiquitous bar code. Once an object has been recognized by a computer, the computer can give it an address on the network.

The network identity of a physical object can be centrally assigned and universally available, or it can be provisional. It can be used only by a small subset of devices on a larger network or used only for a short time. RFID is an interesting case in point. The RFID tag pasted on the side of a book may seem like a universal marker, but what it means depends on who reads it. The owner of a store may assign that tag's number a place in his inventory, but to the consumer who buys it, it means nothing unless she has a tool to read it and a database in which to categorize it. She has no way of knowing what the number meant to the store owner unless she has access to his database. Perhaps he linked that ID tag number to the book's title, or to the date on which it arrived in the store. Once it leaves the store, he may delete it from his database, so it loses all meaning to him. The consumer, on the other hand, may link it to entirely different data in her own database, or she may choose to ignore it entirely, relying on other means to identify it. In other words, there is no central database linking RFID tags and the things they're attached to, or who's possessed them.

Like locations, identities become more uniquely descriptive as the context they describe becomes larger. For example, knowing that my name is Tom doesn't give you much to go on. Knowing my last name narrows it down some more, but how effective that is depends on where you're looking. In the United States, there are dozens of Tom Igoes. In New York, there are at least three. When you need a unique identifier, you might choose a universal label, like using my Social Security number, or you might choose a provisional label, like calling me "Frank's son Tom." Which you choose depends on your needs in a given situation. Likewise, you may choose to identify physical objects on a network using universal identifiers, or you might choose to use provisional labels in a given temporary situation.

The capabilities assigned to an identifier can be fluid as well. Taking the RFID example again: in the store, a given tag's number might be enough to set off alarms at the entrance gates or to cause a cash register to add a price to your total purchase. In another store, that same tag might be assigned no capabilities at all, even if it's using the same protocol as other tags in the store. Confusion can set in when different contexts use similar identifiers. Have you ever left a store with a purchase and tripped the alarm, only to be waved on by the clerk who forgot to deactivate the tag on your purchase? Try walking into a Barnes & Noble bookstore with jeans you just bought at a Gap store and you're likely to trip the alarms because the two companies use the same RFID tags, but don't always set their security systems to filter out tags that are not in their inventory.

Video color recognition in Processing, using the code in Project 21. This simple sketch works well with vibrantly pink monkeys.

Video Identification

All video identification relies on the same basic method: the computer reads a camera's image and stores it as a two-dimensional array of pixels. Each pixel has a characteristic brightness and color that can be measured using any one of a number of palettes: red-green-blue is a common scheme for video and screen-based applications, as is hue-saturation-value. Cyan-magenta-yellow-black is common in print applications. The properties of the pixels, taken as a group, form patterns of color, brightness, and shape. When those patterns resemble other patterns in the computer's memory, it can identify those patterns as objects.

Color Recognition

Recognizing objects by color is a relatively simple process, if you know that the color you're looking for is unique in the camera's image. This technique is used in film and television production to make superheroes fly. The actor is filmed against a screen of a unique color, usually green, as green isn't a natural color for human skin. Then the pixels of that color are removed, and the image is combined with a background image.

Color identification can be an effective way to track physical objects in a controlled environment. Assuming that you've got a limited number of objects in the camera's view, and each object's color is unique and doesn't change as the lighting conditions change, you can identify each object reasonably well. Even slight changes in lighting can change the color of a pixel, however, so lighting conditions need to be tightly controlled, as the following project illustrates.

X

Project 21

Color Recognition Using a Webcam

In this project, you'll get a firsthand look at how computer vision works. The Processing sketch shown here uses a video camera to generate a digital image, looks for pixels of a specific color, and then marks them on the copy of the image that it displays onscreen for you. Processing has a video library that enables you to capture the image from a webcam attached to your computer and manipulate the pixels.

MATERIALS

- **Personal computer with USB or FireWire port**
- **USB or FireWire webcam**
- **Colored objects**

The following Processing sketch is an example of color tracking using the video library. To use this, you'll need to have a camera attached to your computer, and have the drivers installed. The one you used in Chapter 3 for the cat camera should do the job fine. You'll also need some small colored objects. Stickers or toy balls can work well.

First, import the video library and write a program to read the camera. Every time a new frame of video is available, the video library generates a captureEvent. You can use this to read the video, then paint it to the stage:

Run this sketch and you should see yourself onscreen.

```
/*
    Color Sensor
    Language: Processing
*/
import processing.video.*;

Capture myCamera;                    // instance of the Capture class

void setup()
{
  // set window size:
  size(320, 240);

  // List all available capture devices. Macintoshes generally
  // identify three cameras, and the first or second is the built-in
  // iSight of the laptops.  Windows machines need a webcam
  // installed before you can run this program.
  println(Capture.list());

  // capture from the second device in the list (in my case,
  // my iSight).
  //
  // change this to match your own camera:
  String myCam = Capture.list()[1];
  myCamera = new Capture(this, myCam, width, height, 30);
}
```

Continued from opposite page.

```
void draw(){
  // draw the current frame of the camera on the screen:
  image(myCamera, 0, 0);
}

void captureEvent(Capture myCamera) {
  // read the myCamera and update the pixel array:
  myCamera.read();
}
```

Now, add a method to capture the color of a pixel at the mouse location. First, add two new global variables at the top of the program for the target color and the pixel location where you find it:

```
color targetColor = color(255,0,0);    // the initial color to find
int[] matchingPixel = new int[2];      // matching pixel's coordinate
```

Then add the following mouseReleased() method at the end of the sketch:

```
/* when the mouse is clicked, capture the
 color of the pixel at the mouse location
 to use as the tracking color:
 */
void mouseReleased() {
  // get the color of the mouse position's pixel:
  targetColor = myCamera.pixels[mouseY*width+mouseX];
  // get the pixel location
  matchingPixel[0] = mouseX;
  matchingPixel[1] = mouseY;
}
```

Then add two lines of code at the end of the draw() method to draw the ball:

When you run the sketch this time, you can click anywhere on the image and you'll get a dot at that location matching the color of the video at the spot.

```
  // draw a dot at the matchingPixel:
  fill(targetColor);
  ellipse(matchingPixel[0], matchingPixel[1], 10, 10);
```

To find other colors matching this color, you have to scan through all the pixels in the image to see which one's red, blue, and green matches yours. To do this, add a method called findColor() that takes the target color as a parameter, and returns the pixel matching that color as a return value:

```
int[] findColor(color thisColor) {
  // initialize the matching position with impossible numbers:
  int[] bestPixelYet =  {
    -1,-1  };
  // intialize the smallest acceptable color difference:
  float smallestDifference = 1000.0;

  // scan over the pixels  to look for a pixel
  // that matches the target color:
  for(int row=0; row<height; row++) {
```

Continued from previous page.

```
    for(int column=0; column<width; column++) { //for each column
      //get the color of this pixel
      // find pixel in linear array using formula:
      // pos = row*rowWidth+column
      color pixelColor = myCamera.pixels[row*width+column];

      // determine the difference between this pixel's color
      // and the target color:
      float diff = abs(red(targetColor) - red(pixelColor)) +
        abs(green(targetColor) - green(pixelColor)) +
        abs(blue(targetColor) - blue(pixelColor))/3;

      // if this is closest to our target color, take note of it:
      if (diff<= smallestDifference){
        smallestDifference = diff;
        // save the position so you can return it:
        bestPixelYet[0] = row;
        bestPixelYet[1] = column;
      }
    }
  }
  return bestPixelYet;
}
```

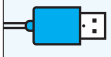
Add these lines to the end of captureEvent():

```
  // look for a pixel matching the target color
  matchingPixel = findColor(targetColor);
```

For the sketch in its entirety, see Appendix C.

As you can see when you run it, it's not the most robust color tracker! You can get it to be more precise by controlling the image and the lighting very carefully. Day-Glo colors under ultraviolet fluorescent lighting tend to be the easiest to track, but they lock you into a very specific visual aesthetic. Objects that produce their own light are easier to track, especially if you put a filter on the camera to block out stray light. A black piece of 35mm film negative (if you can still find 35mm film!) works well as a visible light filter, blocking most everything but infrared light. Two polarizers, placed so that their polarizing axes are perpendicular, also work well. Infrared LEDs track very well through this kind of filter, as do incandescent flashlight lamps. Regular LEDs don't always work well as color tracking objects, unless they're relatively dim. Brighter LEDs tend to show up as white in a camera image because their brightness overwhelms the camera.

Color recognition doesn't have to be done with a camera. There are color sensors that can do the same job. Parallax (www.parallax.com) sells a color sensor, the TAOS TCS230. This sensor contains four photodiodes, three of which are covered with color filters, and the fourth of which is not, so it can read red, green, blue, and white light. It outputs the intensity of all four channels as a changing pulse width.



Challenges of Identifying Physical Tokens

Designer Durrell Bishop's marble telephone answering machine is an excellent example of the challenges of identifying physical tokens. With every new message the machine receives, it drops a marble into a tray on the front of the machine. The listener hears the messages played back by placing a marble on the machine's "play" tray. Messages are erased and the marbles recycled when the marble is dropped back into the machine's hopper. Marbles become physical tokens representing the messages, making it very easy to tell at a glance how many messages there are.

Bishop tried many different methods to reliably identify and categorize physical tokens representing the messages:

> "I first made a working version with a motor and large screw (like a vending machine delivery mechanism), with pieces of paper tickets hung on the screw and had different color gray levels on the back. When it got a new message the machine read the next gray before it rotated once and dropped the ticket.
>
> It was a bit painful so I bought beads and stuffed resistors in to the hole which was capped (soldered) with sticky backed copper tape. When I went to Apple and worked with Jonathan Cohen we built a properly hacked version for the Mac with networked bar codes.
>
> Later again with Jonathan but this time at Interval Research, we used the Dallas ID chips."

Color by itself isn't enough to give you identity in most cases, but there are ways in which you can design a system to use color as a marker of physical identity. However, it has its limitations. In order to tell the marbles apart, Bishop could have used color recognition to read the marbles, but that would limit the design in at least two ways. First, there would be no way to tell the difference between multiple marbles of the same color. If, for example, he wanted to use color to identify the different people who received messages on the same answering machine, there would then be no way to tell the difference between multiple messages for each person. Second, the system would be limited by the number of colors between which the color recognition can reliably differentiate.

Shape and Pattern Recognition

Recognizing a color is relatively simple computationally, but recognizing a physical object is more challenging. To do this, you need to know the two-dimensional geometry of the object from every angle, so that you can compare any view you get of the object.

A computer can't actually "see" in three dimensions using a single camera. The view it has of any object is just a two-dimensional shadow. Furthermore, it has no way of distinguishing one object from another in the camera view without some visual reference. The computer has no concept of a physical object. It can only compare patterns. It can rotate the view, stretch it, and do all kinds of mathematical transformations on the pixel array, but the computer doesn't understand an object as a discrete entity the same way a human does. It just matches patterns. You could use two or more cameras to get a stereoscopic view, or write an AI program to give the computer a concept of a physical object, but a simpler solution is usually to restrict the view to a single two-dimensional plane, and to simplify the pattern. This is where bar codes come in handy.

Bar Code Recognition

A bar code (see Figure 9-1) is simply a pattern of dark and light lines or cells used to encode an alphanumeric string. A computer reads a bar code by scanning the image and interpreting the widths of the light and dark bands as zeroes or ones. This scanning can be done using a camera, or even a single photodiode, if the bar code can be passed over the photodiode at a constant speed. Many handheld bar code scanners work by having the user run a wand with an LED and a photodiode in the tip over the bar code, and reading the pattern of light and dark that the photodiode detects.

Figure 9-1
A one-dimensional bar code. This is the ISBN bar code for this book.

The best known bar code application is the Universal Product Code, or UPC, used by nearly every major manufacturer on the planet to label goods. There are many dozen different bar code symbologies, used for a wide range of applications. POSTNET is used by the U.S. Postal Service to automate mail sorting. European Article Numbering, or EAN, and Japanese Article Numbering, or JAN, are supersets of the UPC system developed to facilitate international exchange of goods. Each symbology represents a different mapping of bars to alphanumeric characters. The symbologies are not interchangeable, so you can't properly interpret a POSTNET bar code if you're using an EAN interpreter. This means that either you have to write a more comprehensive piece of software that can interpret several symbologies, or you have to know the symbology in advance if you want to get a reliable reading. There are numerous software libraries for generating bar codes, and even several bar code fonts for some of the more popular symbologies.

Bar codes such as the one shown earlier are called one-dimensional bar codes because the scanner or camera needs to read the image only along one axis. There are also two-dimensional bar codes that encode data in a two-dimensional matrix for more information density. As with one-dimensional bar codes, there are a variety of symbologies. Figure 9-2 shows a typical two-dimensional bar code. This type of code, the QR (Quick Response) code, was originally created in Japan for tracking vehicle parts, but it's since become popular for all kinds of product labeling. The inclusion of software to read these tags on many camera phones in Japan has made the tags more popular. The following example uses an open source Java library to read QR codes in Processing. The Java library used here was originally developed for use with the Java 2 Mobile Edition (J2ME) on mobile phones.

Figure 9-2
A two-dimensional bar code.

ConQwest, designed for Qwest Wireless in 2003, by Area/Code
www.playareacode.com

The first ever use of semacode, 2D bar codes scanned by phonecams. A city-wide treasure-hunt designed for high school students, players went through the city "shooting treasure" with Qwest phonecams and moving their totem pieces to capture territory. Online, a web site showed the players' locations and game progress, turning it into a spectacular audience-facing event.
Photo courtesy Area/Code and Kevin Slavin

Project 22

2D Bar Code Recognition Using a Webcam

In this project, you'll generate some two-dimensional bar codes from text using an online QR Code generator. Then you'll decode your tags using a camera and a computer. Once this works, try decoding the QR Code illustrations in this book.

MATERIALS

- **Personal computer with USB or FireWire port**
- **USB or FireWire Webcam**
- **Printer**

This sketch reads QR Codes using a camera attached to a personal computer. The video component is very similar to the color tracking example earlier. Before you start on the sketch, though, you'll need some QR Codes to read. Fortunately, there are a number of QR Code generators available online. Just type the term into a search engine and see how many pop up. There's a good one at qrcode.kaywa.com, from which you can generate URLs, phone numbers, or plain text. The more text you enter, the larger the symbol is. Generate a few codes and print them out for use later. Save them as **.jpg** or **.png** files as well, because you'll need them for the sketch.

To run this sketch, you'll need to download the pqrcode library for Processing by Daniel Shiffman, It's based on the qrcode library from qrcode.sourceforge.jp. You can download the pqrcode library from www.shiffman.net/p5/pqrcode. Unzip it, and you'll get a directory called **pqrcode/**. Drop it into the **libraries/** subdirectory of your Processing application directory and restart Processing. Make a new sketch, and within the sketch's directory, make a subdirectory called **data/** and put the **.jpg** or **.png** files of the QR Codes that you generated earlier there. Now you're ready to begin writing the sketch.

In the setup() for this sketch, you'll import the pqrcode and video libraries, initialize a few global variables, and establish a text font for printing on the screen:

```
/*
    QRCode reader
    Language: Processing
*/
import processing.video.*;
import pqrcode.*;

Capture video;                              // video capture object
String statusMsg = "Waiting for an image"; // a string for messages

// decoder object from prdecoder library
Decoder decoder;

// make sure to generate your own image here:
String testImageName = "qrcode.png";

void setup() {
  size(400, 320);
  video = new Capture(this, width, height-20, 30);
  // create a decoder object:
  decoder = new Decoder(this);

  // create a font with a font available to the system:
  PFont myFont = createFont(PFont.list()[2], 14);
  textFont(myFont);
}
```

The draw() method draws the camera image and prints a status message to the screen, and the captureEvent() updates the camera as in the previous project. Once you've entered this much, you can run the sketch to make sure that you got the libraries in the right places, and that the video works:

```
void draw() {
  background(0);

  // Display video
  image(video, 0, 0);
  // Display status
  text(statusMsg, 10, height-4);
}

void captureEvent(Capture video) {
  video.read();
}
```

The pqrcode library has a method called decodeImage(). In order to use it, you pass it an image. You'll do this in the keyReleased() method. A switch statement checks to see which key has been pressed. If you type `f`, it passes the decoder a file called **qrcode.png** from the **data/** subdirectory. If you press the spacebar, it passes the camera image. If you type `s`, it brings up a camera settings dialog box:

```
void keyReleased() {
  String code = "";
  // Depending on which key is hit, do different things:
  switch (key) {
  case ' ':        // Spacebar takes a picture and tests it:
    // copy it to the PImage savedFrame:
    PImage savedFrame = createImage(video.width,video.height,RGB);
    savedFrame.copy(video, 0,0,video.width,video.height,0,0,
      video.width,video.height);
    savedFrame.updatePixels();
    // Decode savedFrame
    decoder.decodeImage(savedFrame);
    break;
  case 'f':    // f runs a test on a file
    PImage preservedFrame = loadImage(testImageName);
    // Decode file
    decoder.decodeImage(preservedFrame);
    break;
  case 's':      // s opens the settings for this capture device:
    video.settings();
    break;
  }
}
```

Once you've given the decoder an image, you wait. When it's decoded the image, it generates a decoderEvent(), and you can read the tag's ID using the getDecodedString() method:

```
// When the decoder object finishes
// this method will be invoked.
void decoderEvent(Decoder decoder) {
  statusMsg = decoder.getDecodedString();
}
```

Finally, add an if() statement to the end of the draw() method to update the user as to the status of an image being decoded:

```
  // If we are currently decoding
  if (decoder.decoding()) {
    // Display the image being decoded
    PImage show = decoder.getImage();
```

Continued from opposite page.

```
    image(show,0,0,show.width/4,show.height/4);
    statusMsg = "Decoding image";
    // fancy code for drawing dots as a progress bar:
    for (int i = 0; i < (frameCount/2) % 10; i++)
    {
      statusMsg += ".";
    }
  }
```

When you run this, notice how the **.jpg** or **.png** images scan much more reliably than the camera images. The distortion from the analog-to-digital conversion through the camera causes many errors. This error is made worse by poor optics or low-end camera imaging chips in mobile phones and web cams. Even with a good lens, if the code to be scanned isn't centered, the distortion at the edge of an image can throw off the pattern-recognition routine. You can improve the reliability of the scan by guiding the user to center the tag before taking an image. Even simple graphic hints like putting crop marks around the tag, as shown in Figure 9-3, can help. When you do this, users framing the image tend to frame to the crop marks, which ensures more space around the code, and a better scan. Methods like this help with any optical pattern recognition through a camera, whether it's one- or two-dimensional bar codess, or another type of pattern altogether.

Figure 9-3
A two-dimensional bar code (a QR Code, to be specific) with crop marks around it. The image parsers won't read the crop marks, but they help users center the tag for image capture.

Optical recognition forces one other limitation on you besides various limitations mentioned earlier: you have to be able to see the bar code. By now most of the world is familiar with bar codes, because they decorate everything we buy or ship. This limitation is not only aesthetic. If you've ever turned a box over and over looking for the bar code to scan, you know that it's also a functional limitation. A system that allowed for machine recognition of physical objects, but didn't rely on a line of sight to the identifying tag would be an improvement. This is one of the main reasons that RFID is beginning to supersede bar codes in inventory control and other ID applications.

Radio Frequency Identification

Like bar code recognition, RFID relies on tagging objects in order to identify them. Unlike bar codes, however, RFID tags don't need to be visible to be read. An RFID reader sends out a short-range radio signal, which is picked up by an RFID tag. The tag then transmits back a short string of data. Depending on the size and sensitivity of the reader's antenna and the strength of the transmission, the tag can be several feet away from the reader, enclosed in a book, box, or item of clothing. In fact, some large clothing manufacturers are now sewing RFID tags into their merchandise, to be removed by the customer.

There are two types of RFID system: passive and active, just like distance ranging systems. Passive RFID tags contain an integrated circuit that has a basic radio

transceiver and a small amount of nonvolatile memory. They are powered by the current that the reader's signal induces in their antennas. The received energy is just enough to power the tag to transmit its data once, and the signal is relatively weak. Most passive readers can only read tags a few inches to a few feet away.

In an active RFID system, the tag has its own power supply and radio transceiver, and transmits a signal in response to a received message from a reader. Active systems can transmit for a much longer range than passive systems, and are less error-prone. They are also much more expensive. If you're a regular automobile commuter and you have to pass through a toll gate in your commute, you're probably an active RFID user. Systems like E-ZPass use active RFID tags so that the reader can be placed several meters away from the tag.

You might think that because RFID is radio-based, you could use it to do radio distance ranging as well, but that's not the case. Neither passive nor active RFID systems are typically designed to report the signal strength received from the tag. Without this information, it's impossible to use RFID systems to determine the actual location of a tag. All the reader can tell you is that the tag is within reading range. Although some high-end systems can report the tag signal strength, the vast majority of readers are not made for location as well as identification.

RFID systems vary widely in cost. Active systems can cost tens of thousands of dollars to purchase and install. Commercial passive systems can also be expensive. A typical passive reader that can read a tag a meter away from the antenna typically costs a few thousand dollars. At the low end, short-range passive readers can come as cheap as $30 or less. As of this writing, $100 gets you a reader that can read no more than a few inches. Anything that can read a longer distance will be more expensive.

There are many different RFID protocols, just as with bar codes. Short-range passive readers come in at least three common frequencies: two low-frequency bands at 125 and 134.2Khz, and high-frequency readers at 13.56MHz. The higher-frequency readers allow for faster read rates and longer-range reading distances. In addition to different frequencies, there are also different protocols. For example, in the 13.56 band alone, there are the ISO 15693 and ISO 14443 and 14443-A standards; within the ISO 15693 standard, there are different implementations by different manufacturers: Philips' I-Code, Texas Instruments' Tag-IT HF, Picotag, and implementations by Infineon, STMicroelectronics, and others. Within the ISO 14443 standard, there's Philips' Mifare, Mifare UL, ST's SR176, and others. So you can't count on one reader to read every tag. You can't even count on one reader to read all the tags in a given frequency range. You have to match the tag to the reader.

There are a number of inexpensive and easy-to-use readers on the market now, covering the range of passive RFID frequencies and protocols. Parallax (www.parallax.com) sells a 125KHz reader that can read EM Microelectronic tags such as EM4001 tags. It has a built-in antenna, and the whole module is about 2.5" x 3.5", on a flat circuit board. ID Innovations makes a number of small low-frequency readers less than 1.5 inches on a side, capable of reading the EM4001 protocol tags. SparkFun (www.sparkfun.com) and CoreRFID (www.rfidshop.com) both sell the ID Innovations readers and matching tags. The ID Innovations readers and the Parallax readers can read the same tags. Trossen Robotics (www.trossenrobotics.com) sells a range of readers, the least expensive of which is the APSX RW-210, a 13.56MHz module that can read and write to tags using the ISO 15693 protocol. Trossen's also got a wide range of tags for everyone's readers, including the EM tags that match the Parallax and ID Innovations readers. SkyeTek (www.skyetek.com) makes a number of small readers like the M1 and the M1-mini that operate in the 13.56MHz range as well. Though their readers are moderately priced, SkyeTek generally doesn't sell them until you've bought their development kit, which is priced considerably higher. Texas Instruments (www.ti.com/rfid/shtml/rfid.shtml) makes a 134.2KHz reader, the RI-STU-MRD1. The Texas Instruments reader and the SkyeTek readers are the only ones mentioned here that don't come with a built-in antenna. You can make your own, however, and TI helpfully provides advice on how to do it in the reader's data sheet.

RFID tags come in a number of different forms, as shown in Figure 9-4: sticker tags, coin discs, key fobs, credit cards, playing cards, even capsules designed for injection under the skin. The last are used for pet tracking and are not designed for human use, though there are some adventurous hackers who have had these tags inserted under their own skin. Like any radio signal, RFID can be read through a number of materials, but is blocked by any kind of RF shielding, like wire mesh, conductive fabric lamé, metal foil, or adamantium skeletons. This feature means that you can embed it in all kinds of projects, as long as your reader has the signal strength to penetrate.

Figure 9-4
RFID tags in all shapes and sizes.

Before picking a reader, think about the environment in which you plan to deploy it, and how that affects both the tags and the reading. Will the environment have a lot of RF noise? In what range? Consider a reader outside that range. Will you need a relatively long-range read? If so, look at the high-frequency readers, if possible. If you're planning to read existing tags rather than tags you purchase yourself, research carefully in advance, because not all readers will read all tags. Pet tags can be some of the trickiest, as many of them operate in the 134.2KHz range, in which there are fewer readers to choose from.

In picking a reader, you also have to consider how it behaves when tags are in range. For example, even though the Parallax reader and the ID Innovations readers can read the same tags, they behave very differently when a tag is in range. The ID Innovations reader reports the tag ID only once. The Parallax reader reports it continually until the tag is out of range. The behavior of the reader can affect your project design, as you'll see later on.

All of the readers mentioned here have TTL serial interfaces, so they can be connected to a microcontroller or a USB-to-serial module very easily. Sketches in Processing for the APSX reader, the Parallax reader, and the ID Innovations reader follow. All of these readers have a similar operating scheme. The APSX is the only one you need to send a serial command to; all of the others simply transmit a tag ID whenever a tag is in range.

Most RFID capsules are not sanitized for internal use in animals (humans included), and they're definitely not designed to be inserted without qualified medical supervision. Besides, insertion hurts. Don't RFID-enable yourself or your friends. Don't even do it to your pets—let your vet do it for you. If you're really gung-ho to be RFID-tagged, make yourself a nice set of RFID tag earrings.

X

Project 23

Reading RFID Tags in Processing

In this project, you'll read some RFID tags and get a sense of how the readers behave. You'll see how far away from your reader a tag can be read. This is a handy test program for use any time you're adding RFID to a project.

MATERIALS

- **RFID reader** Either the RFID Reader Module from Parallax, part number 28140; the ID Innovations ID-12 from CoreRFID or from SparkFun as part number SEN-08419; or the APSX RW-210 from Trossen Robotics.
- **Two 2mm female header rows** If you're using the ID12 reader, you'll need Samtec (www.samtec.com) part number MMS-110-01-L-SV. Samtec, like many part makers, supplies free samples of this part in small quantities. SparkFun sells these as part number PRT-08272. They also sell a breakout board for the module, part number SEN-08423.
- **RFID tags** Get the tags that match your reader. All three of the retailers listed earlier in this list sell tags that match their readers in a variety of physical packages, so choose the ones you like the best.
- **1 USB-to-TTL serial adaptor** SparkFun's BOB-00718 from Chapter 2 can do the job. If you use a USB-to-RS232 adaptor such as a Keyspan or logear dongle, refer to Chapter 2 for the schematics to convert RS-232-to-5V TTL serial.

The Circuits

The circuits for all three readers are fairly similar. Connect the module to 5V and ground, and connect the reader's serial transmit line to the serial adaptor's serial receive line, and vice versa. For the Parallax reader, you'll also need to attach the enable pin to ground. For the ID12, connect the reset pin to 5V, and connect an LED from the Card Present pin to ground as well. Figures 9-6, 9-7, and 9-8 show the circuits for the Parallax, ID Innovations, and ASPX readers, respectively.

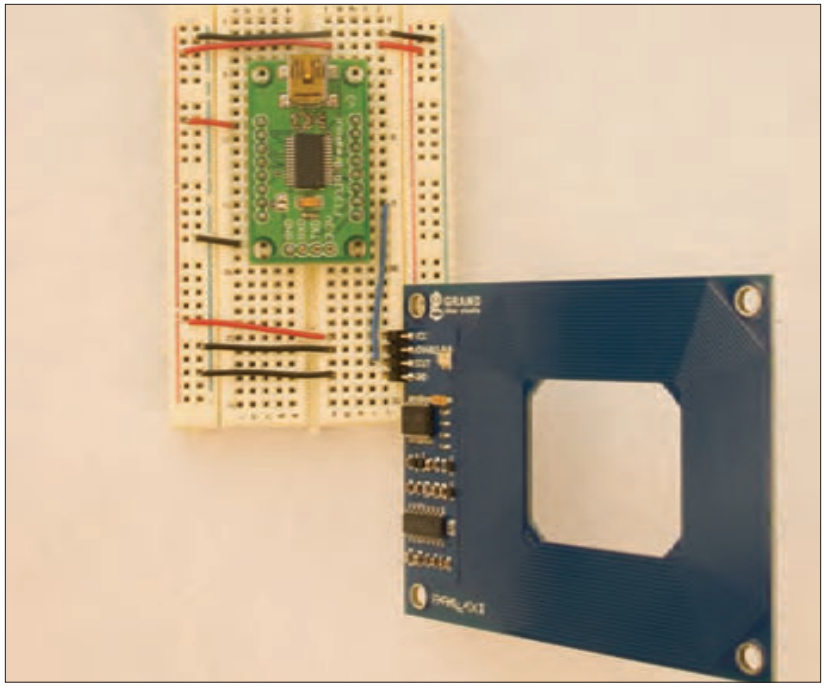
Parallax RFID Reader

The Parallax reader is the simplest of the three readers to read. It communicates serially at 2400 bps. When the Enable pin is held low, it sends a reading whenever a tag is in range. The tag ID is a 12-byte string starting with a carriage return (ASCII 13) and finishing with a newline (ASCII 10). The ten digits in the middle are the unique tag ID. The EM4001 tags format their tag IDs as ASCII-encoded hexadecimal values, so the string will never contain anything but the ASCII digits 0 through 9 and the letters A through F.

Try It The following sketch waits for twelve serial bytes, strips out the carriage return and the newline, and prints the rest to the screen:

```
/*
 Parallax RFID Reader
 language: Processing

*/

// import the serial library:
import processing.serial.*;

Serial myPort;      // the serial port you're using
String tagID = "";  // the string for the tag ID

void setup() {
  size(600,200);
```

»



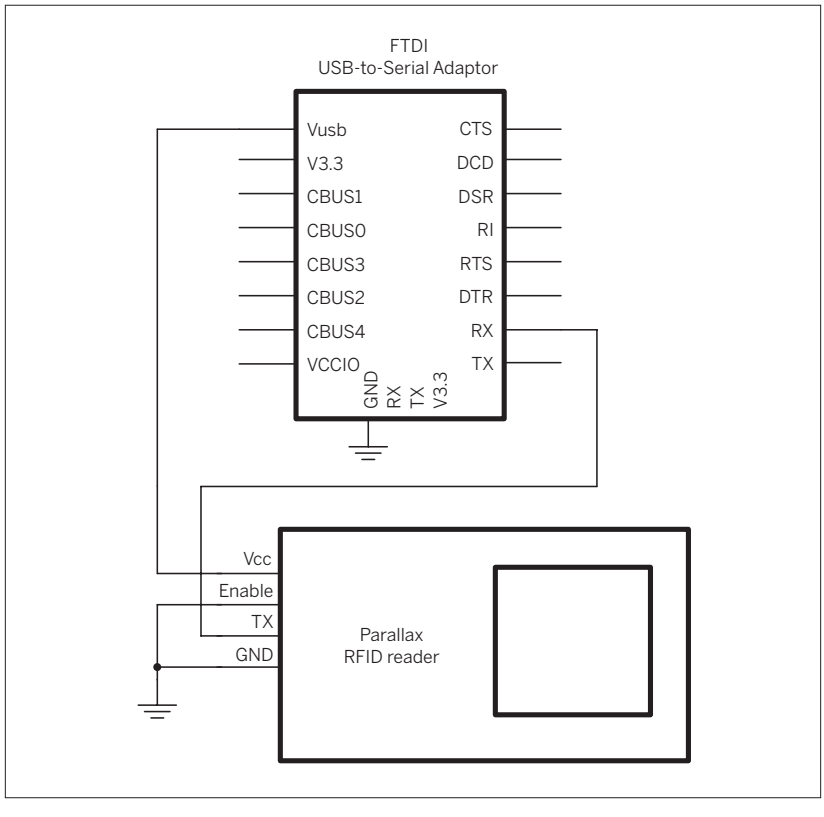


Figure 9-6
The Parallax RFID reader connected to an FTDI USB-to-serial adaptor.

Continued from previous page.

```
  // list all the serial ports:
  println(Serial.list());

  // based on the list of serial ports printed from the
  // previous command, change the 0 to your port's number:
  String portnum = Serial.list()[0];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 2400);
  // incoming string from reader will have 12 bytes:
  myPort.buffer(12);

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 24);
  textFont(myFont);
}

void draw() {
  // clear the screen:
  background(0);
  // print the string to the screen:
  text(tagID, width/4, height/2 - 24);
}

/*
 this method reads bytes from the serial port
 and puts them into the tag string.
 It trims off the \r and \n
 */
void serialEvent(Serial myPort) {
  tagID = trim(myPort.readString());
}
```

ID Innovations ID12 Reader

The ID Innovations reader (see Figure 9-7) is slightly more complex than the Parallax reader. It operates at 9600 bps. It has an output pin that goes high when a tag is present, which is a handy way to know if it's reading your tag, even if you haven't got it connected to anything. It reads the same tags as the Parallax reader, but doesn't format the data the same way. All the ID Innovations readers use the same protocol. It starts with a start-of-transmission (STX) byte (ASCII 02) and ends with an end-of-transmission (ETX) byte (ASCII 03). The STX is followed by the ten-byte tag ID. A checksum follows that, then a carriage return (ASCII 13) and linefeed (ASCII 10), then the ETX. The EM4001 tags format their tag IDs as ASCII-encoded hexadecimal values, so the string will never contain anything but the ASCII digits 0 through 9 and the letters A through F.

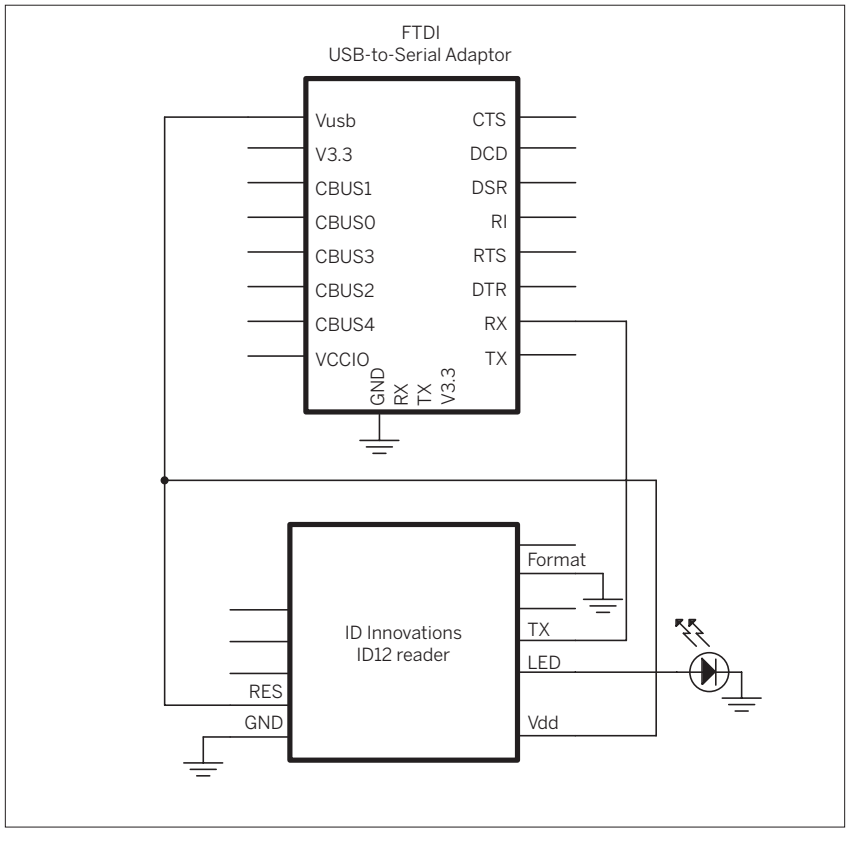
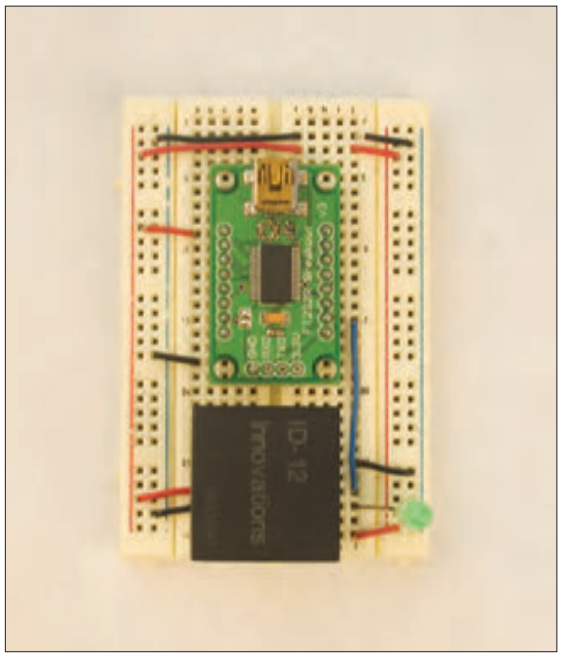
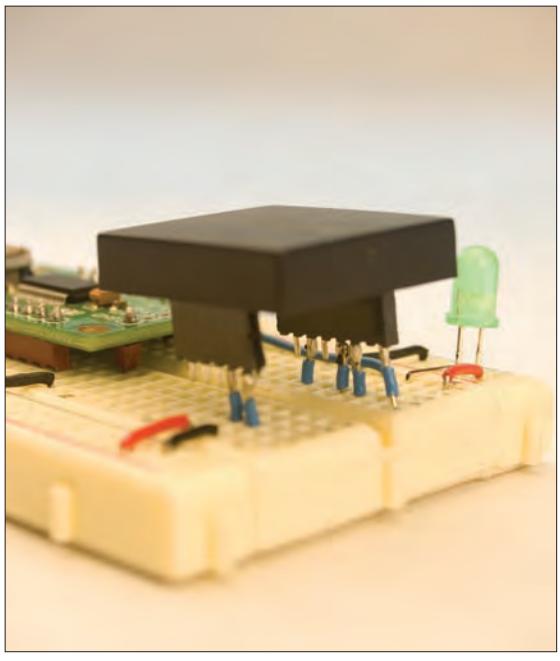


Figure 9-7
The ID Innovations ID12 RFID reader attached to an FTDI USB-to-serial adaptor. The ID12 has pins spaced 2mm apart, so you'll need to solder wires onto them to fit them on a breadboard. You can also use the 2mm female sockets used with the XBee modules, or you can use SparkFun's breakout board.





Try It The sketch shown here is a modification of the Parallax sketch, with a new method, parseString(). It reads the entire string, confirms that the start and end bytes are there, and strips out all but the ten-byte tag ID. The changes to the previous sketch are shown in blue:

```
/*
 ID Innovations RFID Reader
 language: Processing

*/

// import the serial library:
import processing.serial.*;

Serial myPort;      // the serial port you're using
String tagID = "";  // the string for the tag ID

void setup() {
  size(600,200);
  // list all the serial ports:
  println(Serial.list());

  // based on the list of serial ports printed from the
  // previous command, change the 0 to your port's number:
  String portnum = Serial.list()[0];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 9600);
  // incoming string from reader will have 16 bytes:
  myPort.buffer(16);

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 24);
  textFont(myFont);
}

void draw() {
  // clear the screen:
  background(0);
  // print the string to the screen:
  text(tagID, width/4, height/2 - 24);
}

/*
 this method reads bytes from the serial port
 and puts them into the tag string
 */

void serialEvent(Serial myPort) {
  // get the serial input buffer in a string:
  String inputString = myPort.readString();
  // filter out the tag ID from the string:
  tagID = parseString(inputString);
}
```

Continued from opposite page.

```
/*
  This method reads a string and looks for the 10-byte
  tag ID. It assumes that it gets an STX byte (0x02)
  at the beginning and an ETX byte (0x03) at the end.
 */
String parseString(String thisString) {
  String tagString = "";    // string to put the tag ID into

  // first character of the input:
  char firstChar = thisString.charAt(0);
  // last character of the input:
  char lastChar = thisString.charAt(thisString.length() -1);

  // if the first char is STX (0x02) and the last char is ETX (0x03),
  // then put the next ten bytes into the tag string:
  if ((firstChar == 0x02) && (lastChar == 0x03)) {
    tagString = thisString.substring(1, 11);
  }
  return tagString;
}
```

APSX RW-210 Reader

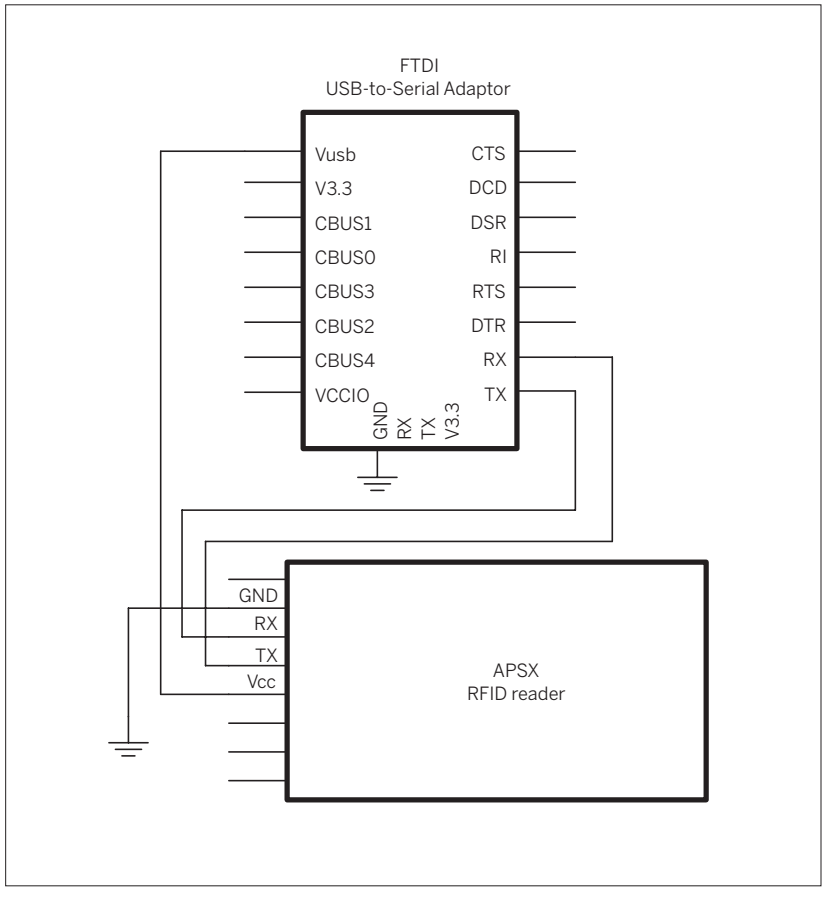
The APSX reader (Figure 9-8) has a totally different format than the previous two. First, you have to send it a command byte to start it reading. You can send it either a byte of value 250 (0xFA), which causes it to read once, or a byte of value 251 (0xFB), which causes it to read continually. The read-once option saves power, as it powers the reader down once a tag is read. The tag ID returned is 12 bytes long. The byte values are not limited to alphanumeric values, so the following sketch converts them to ASCII-encoded hexadecimal values and separates them by spaces for easy reading. Despite the different format, the code is very similar to the Parallax reader sketch.

Try It Changes from the Parallax reader sketch are shown in blue:

```
/*
 APSX RFID Reader
 language: Processing

*/

// import the serial library:
import processing.serial.*;

Serial myPort;      // the serial port you're using
String tagID = "";  // the string for the tag ID

void setup() {
  size(600,200);
  // list all the serial ports:
  println(Serial.list());
```

Continued from previous page.

```
  // based on the list of serial ports printed from the
  // previous command, change the 0 to your port's number:
  String portnum = Serial.list()[0];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 19200);
  // incoming string from reader will have 12 bytes:
  myPort.buffer(12);

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 24);
  textFont(myFont);

  // send the continual read command:
  myPort.write(0xFB);
}

void draw() {
  // clear the screen:
  background(0);
  // print the string to the screen:
  text(tagID, width/8, height/2 - 24);
}

/*
  this method reads bytes from the serial port
  and puts them into the tag string
 */
 void serialEvent(Serial myPort) {
  int thisByte = 0;
  tagID = "";
  while(myPort.available() > 0) {
    int newByte = myPort.read();
    tagID += hex(newByte, 2);
    tagID += " ";
  }
}
```



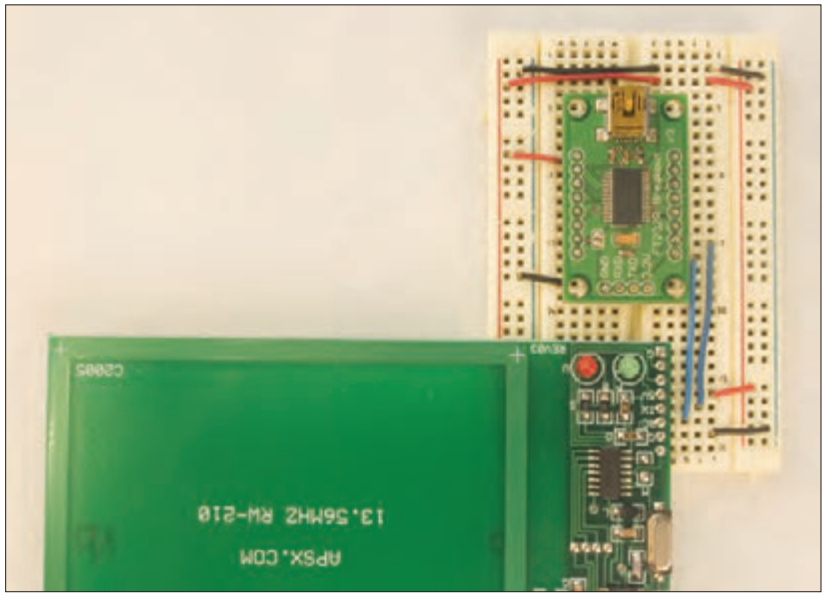


Figure 9-8
The APSX RW-210 RFID reader attached to an FTDI USB-to-serial adaptor.

Project 24

RFID Meets Home Automation

Between my officemate and me, we have dozens of devices drawing power in our office: two laptops, two monitors, four or five lamps, a few hard drives, a soldering iron, Ethernet hubs, speakers, and so forth. Even when we're not here, the room is drawing a lot of power. What devices are turned on at any given time depends largely on which of us is here, and what we're doing. This project is a system to reduce our power consumption, particularly when we're not there.

When either of us comes into the room, all we have to do is throw our keys on a side table by the door, and the room turns on or off what we normally use. Each of us has a key ring with an RFID-tag key fob. The key table has an RFID reader in it, and reads the presence or absence of the tags.

The reader is connected to a microcontroller module that communicates over the AC power lines using the X10 protocol. Each of the various power strips is plugged into an X10 appliance module. Depending on which tag is read, the microcontroller knows which modules to turn on or off. Figure 9-9 shows the system.

The Circuit

The RFID module is connected to the microcontroller as you might expect: the module's transmit pin connects to the microcontroller's receive. It's basically the same circuit as shown in the previous project with the X10 module added. The X10 interface module connects to the microcontroller via the phone cable. Clip one end of the cable and solder headers onto the four wires. Then connect them to the microcontroller as shown in Figure 9-10. The schematic shows the phone jack (an RJ-11 jack) on the interface module as you're looking at it from the bottom. Make sure the wires at the header ends correspond with the pins on the jack from right to left.

MATERIALS

- **1 solderless breadboard** such as Digi-Key (www.digikey.com) part number 438-1045-ND, or Jameco (www.jameco.com) part number 20601. For the photos in this example, I used an Arduino-compatible Protoshield module from SparkFun, part number DEV-07914.
- **1 Arduino module** or other microcontroller
- **RFID reader** Either the RFID Reader Module from Parallax, part number 28140; the ID Innovations ID-12 from CoreRFID or SparkFun part number SEN-08419; or the APSX RW-210 from Trossen Robotics.
- **Two 2mm female header rows** if you're using the ID12 reader, Samtec part number MMS-110-01-L-SV. Samtec, like many part makers, supplies free samples of this part in small quantities. SparkFun sells these as part number PRT-08272. They also sell a breakout board for the module, SparkFun part number SEN-08423.
- **RFID tags** Get the tags that match your reader. All three of the retailers listed sell tags that match their readers in a variety of physical packages, so choose the ones you like the best.
- **Interface module: X10 One-Way Interface Module** from Smarthome (www.smarthome.com), part number 1134B.
- **2 X10 modules** *Either:* 2 Appliance Modules from Smarthome, part number 2002, or 2 Powerhouse X10 Lamp Modules from Smarthome, part number 2000. You'll need two modules total. Choose one of each, or two of one as you see fit. If you're going to control only incandescent lamps, get lamp modules. For anything else, get appliance modules.
- **4-wire phone cable with RJ-11 connector** You can take this from any discarded phone, or get one at your local electronics shop.

I used an extra few inches of the phone cable to make an extension cable for the RFID reader. You may or may not need to, depending on how you plan to enclose the electronics. Separating the two makes it easier to hide the microcontroller and reader in two separate places, so the antenna can get closer to what you need to read.

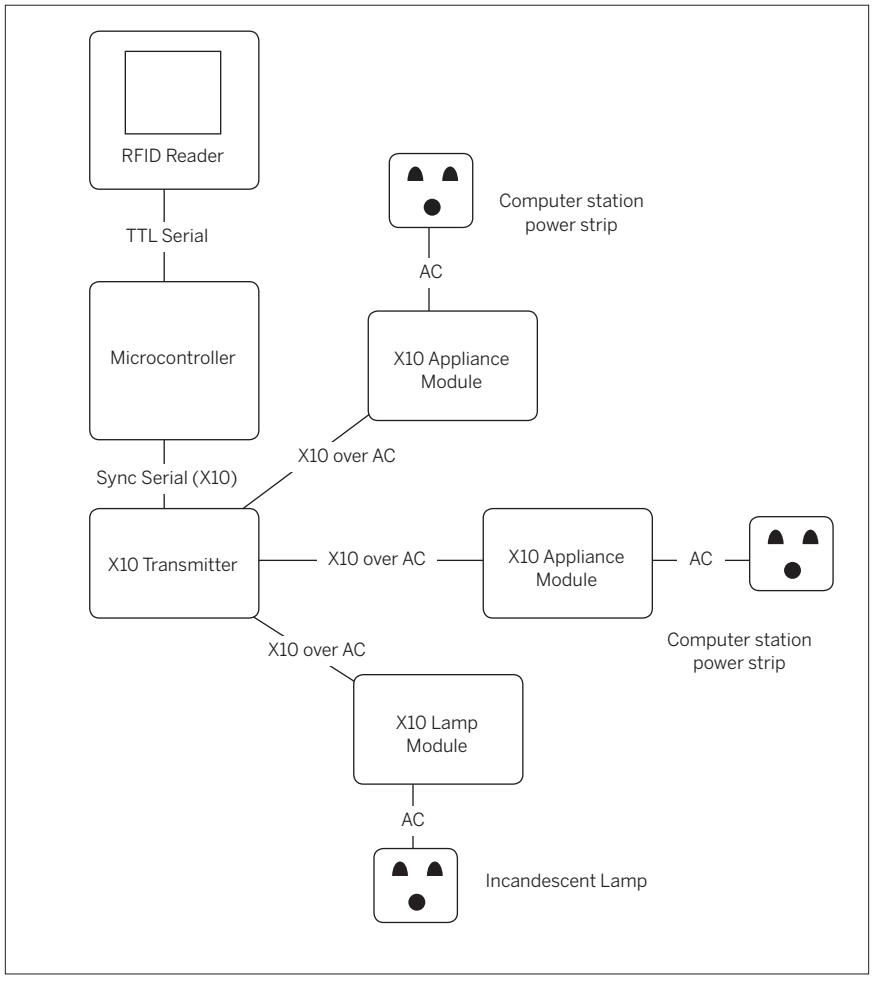


Figure 9-9
An RFID-controlled home (or office) automation system using X10.

To send X10 commands, use the X10 library for Arduino. You can download it from www.arduino.cc/en/Tutorial/X10. Unzip it and place the resulting directory in the **lib/targets/libraries/** subdirectory of your Arduino application directory. Then restart the Arduino program.

X10 device addresses have a two-tier structure. There are 16 house codes, labeled A through P, and within each house code, you can have 16 individual units. These are assigned unit codes. Each X10 module has two click-wheels to set the house code and the unit code. For this project, get at least two appliance or lamp modules. Set the first module to house code A, unit 1, and the second code A, unit 2.

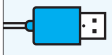
NOTE: The X10 library may be already included with later versions of Arduino, after version 0009. Check the Arduino website at www.arduino.cc to be sure.

The Code

Once you've got the circuit connected, program the microcontroller to read the RFID tags, just to make sure that works. A test sketch is shown next.

NOTE: You'll probably need to disconnect the serial line between the RFID reader and the Arduino in order to program the module, as you did with the XPort and XBee modules in earlier chapters.

NOTE: You also need to set the serial monitor speed to 2400 bps to match the speed this program uses. The default for the Arduino serial monitor is 9600, and you can change it from a pop-up menu in the serial monitor.



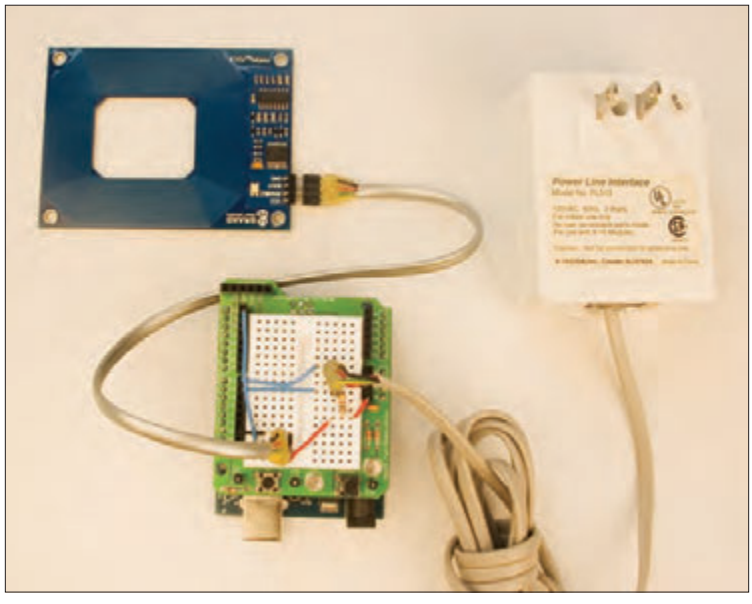
What is X10?

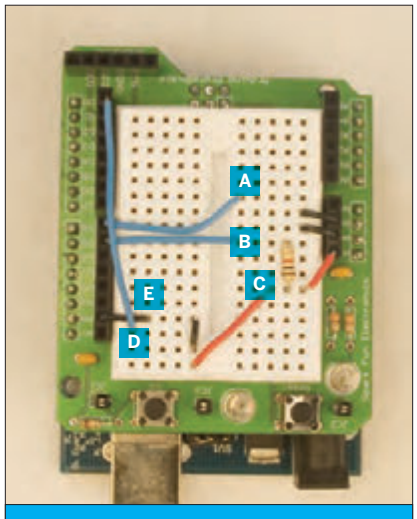
X10 is a communications protocol that works over AC power lines. It's designed for use in home automation. Companies such as Smarthome (www.smarthome.com) and X10.com (www.x10.com) sell various devices that communicate over power lines using X10: cameras, motion sensors, switch control panels, and more. It's a slow and limited protocol, but has been popular with home automation enthusiasts for years, because the equipment is relatively inexpensive and easy to obtain.

X10 is basically a synchronous serial protocol, like I2C and SPI. Instead of sending bits every time a master clock signal changes, X10 devices send a bit every time the AC power line crosses zero volts. This means that X10's maximum data rate is 120 bits per second in the U.S., as the AC signal crosses the zero point twice per cycle, and AC signals are 60Hz in the U.S. The protocol is tricky to program if you have to do it yourself, but many microcontroller development systems include libraries to send X10 signals.

There are four devices that come in handy for developing X10 projects: an interface module, an appliance control module, a lamp control module, and a control panel module. You'll be building your own controllers, but the control panel module is useful as a diagnostic tool, because it already works. When you can't get the appliance or lamp modules to respond to your own projects, you can at least get them to respond to the control panel module — that way, you know whether the bits are passing over the power lines. Smarthome sells versions of all four of these:

- Interface module: X10 One-Way Interface Module, part number 1134B. You'll see two common versions of this, the PL513 and the TW523. They both work essentially the same way. The TW523 is a two-way module, and can send and receive X10 signals, while the PL513 can only send.
- Appliance control module: X10 Appliance Module 3-Pin, part number 2002. These can control anything you can plug into an AC socket, up to 15 Amps.
- Lamp control module: Powerhouse X10 Lamp Module, part number 2000. These can control only incandescent (not fluorescent or neon) lamps.
- Control panel module: X10 Mini Controller, part number 4030





A. Data B. Zero crossing C. +5V
D. RFID TX E. RFID enable

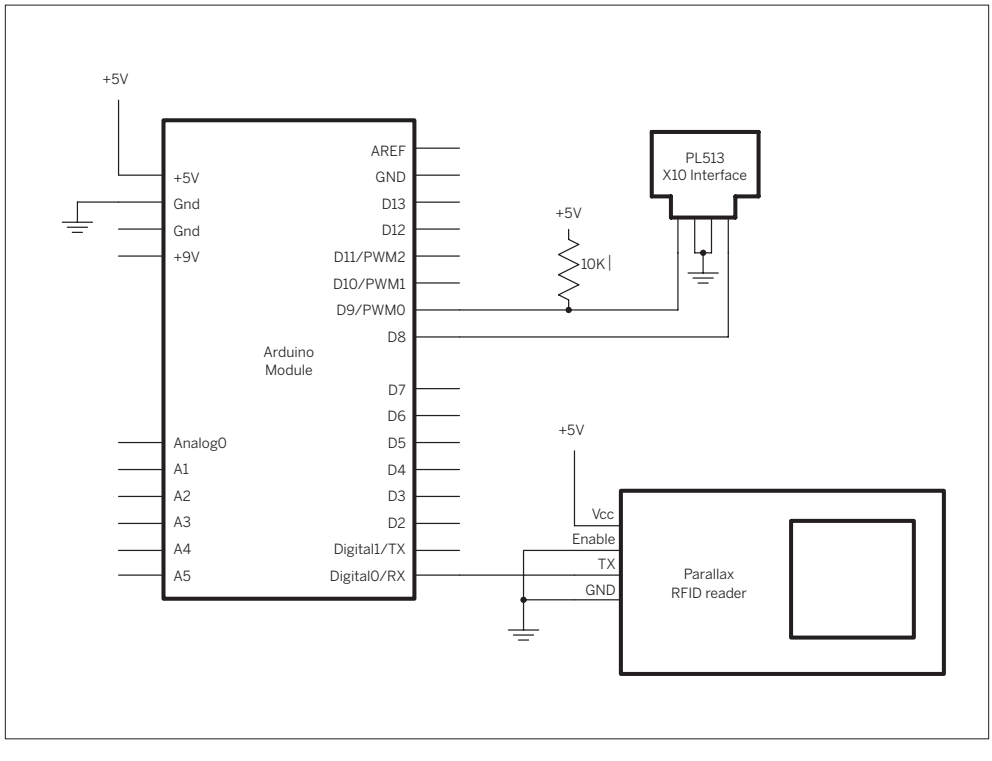


Figure 9-10
The circuit for the RFID-to-X10 project. The detail shows the connections on the breadboard.

Test It This sketch reads in bytes similar to the Processing sketches earlier. The readByte() method does all the work with the serial data. It reads the first byte, and if it's the correct value, it resets an array called tagID. It saves the next ten bytes into the array. When it gets the final byte, it changes a variable called tagComplete so that the rest of the program knows it can use the tag array. You can use the same code for the Parallax reader or the ID Innovations reader; all you have to do is change the startByte and endByte values and the data rate.

```
/*
 RFID Reader
 language: Wiring/Arduino

*/

#define tagLength 10    // each tag ID contains 10 bytes
#define startByte 0x0A  // for the ID Innovations reader, use 0x02
#define endByte 0x0D    // for the ID Innovations reader, use 0x03
#define dataRate 2400   // for the ID Innovations reader, use 9600


char tagID[tagLength];            // array to hold the tag you read
int tagIndex = 0;                 // counter for number of bytes read
int tagComplete = false;          // whether the whole tag's been read

void setup() {
  // begin serial:
  Serial.begin(dataRate);
}

void loop() {
  // read in and parse serial data:
  if (Serial.available() > 0) {
    readByte();
  }
  if(tagComplete == true) {

    Serial.println(tagID);
  }
}

/*
  This method reads the bytes, and puts the
  appropriate ones in the tagID

 */
void readByte() {
  char thisChar = Serial.read();
Serial.print(thisChar, HEX);
  switch (thisChar) {
  case startByte:     // start character
    // reset the tag index counter
    tagIndex = 0;
    break;
  case endByte:          // end character
    tagComplete = true;  // you have the whole tag
    break;
  default:               // any other character
    tagComplete = false; // there are still more bytes to read
    // add the byte to the tagID
```

Continued from opposite page.

```
    if (tagIndex < tagLength) {
      tagID[tagIndex] = thisChar;
      // increment the tag byte counter
      tagIndex++;
    }
    break;
  }
}
```

When you know that works, run this sketch to test the X10:

```
/*
 X10 test
 language: Wiring/Arduino

*/
// include the X10 library files:
#include <x10.h>
#include <x10constants.h>

#define zcPin 9         // the zero crossing detect pin
#define dataPin 8       // the X10 data out pin
#define repeatTimes 1   // how many times to repeat each X10 message
                        // in an electrically noisy environment, you
                        // can set this higher.

// set up a new x10 library instance:
x10 myHouse =  x10(zcPin, dataPin);

void setup() {
  // turn off all lights:
  myHouse.write(A, ALL_UNITS_OFF,repeatTimes);
}

void loop() {
        // turn on first module:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
     delay(500);
        // turn on second module:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        delay(500);

}
```

It's unlikely that this will work the first time. X10 is notorious for having synchronization problems, and it doesn't work when the transmitter and receiver are on different circuits. Some of the more expensive surge protectors might filter out X10 as well. If your lights don't turn on correctly, start by unplugging everything, then set the addresses, then plug everything in, then reset the Arduino. If that fails, make sure that your units are on the same circuit, and eliminate surge protectors, if you're using them. Try to turn the modules using a control panel module. Once you've got control over your modules, you can combine the RFID and the X10 programs.

Refine It First, combine the initialization and setup routines like so (be sure to set tagOne and tagTwo to the values of your tags):

```
/*
 RFID –to-X10 translator
 language: Wiring/Arduino

*/

// include the X10 library files:
#include <x10.h>
#include <x10constants.h>

#define zcPin 9         // the zero crossing detect pin
#define dataPin 8       // the X10 data out pin
#define repeatTimes 1   // how many times to repeat each X10 message
                        // in an electrically noisy environment, you
                        // can set this higher.

#define tagLength 10    // each tag ID contains 10 bytes
#define startByte 0x0A  // for the ID Innovations reader, use 0x02
#define endByte 0x0D    // for the ID Innovations reader, use 0x03
#define dataRate 2400   // for the ID Innovations reader, use 9600

// set up a new x10  library instance:
x10 myHouse =  x10(zcPin, dataPin);

char tagID[tagLength];          // array to hold the tag you read
int tagIndex = 0;               // counter for number of bytes read
int tagComplete = false;        // whether the whole tag's been read
char tagOne[] = "0415AB6FB7";  // put the values for your tags here
char tagTwo[] = "0415AB5DAF";
char lastTag = 0;               // value of the last tag read

void setup() {
  // begin serial:
  Serial.begin(dataRate);
  // turn off all lights:
  myHouse.write(A, ALL_LIGHTS_OFF,repeatTimes);
}
```

The loop() method is a bit more complex now. You need to add a block to compare the RFID tag to the two existing tag numbers. You'll call a method called compareTags() to do this. This method just iterates over the arrays and compares them byte by byte. Once you've got a match, you send the appropriate X10 commands. Here's the rest of the sketch:

```
void loop() {
  // read in and parse serial data:
  if (Serial.available() > 0) {
    readByte();
  }

  // if you've got a complete tag, compare your tag
  // to the existing values:

  if (tagComplete == true) {
    if (compareTags(tagID, tagOne) == true) {

      if (lastTag != 1) {
        // if the last tag wasn't this one,
        // send commands:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
        // note that this was the last tag read:
        lastTag = 1;
      }
    }
    if (compareTags(tagID, tagTwo) == true) {
      if (lastTag != 2) {
        // if the last tag wasn't this one,
        // send commands:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        // note that this was the last tag read:
        lastTag = 2;
      }
    }
  }
}

/*
  this method compares two char arrays, byte by byte:
 */
char compareTags(char* thisTag, char* thatTag) {
  char match = true;  // whether they're the same

    for (int i = 0; i < tagLength; i++) {
    // if any two bytes don't match, the whole thing fails:
    if (thisTag[i] != thatTag[i]) {
      match = false;
    }
  }
  return match;
}
```

Continued from previous page.

```
/*
  This method reads the bytes, and puts the
  appropriate ones in the tagID

 */
void readByte() {
  char thisChar = Serial.read();

  switch (thisChar) {
  case startByte:         // start character
    // reset the tag index counter
    tagIndex = 0;
    break;
  case endByte:           // end character
    tagComplete = true;   // you have the whole tag
    break;
  default:                // any other character
    tagComplete = false;  // there are still more bytes to read
    // add the byte to the tagID
    if (tagIndex < tagLength) {
      tagID[tagIndex] = thisChar;
      // increment the tag byte counter
      tagIndex++;
    }
    break;
  }
}
```

When you run this code, you'll see that the RFID signals are slow enough that you can actually see one complete before the other begins. It's not a good protocol for real-time interaction. The assumption in this application is that the RFID tag is going to remain in place for a long time, so a few seconds' delay is not a problem. However, this is where the reader's behavior in the presence of a tag makes a difference. The ID Innovations reader doesn't continue reporting tags if they remain in the field, while the Parallax one does. With the Parallax reader, you can detect not only the presence of a tag, but also the absence. In this application, it means that you can leave your key tag in the bowl as long as you want the lights on, then remove it when you want them off.

None of the readers shown here features the ability to read multiple tags if more than one tag is in the field. That's an important limitation. It means that you have to design the interaction so that the person using the system places only one tag at a time, then removes it before the second one is placed. In effect, it means that two people can't place their key tags in the bowl at the same time. In other words, users of the system need to take explicit action to make something happen. Presence isn't enough.

X

Figure 9-11
The finished RFID reader bowl. A bamboo box and dessert plate from a nearby gift shop made a nice housing. Double-stick tape holds the reader to the top of the box. A hole drilled in the back of the box provides access for the X10 and power cables.

Network Identification

So far, you've identified network devices computationally by their address. For devices on the Internet, you've seen both IP addresses and MAC addresses. Bluetooth and 802.15.4 devices have standard addresses as well. The address of a device doesn't tell you anything about what the device is or what it does.

Recall the networked air quality project in Chapter 4. The microcontroller made a request via HTTP and the PHP script sent back a response. Because you already knew the microcontroller's capabilities, you could send a response that was short enough for it to process efficiently, and format it in a way that made it easy to read. But what if that same PHP script had to respond to HTTP requests from an XPort, a desktop browser like Safari or Internet Explorer, and a mobile phone browser? How would it know how to format the information?

Most net communications protocols include a basic exchange of information about the sender's and receiver's identity and capabilities as part of the initial header messages. You can use these to your advantage when designing network systems like the ones you've seen here. There's not room here to discuss this concept comprehensively, but following are two examples that use HTTP and mail.

HTTP Environment Variables

When a server-side program, such as a PHP script, receives an HTTP request, it has access to a lot more information than you've seen thus far about the server, the client, and more.

To see some of it, save the following PHP script to your web server, then open it in a browser. Call it **env.php**:

```
<?php
/*
    Environment Variable Printer
    Language: PHP

    Prints out the environment variables
*/
   foreach ($_REQUEST as $key => $value)
   {
       echo "$key: $value<br>\n";
   }
   foreach ($_SERVER as $key => $value)
   {
       echo "$key: $value<br>\n";
   }
?>
```

You should get something like this in your browser.

```
DBENTRY: /home/youraccountname/:d0000#CPU 6 #MEM 10240 #CGI
16734 #NPROC 12 #TAID 36811298 #WERB 0 #LANG 3 #PARKING 1
#STAT 1
DOCUMENT_ROOT: /home/youraccountname/
HTTP_ACCEPT: text/xml,application/xml,application/
xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/
*;q=0.5
HTTP_ACCEPT_CHARSET: ISO-8859-1,utf-8;q=0.7,*;q=0.7
HTTP_ACCEPT_ENCODING: gzip,deflate
HTTP_ACCEPT_LANGUAGE: en-us,en;q=0.5
HTTP_CONNECTION: keep-alive
HTTP_COOKIE: __utmz=152494652.1182194862.12.12.utmccn=(r
eferral)|utmcsr=www.someserver.com|utmcct=/~youraccount/
|utmcmd=referral; __utma=152494652.116689300.1180965223.1181
918391.1182194862.12; __utmc=152494652
HTTP_HOST: www.example.com
HTTP_KEEP_ALIVE: 300
HTTP_USER_AGENT: Mozilla/5.0 (Macintosh; U; Intel Mac OS X;
en-US; rv:1.8.1.4) Gecko/20070515 Firefox/2.0.0.4
PATH: /bin:/usr/bin
REDIRECT_DBENTRY: /home/youraccountname/:d0000#CPU 6 #MEM
10240 #CGI 16734 #NPROC 12 #TAID 36811298 #WERB 0 #LANG 3
#PARKING 1 #STAT 1
REDIRECT_SCRIPT_URI: http://www.example.com/php/09_env.php
REDIRECT_SCRIPT_URL: /php/09_env.php
REDIRECT_STATUS: 200
REDIRECT_UNIQUE_ID: RnfW1UrQECcAAFcpZ2w
REDIRECT_URL: /php/09_env.php
REMOTE_ADDR: 66.168.47.40
REMOTE_PORT: 39438
SCRIPT_FILENAME: /home/youraccountname/php/09_env.php
SCRIPT_URI: http://www.example.com/php/09_env.php
SCRIPT_URL: /php/09_env.php
SERVER_ADDR: 77.248.128.3
SERVER_ADMIN: webmaster@example.com
SERVER_NAME: example.com
SERVER_PORT: 80
SERVER_SIGNATURE:
SERVER_SOFTWARE: Apache/1.3.33 (Unix)
UNIQUE_ID: RnfW1UrQECcAAFcpZ2w
GATEWAY_INTERFACE: CGI/1.1
SERVER_PROTOCOL: HTTP/1.1
REQUEST_METHOD: GET
QUERY_STRING:
REQUEST_URI: /php/09_env.php
SCRIPT_NAME: /php/09_env.php
PATH_INFO: /php/09_env.php
PATH_TRANSLATED: /home/youraccountname/php/09_env.php
STATUS: 200
```

As you can see, there's a lot of information there: the web server's IP address, the client's IP address, the browser type, the directory path to the script, and more. You probably never knew you were giving up so much information when you make a simple HTTP request, and this is only a small part of it! This is very useful when you want to write CGI scripts that can respond to different clients in different ways.

Project 25

IP Geocoding

The next example uses the client's IP address to get its latitude and longitude. It gets this information from www.hostip.info, a community-based IP geocoding project. The data there is not always the most accurate, but it is free. This script also uses the HTTP user agent to determine whether the client is a desktop browser or a Lantronix device. It then formats its response appropriately for each device.

Locate It Save this to your server as **ip_geocoder.php**:

```
<?php
/*  IP geocoder
    language: PHP

    Uses a client's IP address to get a latitude and longitude.
    Uses the client's user agent to format the response.
*/
    // initialize variables:
    $lat = 0;
    $long = 0;
    $ipAddress = "0.0.0.0";
    $country = "unknown";

    // check to see what type of client this is:
    $userAgent  = getenv('HTTP_USER_AGENT');
    // get the client's IP address:
    $ipAddress = getenv('REMOTE_ADDR');
```

» The site www.hostip.info will return the latitude and longitude from the IP address. First, format the HTTP request string and make the request. Then wait for the results in a while loop, and separate the results into the constituent parts:

```
    $IpLocatorUrl =
      "http://api.hostip.info/get_html.php?&position=true&ip=";
    $IpLocatorUrl = $IpLocatorUrl.$ipAddress;

    // make the HTTP request:
    $filePath = fopen ($IpLocatorUrl, "r");

    // as long as you haven't reached the end of the incoming text:
    while (!feof($filePath)) {
        // read one line at a time, and strip all HTML tags from the line:
        $line = fgetss($filePath, 4096);
        // break each line into fragments at the colon:
        $fragments = explode(":", $line);

        switch ($fragments[0]) {
            // if the first fragment is "country", the second
            // is the country name:
            case "Country":
                $country = trim($fragments[1]);  // trim any whitespace:
                break;
```

»

Continued from opposite page.

```
            // if the first fragment is "Latitude", the second
            // is the latitude:
            case "Latitude":
                // trim any whitespace:
                $lat = trim($fragments[1]);
                break;
            // if the first fragment is "Longitude", the second
            //is the longitude:
            case "Longitude":
                // trim any whitespace:
                $long = trim($fragments[1]);
                break;
        }
    }
    // close the connection:
    fclose($filePath);
```

Now that you've got the location, it's time to find out who you're sending the results to, and format your response appropriately. The information you want is in the HTTP user agent:

```
    // decide on the output based on the client type:
    switch ($userAgent) {
        case "lantronix":
            // Lantronix device wants a nice short answer:
            echo "<$lat,$long,$country>\n";
            break;
        case "processing":
            // Processing does well with lines:
            echo "Latitude:$lat\nLongitude:$long\nCountry:$country\n\n";
            break;
        default:
            // other clients can take a long answer:
            echo <<<END
<html>
<head></head>

<body>
    <h2>Where You Are:</h2>
        Your country: $country<br>
        Your IP: $ipAddress<br>
        Latitude: $lat<br>
        Longitude: $long<br>
</body>
</html>
END;
    }
?>
```

If you call this script from a browser, you'll get the HTML version. If you want to get the "processing" or "lantronix" responses, you'll need to send a custom HTTP request. Try calling it from your terminal program as follows:

First. connect to the server as you did before:

```
telnet example.com 80
```

Then send the following (press Enter one extra time after you type that last line):

```
GET /~yourAccount/ip_geocoder.php HTTP/1.1
HOST: example.com
USER-AGENT: lantronix
```

You should get a response like this:

```
HTTP/1.1 200 OK
Date: Thu, 21 Jun 2007 14:44:11 GMT
Server: Apache/2.0.52 (Red Hat)
Content-Length: 38
Connection: close
Content-Type: text/html; charset=UTF-8

<40.6698,-73.9438,UNITED STATES (US)>
```

If you change the user agent from lantronix to processing, you'll get:

```
HTTP/1.1 200 OK
Date: Thu, 21 Jun 2007 14:44:21 GMT
Server: Apache/2.0.52 (Red Hat)
Content-Length: 64
Connection: close
Content-Type: text/html; charset=UTF-8

Latitude:40.6698
Longitude:-73.9438
Country:UNITED STATES (US)
```

As you can see, this is a powerful feature, and all you need to do to use it is to add one line to your HTTP requests from Processing or the microcontroller (see Chapter 3). Just add an extra *print* statement to send the user agent, and you're all set. In Processing, the HTTP request would now look like this:

```
// Send the HTTP GET request:
String requestString = "/~yourAccount/ip_geocoder.php";
  client.write("GET  " + requestString + " HTTP/1.0\r\n");
  client.write("HOST: example.com\r\n");
  client.write("USER-AGENT: processing\r\n\r\n");
```

The equivalent for Arduino would look like this:

```
  //  Make HTTP GET request. Fill in the path to your version
  //  of the CGI script:
  Serial.print("GET /~yourAccount/ip_geocoder.php HTTP/1.0\
r\n");
  //  Fill in your server's name:
  Serial.print("HOST:example.com\r\n");
  // Print the user agent:
  Serial.print("USER-AGENT: lantronix\r\n\r\n");
```

Using the user agent variable like this can simplify your development a great deal, because it means that you can easily use a browser or the command line to debug programs that you're writing for any type of client.

Mail Environment Variables

Email can be a very flexible way to exchange messages between objects, as well. It affords a more flexible relationship between objects than you get with IP addresses, because it gives you the ability to structure complex conversations. An object can communicate not only who it is (the from: address), but who it would like you to reply to (using the reply-to: field), and whom you should include in the conversation (cc: and bcc: fields). All of that information can be communicated without even using the subject or the body of the messages. PHP gives you simple tools to do the parsing. Because so many devices communicate via email (mobile phone text messaging can interface with email as well), it expands the range of possible devices you can add to a system.

Like HTTP, email protocols have environment variables that you can take advantage of as well. If you've ever viewed the full headers of an email in your favorite mail client, you've seen some of these. To look at mail in more depth, there's a useful PHP extension library you can use, called Net_POP3. It lets you retrieve mail from a mail server and parse the whole exchange from server to client. To use it, download it from pear.php.net/package/Net_POP3. Unzip the downloaded file and copy the file **POP3.php** to your server. (Depending on your server's configuration, you may need additional files; check the documentation for Net_POP3.)

Send It Put the following PHP script on your server:

```
<?php
/*
    mail reader
    language: PHP
*/

include('POP3.php');

// keep your personal info in a separate file:
@include_once("pwds.php");

// new instance of the Net_POP3 class:
$pop3 =& new Net_POP3();

// connect to the mail server:
$pop3->connect($host , $port);

// send login info:
$pop3->login($user , $pass , 'APOP');

// get a count of the number of new messages waiting:
$numMsgs = $pop3->numMsg();

echo "<pre>\n";
echo "Checking mail...\n";
echo "Number of messages: $numMsgs\n";

// get the headers of the first message:
echo($pop3->getRawHeaders(1));
echo "\n\n\n";
echo "</pre>\n";

// disconnect:
$pop3->disconnect();
?>
```

» Next, make a separate file called **pwds.php** on your server. This file contains your username and password info. Keep it separate from the main PHP file so that you can protect it. Format it like this:

```
<?php
$user='username';        // your mail login
$pass='password';         // exactly as you normally type it --
$host='pop.example.com'; // usually pop.yourmailserver.com --
$port="110";             // this won't work on gmail.com and
                         // other servers using SSL
?>
```

As soon as you've saved the **pwd.php** file, change its permissions so that only the owner (you) can read and write from it. From the command line, type:

```
chmod go-rwx pwd.php
```

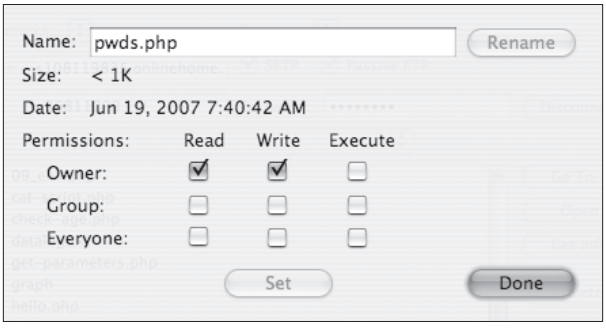


Figure 9-12
Permissions for the **pwds.php** file. Make sure that no one can read from and write to it — besides you.

NOTE: If you're using a graphic SFTP or FTP client, your settings for this file will look like Figure 9-12. This protection will deter anyone who doesn't have access to your account from accessing your account info. It isn't an ideal security solution, but serves for demonstration purposes, and can be made more secure by changing your password frequently.

You'll need to make sure that you have at least one unread mail message on your server for that code to work. When you do, you should get something like this when you open the script in a browser:

```
Checking mail...
Number of messages: 1
Return-Path:
Delivery-Date: Tue, 19 Jun 2007 15:08:33 -0400
Received-SPF: none (mxus2: 12.34.56.78 is neither
permitted nor denied by domain of www.example.com) client-
ip=12.34.56.78; envelope-from=apache@www.example.com;
helo=mx.example.com;
Received: from [12.34.56.78] (helo=mx.example.com)
    by mx.yourserver.com (node=mxus2) with ESMTP (Nemesis),
    id 0MKobQ-1I0j432Ui8-0002oD for youraccount@yourserver.
com; Tue, 19 Jun 2007 15:08:32 -0400
Received: from www.example.com (localhost [127.0.0.1])
    by www.example.com (8.13.1/8.13.1) with ESMTP id
l5JJ8UuJ029983
    for ; Tue, 19 Jun 2007 15:08:31 -0400
Received: (from apache@localhost)
    by www.example.com (8.13.1/8.13.1/Submit) id
l5JJ8Ua2029982;
    Tue, 19 Jun 2007 15:08:30 -0400
Date: Tue, 19 Jun 2007 15:08:30 -0400
Message-Id: <200706191908.l5JK8Ub2129982@www.example.com>
To: youraccount@yourserver.com
Subject: the cat
From: cat@catmail.com
Envelope-To: youraccount@yourserver.com
```

There's a lot of useful information in this header. You can see though the mail says it's from cat@catmail.com, it's actually from a server that's run by example.com. It's common to put an alias on the from: address, or to assign a different reply-to: address than the from: address, or both. It allows sending from a script such as the **cat** script in Chapter 4, yet the reply goes a real person who can answer it. It's important to keep this in mind if you're writing scripts that reply to each other. If you were using email to communicate between networked devices, the program for each device must be able to tell the from: address from the reply-to: address — otherwise, they might not get each other's messages.

This particular message doesn't have a field called X-Mailer:, though many do. X-Mailer tells you which program sent the mail. For example, Apple Mail messages always show up with an x-mailer of Apple Mail, followed by a version number such as (2.752.3). Like the HTTP User Agent, the X-Mailer field can help you to decide how to format mail messages. You could use it in a similar fashion, to tell something about the device that's mailing you, so you can format messages appropriately when mailing back.

Project 26

Email from RFID

This project shows you some of the possibilities of email for communication, and of using email headers for identification. It's an RFID reader that emails you when it's seen one of three tags. It can be used to notify you when certain RFID-embedded objects have been seen at a particular location. For example, if the RFID reader opened a door latch, you'd know whenever people with the appropriate RFID-enabled door tag had been at the door. For simplicity's sake, the system consists of an RFID reader and a Lantronix module — in this case, an XPort. You could easily add a microcontroller to control other things as well.

NOTE: You'll need an XPort, WiMicro, WiPort, or later for this project; the Lantronix Micro can't send mail by itself.

By sending an email notification, you get a lot of information all at once, including:

- Which person you're dealing with (based on the RFID tag number)
- Which reader the person was at (based on the IP address of the XPort that sends the mail)
- What time the person's tag was read (based on the time stamp of the email)

Though only one unit is described here, this project is designed to be duplicated, so that you can get messages from multiple locations. The system is shown in Figure 9-13. The RFID reader is connected to a Lantronix module. The RFID reader sends serial data to the XPort, which sends an email. The output is a PHP script that reads the emails and displays the results.

MATERIALS

- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
- **1 RFID reader module** from Parallax, part number 28140. With a little extra work, the project could be modified to use one of the other readers mentioned in this chapter.
- **3 RFID tags** Get the tags that match your reader. All three of the retailers listed here sell tags that match their readers in a variety of physical packages, so choose the ones you like the best.
- **1 Lantronix embedded device server** Available from many vendors, including Symmetry Electronics (www.semiconductorstore.com) as part number WM11A0002-01 (WiMicro) or XP1001001-03R (XPort). This example uses an XPort.
- **1 RJ45 breakout board** SparkFun part number BOB-00716
- **1 3.3V regulator** The LD1117-33V (SparkFun part number COM-00526) or the MIC2940A-3.3WT (Digi-Key part number 576-1134-ND) work well.
- **1 1µF capacitor** Digi-Key part number P10312-ND
- **1 10µF capacitor** SparkFun part number Comp-10uF, or Digi-Key part number COM-00523
- **1 reset switch** Any momentary switch will do. The ones used here are SparkFun part number COM-00097, or Digi-Key part number SW400-ND.
- **1 5V regulator** The LM7805 series (SparkFun part number COM-00107, Digi-Key part number LM7805CT-ND) work well.
- **1 USB-to-TTL serial adaptor** SparkFun's BOB-00718 from Chapter 2 can do the job. If you use a USB-to-RS-232 adaptor such as a Keyspan or logear dongle, refer to Chapter 2 for the schematics to convert RS-232-to-5V TTL serial. You'll use this for configuring the XPort only. If you've got more than one, it'll be handy for troubleshooting, but you won't need one for the final project.

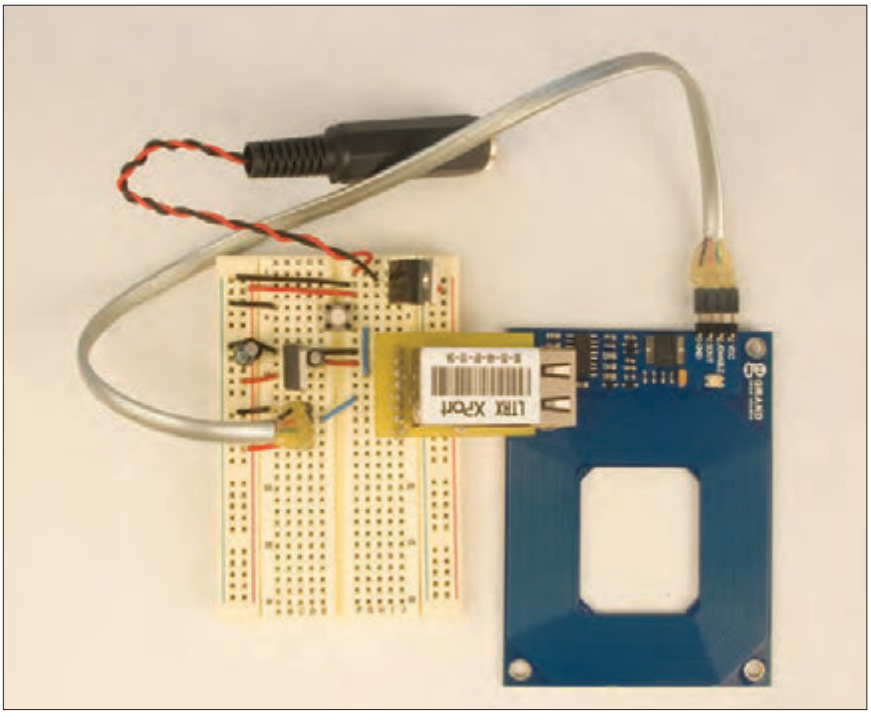
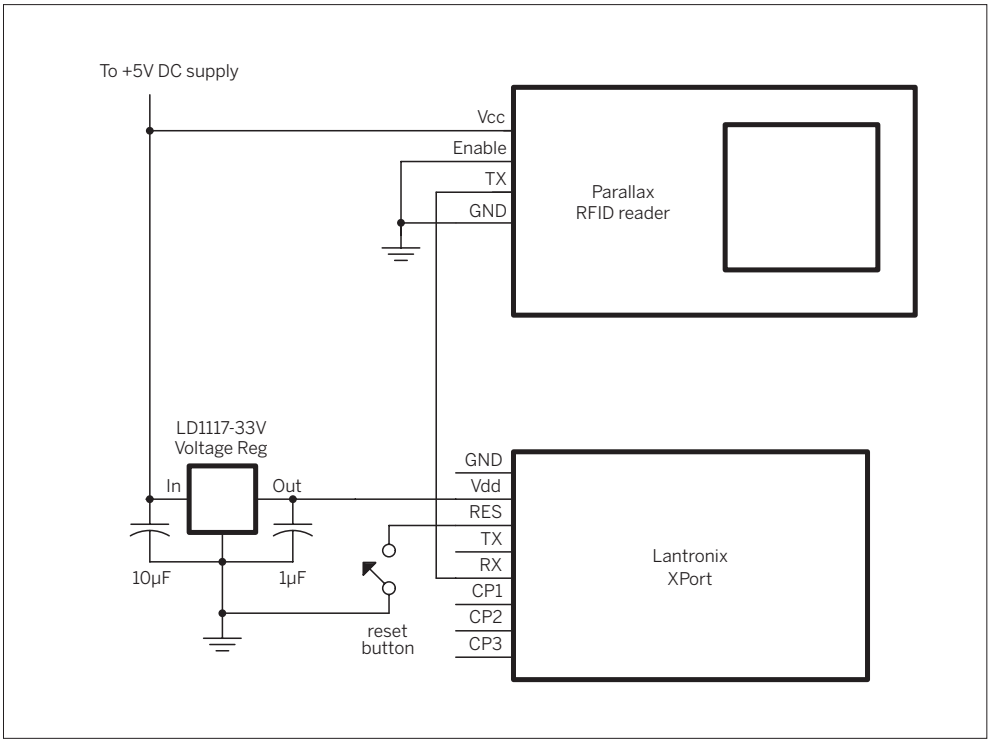


Figure 9-14
The RFID emailer circuit.



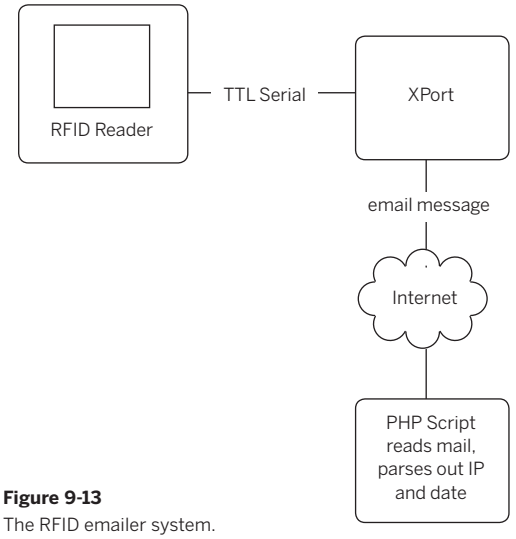


Figure 9-13
The RFID emailer system.

The Circuit

The circuit is fairly straightforward. The RFID reader's serial output is connected to the XPort's serial receive pin. The reader's enable pin is connected to ground. That's it. Figure 9-14 shows the circuit.

To make this work, you'll configure the XPort to send emails. It can automatically generate an email to a preset address when you send it a given serial string. To determine the strings to send, you'll need to read the tags with the RFID reader first. To do this, connect the RFID reader's output to the USB-to serial adaptor's input and connect the reader's enable pin to ground, as shown back in Figure 9-6. Then connect the USB-to-serial adaptor to your computer and open a serial terminal connection to it at 2400 bps. When you wave the tags in front of it, you should see strings like this:

```
0415AB6FB7
0415AB5DAF
0415AB5DAF
0F008F7CE8
0F008F7CE8
```

As you know from the previous projects, each unique string represents a unique tag. Pick two characters that are unique to each tag. In the example, you could use the last two characters — B7, AF, and E8. The values will be different for your tags. Write them down.

Now disconnect the RFID reader and connect the XPort to the USB-to-serial adaptor as shown in Figure 9-15. Open a serial terminal connection at 9600bps.

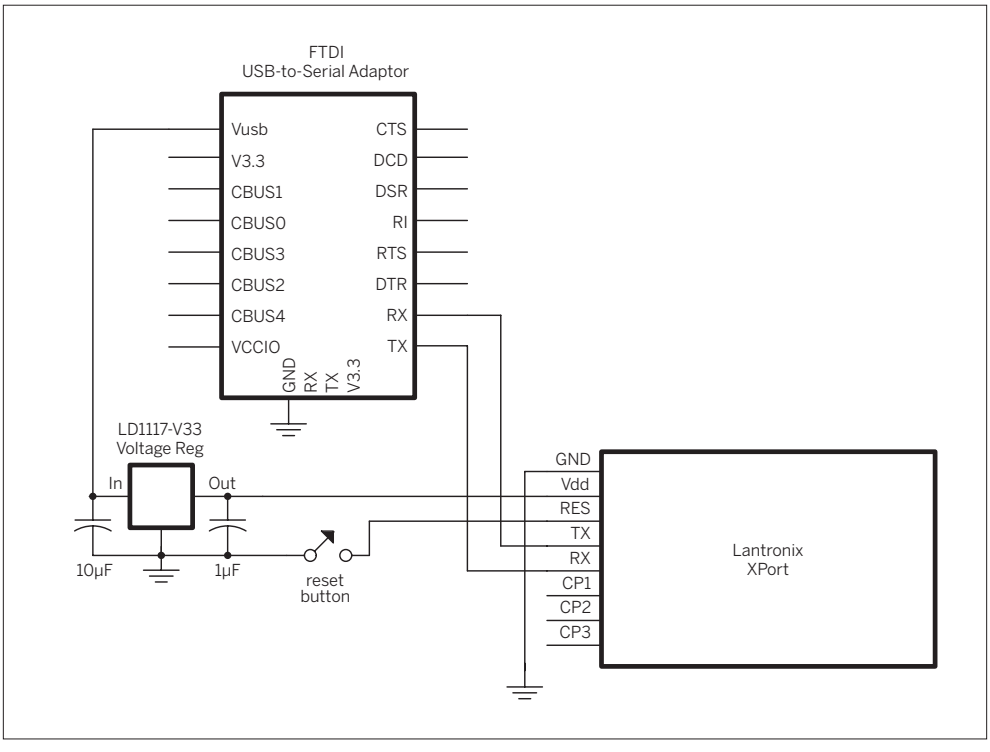
Hold down the x key on your keyboard and press the reset button on the breadboard to reset the XPort. You'll get the usual configuration menu. Type 1 to get the serial setup menu, and enter the following settings:

```
Baudrate (2400) ?
I/F Mode (4C) ?
Flow (00) ?
Port No (10001) ?
ConnectMode (D4) ?
Send '+++' in Modem Mode  (N) ?
Auto increment source port  (N) ?
Remote IP Address : (0) .(0) .(0) .(0)
Remote Port  (0) ?
DisConnMode (00) ?
FlushMode   (00) ?
DisConnTime (00:00) ?:
SendChar 1  (00) ?
SendChar 2  (00) ?
```

Then choose menu item 3 to set the email settings. The XPort can be configured with your email address and SMTP server, and can send up to three different email notifications based on various events. It can send mail based on incoming serial messages, or based on changes on its configurable I/O pins. For this project, you'll use serial messages.

Unfortunately, the XPort expects the bytes you'll send it as hexadecimal values. This means that if you're sending the bytes shown earlier, for example, you'd need to convert them from ASCII to hexadecimal as follows:

ASCII characters	Hexadecimal values
"B7"	0x42, 0x37
"AF"	0x41, 0x46
"E8"	0x45, 0x38



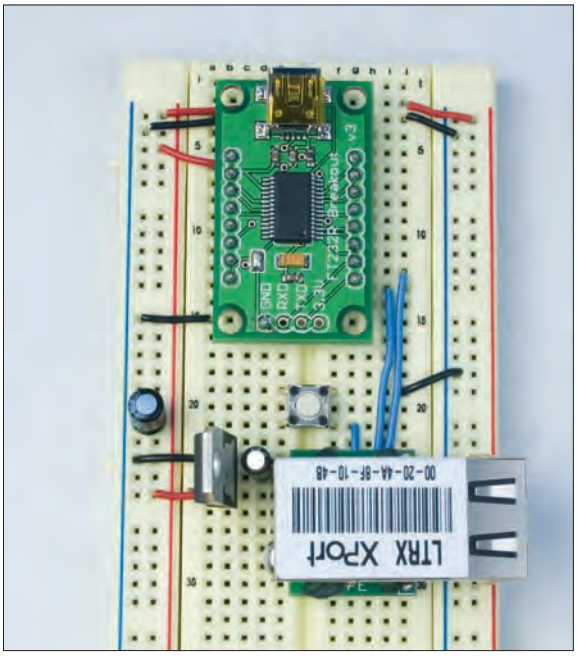


Figure 9-15
XPort connected to a USB-to-serial adaptor.

The first items you'll need to set for menu item 3 are the SMTP mail server IP and your login info. To get your mail server's numeric IP address, you can ping it. For example, if your mail server is smtp.yahoo.com, open a terminal window and type:

```
ping –c 2 smtp.yahoo.com
```

You'll get a reply starting with a line containing the IP you need:

```
PING smarthost.yahoo.com (216.145.54.172): 56 data bytes
```

Here are some initial mail settings for the XPort. Replace these values with the appropriate ones for your server:

```
Mail server () ? (216) .(145) .(54) .(172)
Unit name () ? myAccountName
Domain name () ? example.com
Recipient 1 () ? myAccountName@example.com
Recipient 2 () ?
```

Once you've got the account set up, it's time to set the serial bytes that will trigger the messages, and the messages themselves. Replace the hexadecimal values shown here with your own values:

```
- Trigger 1
Enable serial trigger input (N) ? Y
  No. of bytes (2) ? 2
  Match (,) ? 42,37
Trigger input1 [A/I/X] (X) ?
Trigger input2 [A/I/X] (X) ?
Trigger input3 [A/I/X] (X) ?
Message () ? Tag one
Priority (L) ?
```

Next, set the minimum notification interval and the re-notification interval. The former sets how many seconds have to elapse at minimum between email messages. Because the RFID reader reads repeatedly, set this interval fairly high, so as not to flood the network with emails. The renotification interval sets how soon the XPort should repeat an email if it gets the same trigger string twice. Again, set this fairly high:

```
Min. notification interval (1 s) ? 10
Re-notification interval (0 s) ? 20
```

Repeat the operation for messages 2 and 3, changing the message itself and the trigger strings appropriately. Here are my settings for messages 2 and 3:

```
Enable serial trigger input (N) ? Y
  No. of bytes (2) ? 2
  Match (,) ? 41,46
Message () ? Tag Two

Enable serial trigger input (N) ? Y
  No. of bytes (2) ? 2
  Match (,) ? 45,38
Message () ? Tag Three
```

When you're finished, choose menu item 9 to save your settings and reset the XPort.

NOTE: The XPort listens for the serial reset message (xxx) only at 9600 bps. So if you're trying to get to the setup menu, make sure that you're connected to the serial port at 9600 bps. Because the RFID reader needs to be at 2400 bps, you might accidentally open the port at the wrong rate.

Now connect the RFID reader to the XPort, as shown in Figure 9-14. Connect it to the Internet, power it up, and wave a tag in front of the reader. Then check your email. You should see a message like this:

```
From:    myAccountName@example.com
Subject: Notification: Tag one
Date:    June 21, 2007 6:11:59 PM EDT
To:      myAccountName@example.com
```

When you get this email, you know everything's working. Try the other two tags. You should get a unique message for each one.

You could duplicate this circuit for several locations, but unless you set up an email address for each XPort, the email from the XPorts all comes from the same account. How would you know which XPort sent the mail? Easy: check the IP address of the sender. Open the full header and look for this string:

```
Received: from myAccountName ( [12.34.56.78])
```

That IP address will be the IP address of your XPort. To distinguish between different XPorts, you'd need a program to look for the IP address of the sender. You could get the identity of the person with the RFID tag from the subject, and the time of the tag read from the email's timestamp. This gives you a pretty good picture of who was where at what time.

Try It Following is a PHP script to look for these messages. It filters out all other mail messages and just reports the notifications from the XPort. It needs the same **POP3.php** library and the same **pwd.php** file as the previous program. Save it in the same directory as **rfid_mail_reader.php**:

```
<?php

/*
    RFID mail reader
    language: PHP

    Parses a POP email box for a specific message from an XPort.
    The message looks like this:

    From:    myAccountName@myMailhost.com
    Subject: Notification: Tag one
    Date:    June 21, 2007 6:11:59 PM EDT
    To:      myAccountName@myMailhost.com


*/
include('POP3.php');

// keep your personal info in a separate file:
@include_once("pwds.php");

echo "Checking mail...";

// New instance of the Net_POP3 class:
$pop3 =& new Net_POP3();

// Connect to the mail server:
$pop3->connect($host , $port);

// Send login info:
$pop3->login($user , $pass , 'APOP');

// Get a count of the number of new messages waiting:
$numMsgs = $pop3->numMsg();

echo "<pre>\n";
echo "Number of messages: $numMsgs\n";

// iterate over the messages:
for ($thisMsg = 1; $thisMsg <= $numMsgs; $thisMsg++) {
    // parse the headers for each message into
    // an array called $header:
    $header = $pop3->getParsedHeaders($thisMsg);

    // print the subject header:
    $subject = $header["Subject"];
    // look for the word "Notification" before a colon
    // in the subject:
    $words = explode(":", $subject);
```

When you open this in a browser, you should get output like this:

```
Checking mail...

Number of messages: 234
Tag three showed up at address
12.34.56.78 at
     Thu, 21 Jun 2007 17:16:56 -0400 (EDT)
 Tag one showed up at address
12.34.56.89 at
     Thu, 21 Jun 2007 17:17:10 -0400 (EDT)
 Tag two showed up at address
12.34.56.78 at
     Thu, 21 Jun 2007 17:17:11 -0400 (EDT)
 Tag two showed up at address
12.34.56.89 at
     Thu, 21 Jun 2007 18:11:59 -0400 (EDT)
 Tag two showed up at address
12.34.56.89 at
     Thu, 21 Jun 2007 18:12:12 -0400 (EDT)
That's all folks
```

Continued from opposite page.

```
    // only do the rest if this mail message is a notification:
    if ($words[0] == "Notification"){
        // get the second half of the subject; that's the tag ID:
        $idTag = $words[1];
        // print it;
        echo "$idTag showed up at address\t";

        /*
            the IP address is buried in the "Received" header.
            That header is an array. The second element contains
            who it's from.  In that string, the IP is the first
            thing contained in square brackets. So:
        */

        // get the stuff in the right array element after the
        // opening square bracket:
        $receivedString = explode("[", $header["Received"][1]);
        // throw away the stuff after the closing bracket:
        $recdString2 = explode("]", $receivedString[1]);
        // what's left is the IP address:
        $ipAddress = $recdString2[0];

        // print the IP address:
        echo "$ipAddress at \t";

        // print the date header:
        $date = $header["Date"];
        echo "$date\t";
        echo "\n";
    }

}

echo "That's all folks";
echo "</pre>";

// disconnect:
$pop3->disconnect();
?>
```

You can see that there were many more messages than the script printed out (234 messages, only 5 shown here). You can also see that there were two different XPorts reporting (12.34.56.78 and 12.34.56.89). Finally, you've got the time, location, and ID of every tag that showed up in your system. You've got a information about both identity and activity, just using the headers of email messages.

X

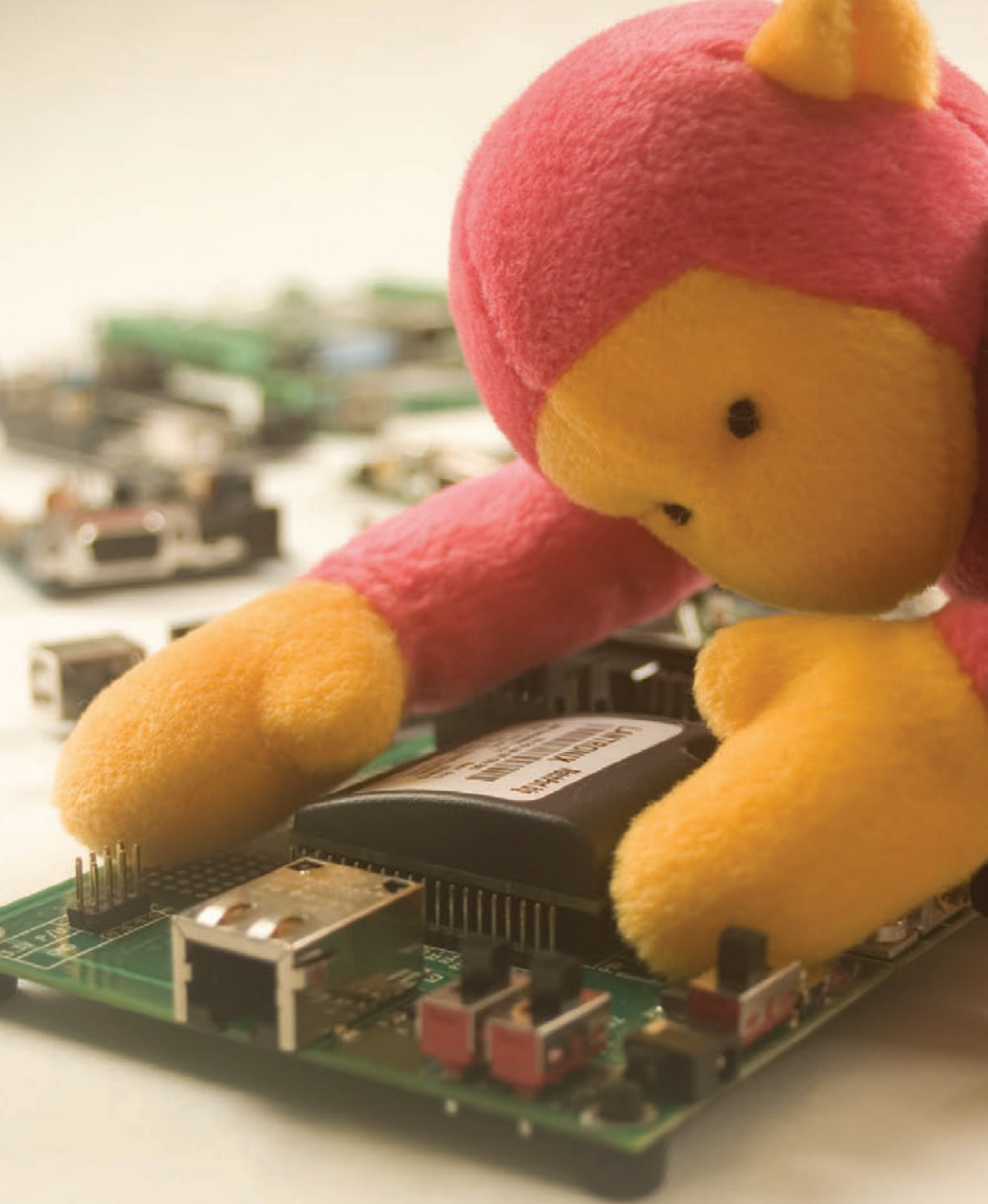
Conclusion

The boundary between physical identity and network identity always introduces the possibility for confusion and miscommunication. No system for moving information across that boundary is foolproof. Establishing identity, capability, and activity are all complex tasks, and the more that you can incorporate human input into the situation, the better your results will be.

Security is essential when you're transmitting identifying characteristics, in order to maintain the trust of the people using what you make and to keep them and yourself safe. Once you're connected to the Internet, nothing's truly private, and nothing's truly closed, so learning to work with the openness makes your life easier. In the end, keep in mind that clear, simple ways of marking identity are the most effective, whether they're universal or not. Many beginners and experienced network professionals often get caught on this point, because they feel that identity has to be absolute, and clear to the whole world. Don't get caught up in how comprehensively you can identify things at first. It doesn't matter if you can identify someone or something to the whole world — it only matters that you can identify them for your own purposes. Once that's established, you've got a foundation on which to build.

When you start to develop projects that use location systems, you usually find that less is more. It's not unusual to start a project thinking you need to know position, distance, and orientation, then pare away systems as you develop the project. The physical limitations of the things you build and the spaces you build them in will solve many problems for you.

X



Appendix A

MAKE: PROJECTS

And Another Thing

This book only touches the tip of the iceberg in terms of how you can connect physical devices to networks. There are many tools and applications that were left out because there wasn't enough space to explain them adequately. Other tools came on the market as I was writing, leaving not enough time to try them out thoroughly enough to write about them. Still others were left out because they were similar to the main tools described already. Following is a collection of pieces that didn't make the main text, but that still may be useful to you as you're networking physical objects.

Other Useful Protocols

There are many more useful protocols than have been covered. Here are a few that you might run across, along with some notes on where to begin learning about them.

MIDI

The Musical Instrument Digital Interface (MIDI) protocol is a protocol for real-time communication between digital musical instruments. It's the granddaddy of digital synthesizer protocols. Most music synthesizers, sequencers, samplers, keyboards, and workstations on the market today speak MIDI. If you plan to make music using hardware, you're going to run across it. It's a serial protocol running at 31,250 bps. There's a standard MIDI connector called a DIN5 connector that you'll find on all MIDI gear. All the connectors on the gear are female plugs, and the cables all have male connectors on both ends. Figure A-1 shows the MIDI connector and a simple MIDI output circuit from an Arduino board. You can find hundreds of examples of how to send and receive MIDI data online.

MIDI messages are divided into three or more bytes. The first byte, a command byte, is always greater than 127 in value. Its value depends on the command. The bytes that follow it are called status bytes. All status bytes have values less than 128. This makes it possible to tell a command byte from a status byte by the value alone.

There are a number of different MIDI commands. The most basic, note on and note off messages, control the playing of notes on 16 different channels of a synthesizer. Each note on or note off command contains two status bytes, specifying the pitch in a range from 0 to 127, and the velocity (how hard the note should be struck) from 0 to 127. Pitch value 69 is defined as A above middle C (A440) by the general MIDI specification. The general MIDI spec also covers the instruments that you're likely to find on each channel.

As MIDI instruments, channels, and banks of sounds are grouped in groups of 16, MIDI messages are generally written in hexadeximal notation. This makes it easy to read commands based on the value. For example, 0x80 to 0x8F are all note off messages, 0x90 to 0x9F are all note on messages. 0x90 is note on channel 1, 0x91 is note on channel 2, and so forth.

For more information on MIDI, see Paul D. Lehrman and Tim Tully's book *MIDI for the Professional* (Amsco, 1993).

Here is the "Hello World!" of MIDI, a program to send MIDI notes to a synthesizer. It plays notes. To use it, build the circuit as shown, connect it to a MIDI synth, and connect the synth to an amplifier and speakers.

```
/*
     MIDI
    Language: Wiring/Arduino

    plays MIDI notes from 30 to 90 (F#-0 to F#-5)
*/

char note = 0;      // The MIDI note value to be played

void setup() {
 // Set MIDI baud rate:
 Serial.begin(31250);
}

void loop() {
 // play notes from F#-0 (30) to F#-5 (90):
 for (note = 0; note < 127; note ++) {
 // Note on channel 1 (0x90), some note value (note), middle velocity
 // (0x45):
```

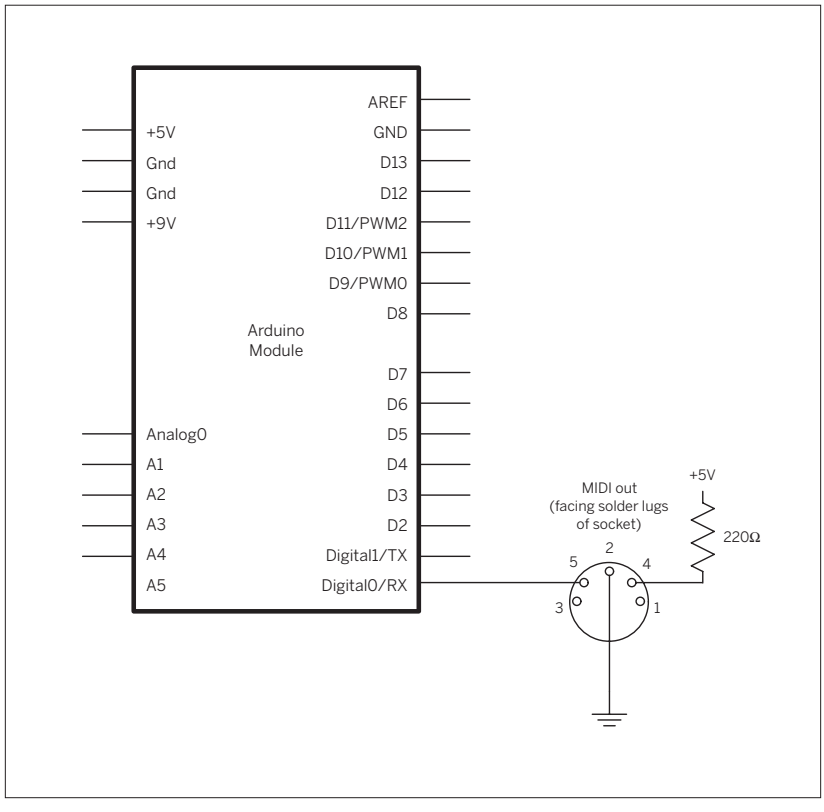
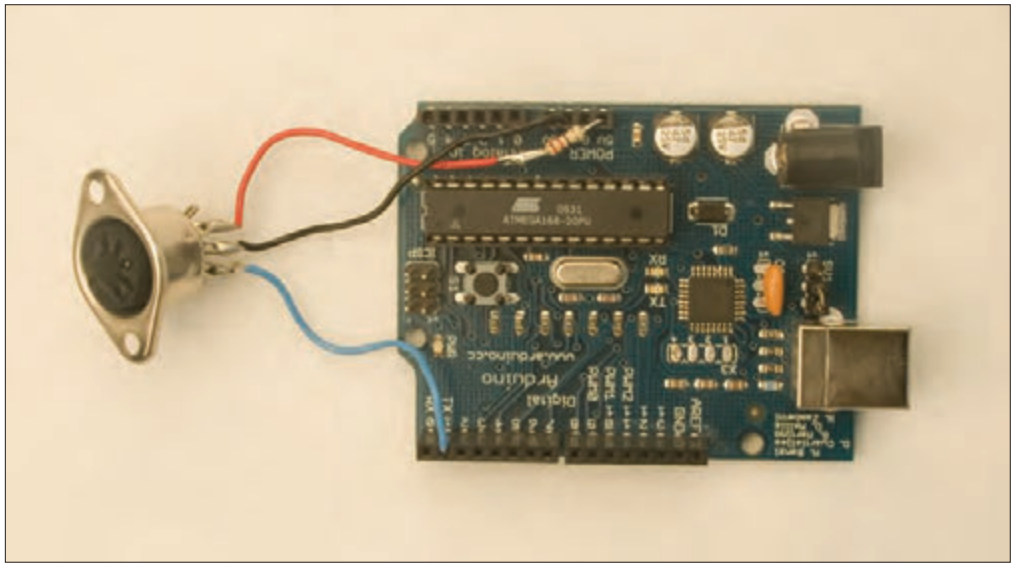


Figure A-1
MIDI output from a microcontroller.



Continued from previous page.

```
  noteOn(0x90, note, 0x70);
  delay(10);
  //Note on channel 1 (0x90), some note value (note), silent velocity (0x00):
  noteOn(0x90, note, 0x00);
  delay(10);
 }
}

// plays a MIDI note. Doesn't check to see that
// cmd is greater than 127, or that data values are less than 127:
void noteOn(char cmd, char data1, char data2) {
 Serial.print(cmd, BYTE);
 Serial.print(data1, BYTE);
 Serial.print(data2, BYTE);
}
```

OpenSound Control (OSC)

OpenSound Control (OSC) was created as a successor to MIDI. MIDI is aging as a protocol. As it's been expanded to other uses, MIDI's limitations have become more apparent. Compared to modern protocols, MIDI's data rate is relatively low. Furthermore, MIDI doesn't travel well over packet networks. OSC was designed to be implemented over many transport protocols, from serial to UDP to whatever comes next. When formatting OSC messages, you define an address space to define the device. For example, you can use OSC to control the MAKE controller (described later in this appendix). Each set of functions has an address. For example, to read the third analog input channel, the address space would be:

```
/analogin/2/value
```

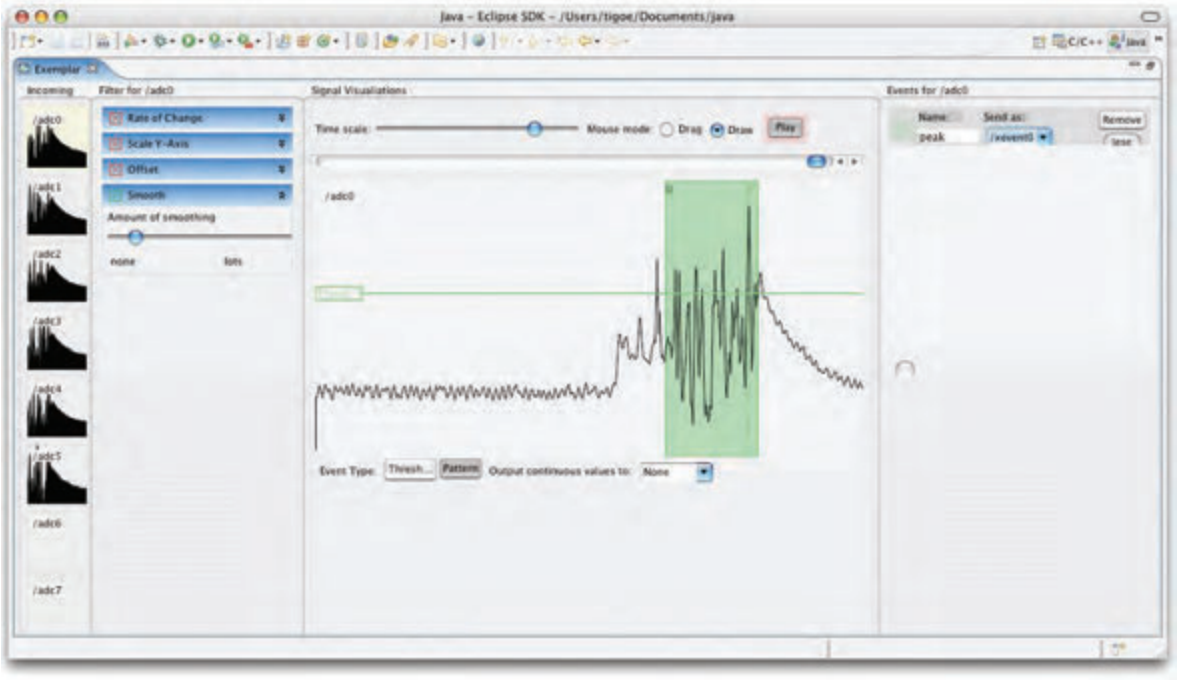
The controller would then send you back the value of that analog channel. OSC is designed to be flexible enough to allow for control of devices that haven't even been invented yet. Because you define the address space, you can define any set of devices and functions that you want.

OSC has been implemented on many different platforms, including Flash, PHP, C, C++, Java, Max/MSP, PD, Processing and more. Though there's not yet an official library for OSC in Wiring or Arduino, there may be soon. For more information on OSC, see the OSC homepage at www.cnmat.berkeley.edu/OpenSoundControl/.

Exemplar (shown in Figure A-2) is a toolkit for prototyping physical devices that uses OSC to communicate with various microcontroller modules. It allows you to read data from the microcontroller and prototype onscreen applications that use that data without having to write a lot of code. The Exemplar firmware for Arduino is a good simple example of how to implement OSC in code. For more on Exemplar, including code to communicate from Wiring and Arduino to Exemplar using OSC, see hci.stanford.edu/research/exemplar/. It's both an analysis tool and a proxy tool, like those covered further on.

DMX512

DMX512 is a real-time serial protocol for communicating between stage-lighting control systems and lighting dimmers. It has been the industry standard for stage lighting and show control equipment for a couple of decades now. It's also used to control special effects machines, moving lights, projection systems, and more. It's fast for a serial protocol, at 250 kbps; fast enough that you can't just send regular serial data from a microcontroller. There are a few examples online as to how to send DMX512 from microcontrollers, though. Numerous examples for the PIC microcontrollers exist, and pointers for implementing it on Arduino can be found on the Arduino playground site at www.arduino.cc/playground/Learning/DMX. For more on DMX, see www.opendmx.net. Like MIDI, DMX is aging as a protocol. The lighting industry has started to develop its successor, Advanced Controller



Network, or ACN, which can run over Ethernet. Over the next few years, you can expect to see it grow in prominence. For more on show control protocols in general, John Huntington's book *Control Systems for Live Entertainment* (Focal Press, 2007) can't be beat.

Figure A-2
The Exemplar interface. You can view the outputs of sensors on time graphs, filter them, match patterns, and have those patterns generate messages that can be sent to other applications.

Proxies of All Kinds

Many of the software programs you've written in this book are basically proxies, converting one form of communication to another. There are a number of useful programs on the market that convert from one protocol to another in order to enable two devices that don't speak the same language to communicate. Here are a few examples.

Network Serial Proxy

One of the most common tasks in networked projects is converting a stream of serial data to a network TCP stream. Most developers of physical computing projects have written their own network serial proxy, so there are many examples to look at. It's a particularly useful thing to do when you want to communicate between applications that can't access a serial port, like Adobe Flash, and a microcontroller. The next page shows just such a proxy.

Here's a very basic proxy sketch written in Processing that works with Flash. It will work with other applications, too, but it's optimized for ActionScript's XMLSocket class, which expects every message to be wrapped in bytes of value 0.

```
/*
 Serial Server
 language: processing

 This program makes a connection between a serial port
 and a network socket.

*/

import processing.serial.*;
import processing.net.*;

int socketNumber = 9001; // the port the server listens on
Server myServer;         // the server
Client thisClient;       // the reference to the client that logs
char terminationString = '\0'; // zero terminator byte

Serial myPort;            // the serial port you're using
String portnum;           // name of the serial port
String outString = "";    // the string being sent out the serial port
String inString = "";     // the string coming in from the serial port
String socketString = ""; // string of bytes in from the socket
int receivedLines = 0;    // how many serial lines have been received
int bufferedLines = 5;    // number of incoming lines to keep

void setup() {
 size(400, 300);          // window size

 // create a font with a font available to the system:
 PFont myFont = createFont(PFont.list()[2], 14);
 textFont(myFont);

 // list all the serial ports:
 println(Serial.list());

 // based on the list of serial ports printed from the
 //previous command, change the 0 to your port's number:
 portnum = Serial.list()[0];
 // initialize the serial port:
 myPort = new Serial(this, portnum, 9600);
 // buffer until a newLine:
 myPort.bufferUntil('\n');
 // start the server:
 myServer = new Server(this, socketNumber); // Starts a server

}

void draw() {
 // clear the screen:
 background(0);
 // print the name of the serial port:
```

Continued from opposite page.

```
 text("Serial port: " + portnum, 10, 20);
 // Print out what you get:
 text("From serial port:\n" + inString, 10, 80);
 text("From socket:\n" + socketString, 200, 80);

 // if the client is not null, and says something, display
 // what it said:
 if (thisClient !=null) {
  // print out the current client:
  text("Active client: " + thisClient.ip(), 10, 60);
  // read what the client said:
  String whatClientSaid = thisClient.readString();
  if (whatClientSaid != null) {
   // save what it said to print to the screen:
   socketString = whatClientSaid;
   // send what it said out the serial port:
   myPort.write(socketString);
  }
 }
}

// this method runs when bytes show up in the serial port:
void serialEvent(Serial myPort) {
 // read the String from the serial port:
 String whatSerialSaid = myPort.readStringUntil('\n');
 if (whatSerialSaid != null) {
  // save what it said to print to the screen:
  inString = whatSerialSaid;
  // if there is a netClient, send the serial stuff to them:
  if (thisClient != null) {
   // put a zero byte before and after everything you send to Flash:
   thisClient.write(terminationString);
   // send the actual text string:
   thisClient.write(inString);
   // add the end zero byte:
   thisClient.write(terminationString);
  }
 }
}

void serverEvent(Server myServer, Client someClient) {
 if (thisClient == null) {
  // don't accept the client if we already have one:
  thisClient = someClient;
 }
}
```

Try It Here's a sample of ActionScript to test it with. Paste this into the action window in the first frame of a new Flash movie and run it. Thanks to Dan O'Sullivan for this code:

```
/*
    Socket Test
    Language: ActionScript 2.0

    Exchanges strings through an XMLSocket
*/

var i = 0;        // counter for the number of clicks

createSocket();

function createSocket() {
    socket = new XMLSocket();
    // 127.0.0.1 is the same as "localhost"
    // i.e. an alias to your local machine
    socket.connect("127.0.0.1",9001);
    // define the functions that get called when these events happen:
    socket.onConnect = success;
    socket.onClose = closed;
    socket.onData = newData;
}

// when you click on the screen, Flash sends a click out
// the server port:
function onMouseUp() {
    trace("click");
    socket.send("click ");
    socket.send(i);
    i++;
}

function success() {
    trace("socket opened");
    socket.send("F\n"); // tell proxy it is talking to Flash
}

function closed() {
    trace("socket closed");
    socket.send("Q\n"); // tell proxy that Flash is closing socket
}

function newData(inString) {
    trace(inString); // trace the packet of data to the Flash
                     // output screen
}
```

Figure A-3
The Tinker.it AppleScript proxy. Any character coming in the serial port can be assigned to an AppleScript command.

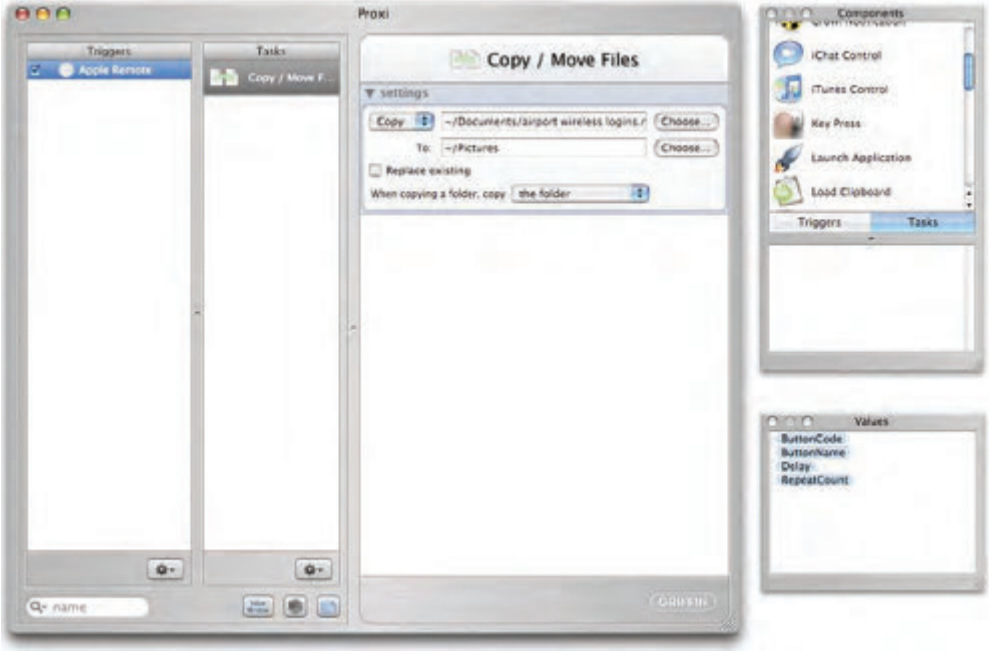


Figure A-4
Griffin Proxi. Connections are made by pulling triggers and tasks from their bins to the main window and setting their characteristics.

Other Proxy Programs

There are numerous programs on the Web that act as proxies, receiving messages from one source and passing it to another. Following are a few that are popular among physical computing enthusiasts.

TinkerProxy

The folks at Tinker.it have written a useful application for Mac OS X that receives serial information and generates AppleScript events for Mac OS X. This app (Figure A-3) allows you to control nearly any application on a Macintosh,

as most applications have at least rudimentary AppleScript controls. They've also written TinkerProxy, a TCP-to-serial proxy for Windows. See tinker.it/now/category/software/ for more details.

NADA

Sketchtools NADA is a proxy tool that takes in MIDI, serial OSC, and other messages and sends them out to various development environments like Flash, or Java. Originally designed to work with the Phidgets line of hardware tools, it can work with the MAKE Controller, MakingThings' earlier Teleo tools, any MIDI device, and many others as well. See www.sketchtools.com for more details.

Griffin Proxi

Griffin Proxi (Figure A-4, preceding page) is a tool that runs various operating system tasks in Mac OS X in response to events like a mouse click, keyboard hit, incoming email, and more. You can use it to generate screen messages, make sounds, run AppleScripts, write to files and send email and test messages from various hardware devices connected to your computer. See proxi.griffintechnology.com for more details.

Girder

Girder is an operating system automator for Windows that lets you generate serial, network and X10 events from operating system events. Like the others, it makes it possible to connect various operating system events with network and serial messages without programming. See www.girder.nl for details.

x

Mobile Phone Application Development

Mobile phone development is limited by a number of factors. To begin with, there are many more phone operating systems than there are desktop operating systems. This makes portability of software difficult. The learning curve for beginners is an order of magnitude greater than what you've encountered here. In addition to that, access to both the network and to many features of the phones is often limited by the mobile service providers. For these reasons, I didn't include mobile phones in this book, even though they are an exciting platform for these kinds of applications.

If you're interested in mobile phone application development, look into phones running the Symbian Series 60 operating system. Linux is beginning to make a strong showing on phones, too. Most of these are Nokia phones. They can run the Java Micro Edition (formerly called J2ME, now Java ME), which is a powerful and fairly accessible toolkit for phone programming. There is a limited variation of Processing called Mobile Processing that runs on these phones as well as most Java-capable phones. In addition, there is a version of the Python scripting language. Python for Series 60 is exciting, because it allows you access to features of the phone that Java and Mobile Processing don't. For example, you can make phone calls and find the cell tower ID using Python. If you know Adobe's Flash programming environment, you should look into Flash Lite, which runs on mobile phones as well.

The most productive approaches to mobile phone development lie in three directions: developing for the browser, for SMS, and for voice and touchtone connections. For the browser, you can use WAP/WML (Wireless Application Protocol/Wireless Markup Language), or you can use HTML. Because SMS messages can travel across email servers, you can use the POP, IMAP, and SMTP mail protocols you saw earlier in this book to send and transmit them. Asterisk is an open-source private branch exchange (PBX) for managing telephony, allowing you to make and receive audio phone calls from a server.

Browsers on the Phone

Because most mobile phones on the market as of this writing have at least a rudimentary web browser, you can develop web applications for them. Some of them use WAP/WML only, but increasingly many of them can read plain old HTML. The functionality of low-end phones is limited, so it's not wise to do lots of graphics-heavy pages, but you can easily browse text pages with a few small graphics on a phone. Any of the HTML pages generated by the PHP scripts in this book should be viewable on most mobile phones. You can even customize the HTML output for mobile phones by reading the HTTP User Agent as you saw in Chapter 9, and outputting a special page for mobile phone browsers.

HTTP User Agent for Phones

Here's a PHP script that runs on most web-enabled phones. It identifies the user agent and IP address and mails it back to you. It's a useful diagnostic tool for finding phones and knowing their browsers. To use this, save this PHP script to your server, then open the script in a browser on your phone:

```
<?php
/*
    Phone Finder
    Language: PHP

    Identifies the user agent and IP address and sends an email
    notification. Runs on mobile phones.

*/
 $userAgent = getenv('HTTP_USER_AGENT');
 $ipAddress = getenv('REMOTE_ADDR');
 sendMessage('you@example.com', 'user agent',
   "$userAgent $ipAddress");

     echo <<<END
<html>
<head></head>

<body>
    <h2>Hi There</h2>
        Your IP: $ipAddress<br>
        Your browser: $userAgent<br>
        Thanks!
</body>
</html>
END;

 end;

function sendMessage($to, $subject, $message) {
    $from = "phone@example.com";
    mail($to, $subject, $message, "From: $from");
}
 ?>
```

You'll get a number of different replies. Here are some examples:

```
Mozilla/5.0 (SymbianOS/9.1; U; en-us) AppleWebKit/413
(KHTML, like
  Gecko) Safari/
NokiaN73-1/2.0628.0.0.1 S60/3.0 Profile/MIDP-2.0
Configuration/
  CLDC-1.1
NokiaN93-1/20.0.058 SymbianOS/9.1 Series60/3.0 Profile/
MIDP-2.0
  Configuration/CLDC-1.1
Nokia6230/2.0 (05.51) Profile/MIDP-2.0 Configuration/
CLDC-1.1
  UP.Link/6.3.0.0.0
BlackBerry8100/4.2.0 Profile/MIDP-2.0 Configuration/CLDC-1.1
  VendorID/100
BlackBerry7290/4.1.0 Profile/MIDP-2.0 Configuration/CLDC-1.1
  VendorID/100 216.9.250.99
LGE-PM325/1.0 UP.Browser/6.2.3.2 (GUI) MMP/2.0
Palm680/RC1 Mozilla/4.0 (compatible; MSIE 6.0; Windows 98;
  PalmSource/Palm-D053; Blazer/4.5) 16;320x320
UP.Link/6.3.0.0.
Mozilla/2.0 (compatible; MSIE 3.02; Windows CE; PPC;
240x320)
  BlackBerry8100/4.2.0 Profile/MIDP-2.0 Configuration/
CLDC-1.1
  VendorID/100
Mozilla/4.0 (compatible; MSIE 6.0; Windows 98; PalmSource/
Palm-D052;
  Blazer/4.5) 16;320x320 68.28.123.118
SAMSUNG-SGH-A707/1.0 SHP/VPP/R5 NetFront/3.3 SMM-MMS/1.2.0
  profile/MIDP-2.0 configuration/CLDC-1.1 UP.Link/6.3.0.0.0
209.183.32.17
Mozilla/4.0 (MobilePhone RL-4920/US/1.0) NetFront/3.1
MMP/2.0
Mozilla/4.0 Sprint:MotoQ (compatible; MSIE 4.01; Windows CE;
  Smartphone; 176x220)
```

As you can see, there's a wide variety of mobile phone browsers out there, so telling what's a phone and what's not can be a challenge. That's a good reason to keep your display output simple if you're planning for it to be seen on mobile phones. Some of the browsers conveniently report their screen size. For example, the final one here, a MotoQ, has a screen size of 176 x 220. For more information on the capabilities of various handsets, check out the WURFL project at wurfl.sourceforge.net.

SMS Text Messaging

SMS is an easy way to get and receive messages from mobile phones, because it's just email, from a server's point of view. Sending an email from a mobile phone via SMS varies with each carrier, but it generally works like this. Type in the short code that your carrier uses for sending email via SMS. You can get this from the carrier's website. For example, T-Mobile in the United States uses the shortcode `500`. Verizon just requires you to send directly to the email address. Each carrier is different, but here's a general approach:

1. Type the email address you want to send to, followed by a space.
2. Type in the subject line, followed by a space.
3. Type # then the body of the message.
4. Hit send.
5. Check your email for the message.

So on my T-mobile phone, I type:

```
To: 500
you@yourmailserver.com hello there #this is the body
```

In your mail, you should receive:

```
From: 19175555555@tmomail.net
To: you@yourmailserver.com
Subject: hello

this is the body
```

You can parse these messages just like mail using the PHP NET_POP3 class.

Sending text messages is just as easy. You need to know the carrier's mail domain, which you can get from the carrier's website. Here are a few of the U.S. carriers' mail domains, as of this week. Mergers and acquisitions could make these domains incorrect by the time you read this, so check with your carrier. Replace the number with your own:

```
T-Mobile: 12225555555@tmomail.net
Sprint: 12225555555@messaging.sprintpcs.com
Verizon: 12225555555@vtext.com
AT&T: 12225555555@txt.att.net
Nextel: 12225555555@messaging.nextel.com
```

Try It To send an SMS, just send mail to the ten-digit phone number at the recipient's carrier. Here's a PHP script to send yourself an SMS. Save this as **sms.php**:

> **Consider password-protecting this script, or removing it after you're done testing. It could become the target of abuse if it's found by a roving spambot with a pocket full of phone numbers.**

There's an extra carrier in the list of carriers in that code that you may not be familiar with. Teleflip (www.teleflip.com) is a text message aggregator service. It allows you to send to any U.S. 10-digit phone number by sending to number@teleflip.com. This frees you from having to look up the carriers' mail host names.

```
<?php
/*
    SMS messenger
    Language: PHP

    Sends SMS messages

*/
    $phoneNumber = $_GET["phoneNumber"];    // get the phone number
    $carrier   = $_GET["carrier"];          // get the carrier
    $message   = $_GET["message"];          // get the message
    $recipient = $phoneNumber."@".$carrier; // compose the recipient

    // if there's a phone number in the form, you can send:
    if ($phoneNumber != null) {
        // send the message:
        $from = "you@yourmailhost.com";
        mail($recipient, "Mail message", $message, $from);
     }

     // finally, print the browser form:
     echo <<<END
<html>
<head></head>

<body>
    <h2>SMS Messenger</h2>
         <form name="txter" lookupMethod="post" action="sms.php">

      Phone number: <input type="text" name="phoneNumber"
        size='15' maxlength='15'><br>
        Message: <input type="text" name="message" size='30'
           maxlength='128'><br>
         Carrier:     $to<br>

          <select name="carrier">
                <option value="teleflip.com">Teleflip</option>
                <option value="tmomail.net">T-Mobile</option>
                <option value="messaging.sprintpcs.com">Sprint</option>
                <option value="txt.att.net">AT&T</option>
                <option value="vtext.com">Verizon</option>
                <option value="messaging.nextel.com">Nextel</option>
            </select>

        <input type="submit" name="Submit" value="submit">

      </form>
</body>
</html>

END;

?>
```

Asterisk

Asterisk is an open source private branch telephone exchange (PBX). It allows you to manage a phone exchange over an IP network. It can make and take phone calls, save messages, play prerecorded messages to people who call in, and offers most of the usual features a phone account does, like caller ID, call waiting, call blocking, and so forth. You can also make connections between the phone and the Internet. For example, you can allow users to control the output of a PHP script using their touchtone keypad. It's not a large step from there to having a mobile phone keypad controlling a physical object through a Lantronix device, XBee radio, or any of the other tools you've seen here. It runs on a Unix or Linux server. Asterisk isn't easy for the beginner, but it's manageable by anyone comfortable with server-side programming like the PHP you've seen here. For more details on Asterisk, see www.asterisk.org.

The mobile phone development landscape is changing rapidly at the moment, and by the time you're reading this, there are likely many more tools for developing on phones, and for using phones as a user interface to networks in nontraditional ways. If you've enjoyed making embedded systems talk to each other, then by all means, jump into mobile phone programming as well.
X

Other Microcontrollers

Though the examples in this book have all been done using Arduino and Wiring, there are many other microcontroller platforms that you can use to do the same work. This section is an introduction to a few others on the market, and what they're good for. Some of these are standalone controllers that you program yourself, as you've done with the controllers in this book. Others are designed to be connected to a personal computer at all times. You don't have to program these, you just configure them via serial or Ethernet, then read from their inputs and write to their outputs from your desktop-based development environment.

Basic Stamp

Parallax (www.parallax.com) Basic Stamp and Basic Stamp 2 (BS-2) are probably the most common microcontrollers in the hobbyist market. Programmed in PBASIC, they are easy to use, and include the same basic functions as Wiring and Arduino: digital in and out, serial in and out, PWM out, and a form of analog in. Their analog in is slower than an analog-to-digital converter, however. In addition, PBASIC lacks the ability to pass parameters to functions, which makes programming many of the examples shown in this book more difficult. It's possible to do everything you've seen here on a Basic Stamp, however. And there are more code samples available on the Net for the BS-2 than for just about any other controller.

BX-24

NetMedia's BX-24 controller (www.basicx.com) affords everything that Wiring and Arduino do; in fact, it's even based on the same microcontroller family that those two are (the Atmel AVR controllers). It's programmed in a variant of Visual BASIC (BasicX), and even includes limited support for multitasking. The programming environment for it is available only on Windows, however. Of the various Basic Stamp–like modules on the market, it's arguably the best, especially for tasks like the ones found in this book. It's a decent alternative for networked objects for beginners.

The BX-24 and the Basic Stamp both cost around $50 apiece, and by the time you've bought the starter kit, around $100.

PIC and AVR

Microchip's PIC and Atmel's AVR microcontrollers are excellent microcontrollers. You'll find the AVRs at the heart of Arduino, Wiring, and BX-24 controllers, and the PICs at the heart of the Basic Stamps. The BX-24, Basic Stamp, Wiring, and Arduino environments are essentially wrappers around these controllers, making them easier to work with. To use PICs or AVRs on their own, you need a hardware programmer that connects to your computer, and you need to install a command-line compiler. There are BASIC and C compilers available for both microcontroller families. The commercial PIC C and BASIC compilers tend to be far more thoroughly developed and supported than any open source compilers for that family. In contrast, avr-gcc, the open source compiler for the AVR controllers, is an excellent tool, and has been expanded enthusiastically by the community using it. In fact, it's the engine that runs Wiring and Arduino.

If you're looking to expand to other compilers, or you want to learn about the technical details underlying what you've seen here, www.atmel.com and www.avrfreaks.net are the best places to start. In addition, Pascal Stang's AVRlib libraries for the AVR controllers offer many useful functions: hubbard.engr.scu.edu/avr/avrlib/. If you want to learn more about the PIC, start at the source: www.microchip.com. If you want a good BASIC compiler for the PIC, check out MicroEngineering Labs PicBasic Pro at www.melabs.com, and if you want a good commercial C compiler for it, try CCS C: www.ccsinfo.com.

Though the microcontrollers themselves are cheap (between $1 and $10 apiece), getting all the tools set up for yourself will cost you some money. It's generally cheaper on the AVR side, as the avr-gcc is a good free compiler, and the hardware programmers for the AVR can be gotten for less than $100. On the PIC side, you could spend a few hundred dollars by the time you get a good programmer and compiler. There's also a pretty significant time investment in getting set up, as the tools for programming these controllers from scratch assume more knowledge than any of the others listed here.

Make Controller

The MAKE Controller from *Make Magazine*, made by Making Things (www.makingthings.com) is a powerful controller. It's got built-in Ethernet, high-current drivers for motors on its outputs, and support for multitasking and communication via OSC. At $150, it's not the least expensive controller here, and at 3.5" x 4.5", it's not the smallest, but it is highly capable. With built-in Ethernet, it can replace the Arduino plus Lantronix combination handily, if your project has room for its footprint.

There are a few different ways to work with the Make module. If you're an experienced programmer, you can set up the compiler and development environment on your own machine and program it in C. If you're not, you can interface to it from other environments by sending it OSC commands, either via USB or via Ethernet. In addition, there are some built-in functions called Poly functions that allow you to build basic applications. Expect to see many exciting new developments in terms of its interface in the near future as well.

Propeller

The Parallax Propeller controller (www.parallax.com/propeller) is similar to the Make controller, in that it's a more powerful controller that affords multitasking. The programming environment for it is not for the beginner, and it's only available for Windows. The Propeller is capable of generating video, handling input from keyboard and mouse, and other tasks simultaneously. In fact, Uncommon Projects have made the YBox (ybox.tv), a device that overlays text from a website on a TV signal using a Propeller and an XPort. Though there are not yet generic examples using the Propeller to communicate over Ethernet, there undoubtedly will be soon.

As advanced processors continue to fall in price, you can expect to see more modules like the Make controller and the Propeller on the market for hobbyists and beginners.

Phidgets

Phidgets (www.phidgets.com) is a set of sensor and actuator modules that connect to your computer via USB. You don't program them, you just read their inputs and outputs from your desktop environment using Flash, Max/MSP, Java, or other tools. The NADA proxy tool mentioned earlier is designed to interface with Phidgets, among other things. Among the Phidgets modules are a number of useful sensors and actuators that can be used with other microcontroller and desktop environments. Their parts are well-designed and their connectors are solid, so if you know your project is going to take some abuse, you may want to spend a little more and buy at least your sensors from them.

SitePlayer

NetMedia's SitePlayer is an alternative to the Lantronix devices. It's basically a web server with telnet on a chip. Its form factor isn't quite as compact as the Lantronix units, and the tools for downloading new configurations for it are available only for Windows. Its web interface is handy, though, if you're looking to build a project in which users control a physical device through a web interface, because the interface can live directly on the chip, with no need for an external server.

X

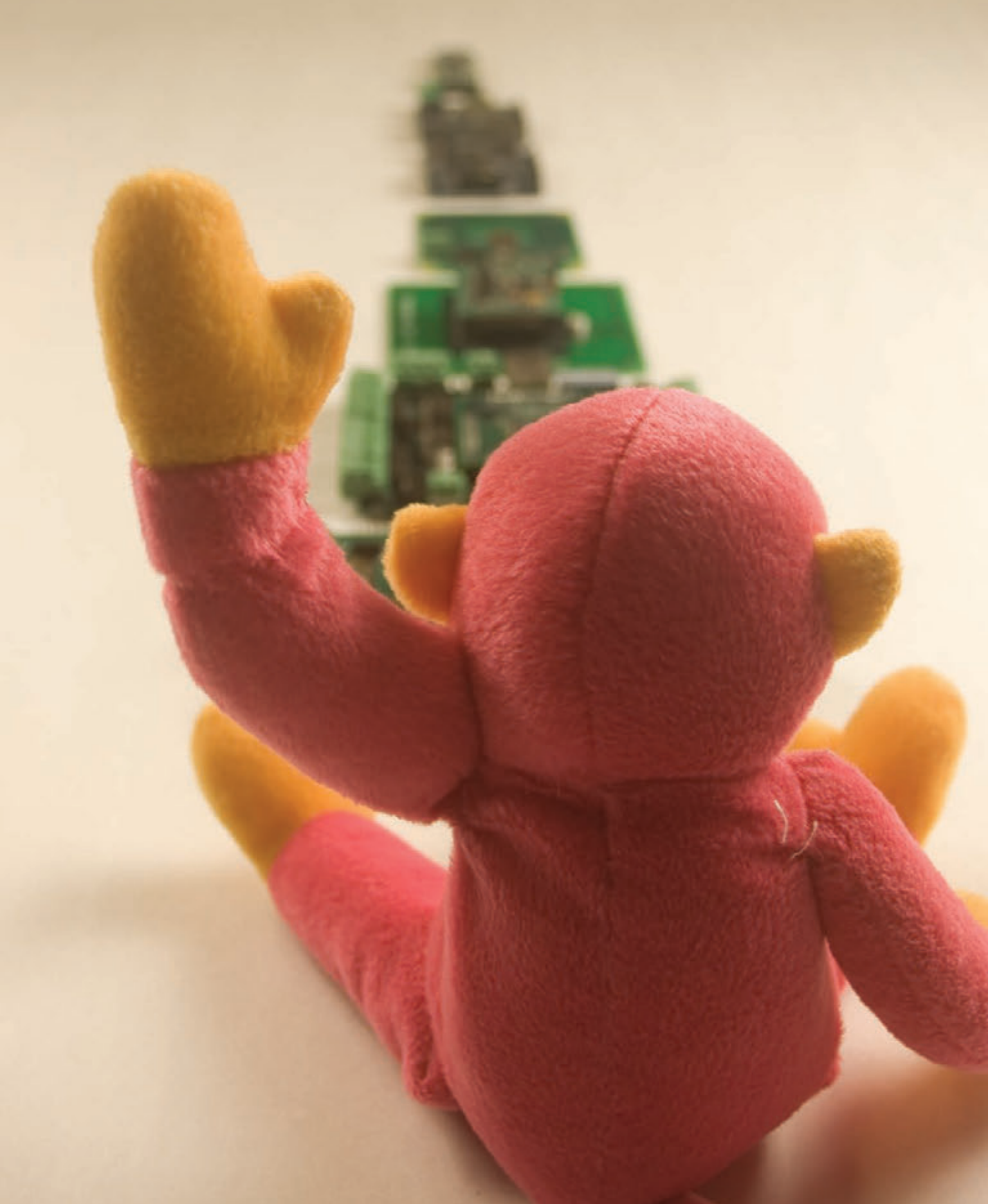
New Tools

There is no such thing as the technology book that's fully up-to-date, and this book is no exception. New tools are coming out for networking physical devices every day. Here are a few of the ones that excited me most that came out as I was finishing this book:

- Lantronix announced two new products: the XPort Direct, a lower cost version of the XPort; and the MatchPort, a lower-cost and smaller version of the WiPort. With these two, both serial-to-Ethernet and serial-to-WiFi connections get ever cheaper.
- The Arduino Bluetooth board is now available, allowing both programming and communication wirelessly from an Arduino board. An Arduino Ethernet board is in the works as well. In addition to Arduino modules made by the original developers, there are many Arduino derivatives coming on the market from the community as well.
- MaxStream announced the next generation of the XBee radios, which will include real ZigBee mesh networking, beyond what the examples in this book have demonstrated.
- IOgear, Netgear, Actiontec, and many others have released Powerline Ethernet modules. These devices enable you to send Ethernet signals over AC powerlines at speeds up to 200 Mbps, several orders of magnitude faster than X10. Powerline Ethernet control modules won't be far behind this.

More tools will follow these. Though some specifics of the examples you've read may be out of date in a few years, the principles will give you the foundations to learn not only the tools you've seen here, but also new ones as they come along.

X




3-3V Zener Diode
TIP120 Transistors
2N2222 Transistors
green LEDs
red LEDs
yellow LEDs
DC power connectors
DSUB9 female connectors for serial
Jumpers
straight headers
right angle headers

Appendix B 

MAKE: PROJECTS

Where to Get Stuff

Many different hardware suppliers and software sources are mentioned in this book. This appendix provides a summary of all of them, along with a brief description of each. It's organized in two sections: hardware and software, sorted alphabetically by source.

Hardware

KEY
- Phone / * Toll Free
- Fax
- Mailing address

Abacom Technologies

Abacom sells a range of RF transmitters, receivers, and transceivers and serial-to-Ethernet modules.
www.abacomdirect.com
email: info@abacomdirect.com
383 Bering Avenue
Toronto, ON M8Z 3B1, Canada

Aboyd Company

The Aboyd Company sells art supplies, costumes, novelties, cardboard standups, home décor, and more. They're also a good source of Charley Chimp cymbal-playing monkeys.
www.aboyd.com
email: info@aboyd.com
+1-888-458-2693
+1-601-948-3477 *International*
+1-601-948-3479
P.O. Box 4568
Jackson, MS 39296, USA

Acroname Easier Robotics

Acroname sells a wide variety of sensors and actuators for robotics and electronics projects. They've got an excellent range of esoteric sensors like UV flame sensors, cameras, and thermal array sensors. They've got a lot of basic distance rangers as well. They also have a number of good tutorials on how to use their parts on their site.
www.acroname.com
email: info@acroname.com
+1-720-564-0373
+1-720-564-0376
Acroname Inc.
4822 Sterling Dr.
Boulder, CO 80301-2350, USA

Adafruit Industries

Adafruit makes a number of useful open source DIY electronics kits, including an AVR programmer, an MP3 player, and more.
www.adafruit.com
email: sales@adafruit.com

Atmel

Atmel makes the AVR microcontrollers that are at the heart of the Arduino, Wiring, and BX-24 modules. They also make the ARM microcontroller that runs the Make controller.
www.atmel.com
+1-408-441-0311
2325 Orchard Parkway
San Jose, CA 95131, USA

Blue Radios

Blue Radios makes and sells Bluetooth radio modules for electronics manufacturers. Their radios are at the heart of SparkFun's BlueSMiRF dongles.
www.blueradios.com
email: sales@blueradios.com
+1-303-957-1003
+1-303-845-7134
7173 S. Havana Street, Suite 600
Englewood, CO 80112, USA

Devantech/Robot Electronics

Devantech makes ultrasonic ranger sensors, electronic compasses, LCD displays, motor drivers, relay controllers, and other useful add-ons for microcontroller projects.
robot-electronics.co.uk
e-mail: sales@robot-electronics.co.uk
+44 (0)1379 640450 or 644285
+44 (0)1379 650482
Unit 2B Gilray Road
Diss, Norfolk, IP22 4EU, England

Digi-Key Electronics

Digi-Key is one of the U.S.'s largest retailers of electronics components. They're a staple source for things you use all the time — resistors, capacitors, connectors, some sensors, breadboards, wire, solder, and more.
www.digikey.com
+1-800-344-4539 or
+1-218-681-6674
+1-218-681-3380
701 Brooks Avenue South
Thief River Falls, MN 56701, USA

ELFA

ELFA is one of Northern Europe's largest electronics components suppliers.
www.elfa.se
email: export@elfa.se
+46 8 580 941 30
S-175 80 Järfälla, Sweden

Farnell

Farnell supplies electronics components for all of Europe. Their catalog part numbers are consistent with Newark in the U.S., so if you're working on both sides of the Atlantic, sourcing Farnell parts can be convenient.
www.farnell.co.uk
email: sales@farnell.co.uk
+44-8701-200-200
+44-8701-200-201
Canal Road,
Leeds, LS12 2TU, United Kingdom

Figaro USA, Inc.

Figaro Sensor sells a range of gas sensors, including volatile organic compound sensors, carbon monoxide sensors, oxygen sensors, and more.
www.figarosensor.com
email: figarousa@figarosensor.com
+1-847-832-1701
+1-847-832-1705
3703 West Lake Ave., Suite 203
Glenview, IL 60026, USA

Future Technology Devices International, Ltd. +(FTDI)

FTDI makes a range of USB-to-serial adaptor chips, including the FT232RL that's on many of the modules in this book.

www.ftdichip.com
email: admin1@ftdichip.com
+44 (0) 141 429 2777
373 Scotland Street
Glasgow, G5 8QB, United Kingdom

Glolab

Glolab makes a range of electronic kits and modules including several useful RF and IR transmitters, receivers, and transceivers.

www.glolab.com
email: lab@glolab.com

Gridconnect

Gridconnect distributes networking products, including those from Lantronix and Maxstream.

www.gridconnect.com
email: sales@gridconnect.com
+1 630 245 1445
+1 630 245 1717
+1 800 975 GRID (4743) *U.S. toll-free*
1630 W. Diehl Road
Naperville, IL 60563, USA

Images SI Inc.

Images SI sells robotics and electronics parts. They carry a range of RFID parts, force-sensing resistors, stretch sensors, gas sensors, electronic kits, speech recognition kits, solar energy parts, and microcontrollers.

www.imagesco.com
email: imagesco@verizon.net
+1-718-966-3694
+1-718-966-3695
109 Woods of Arden Road
Staten Island, NY 10312, USA

Interlink Electronics

Interlink makes force-sensing resistors, touchpads, and other input devices.

www.interlinkelectronics.com
email: specialty@interlinkelectronics.com
+1-805-484-8855
+1-805-484-8989
546 Flynn Road
Camarillo, CA 93012, USA

IOGear

IOGear make computer adaptors. Their USB-to-serial adaptors are good, and they carry Powerline Ethernet products.

www.iogear.com
email: sales@iogear.com
+1-866-946-4327 *Toll-free*
+1-949-453-8782
+1-949-453-8785
23 Hubble Drive
Irvine, CA 92618, USA

Jameco Electronics

Jameco carries bulk and individual electronics components, cables, breadboards, tools, and other staples for the electronics hobbyist or professional.

www.jameco.com
1355 Shoreway Road
Belmont, CA 94002, USA
email:
domestic@jameco.com
international@jameco.com
custservice@jameco.com
+1-800-831-4242
Toll-free 24-hour order line
+1-650-592-8097
International order line
+1-650-592-2503 *International*
+1-800-237-6948* *Toll-free fax*
+001-800-593-1449*
Mexico toll-free fax
+1-803-015-237-6948*
Indonesia toll-free fax

Keyspan

Keyspan makes computer adaptors. Their USA-19*xx* series of USB-to-serial adaptors are very handy for microcontroller work.

www.keyspan.com
email: info@keyspan.com
+1-510-222-0131 *Info/sales*
+1-510-222-8802 *Support*
+1-510-222-0323
4118 Lakeside Dr
Richmond, CA 94806, USA

Lantronix

Lantronix makes the serial-to-Ethernet modules used in this book: the XPort, the WiPort, the WiMicro, the Micro, and many others.

www.lantronix.com
email: sales@lantronix.com
+1-800-526-8766
+1-949-453-3990
+1-949-450-7249
5353 Barranca Parkway
Irvine, CA 92618, USA

Libelium

Libelium makes an XBee shield for Arduino and other wireless products.

www.libelium.com
email: info@libelium.com
Libelium Comunicaciones Distribuidas S.L.
Maria de Luna 11, Instalaciones
CEEIARAGON, C.P: 50018
Zaragoza, Spain

Linx Technologies

Linx makes a number of RF receivers, transmitters, and transceivers.

www.linxtechnologies.com
email: info@linxtechnologies.com
+1-800-736-6677 *U.S.*
+1-541-471-6256 *International*
+1-541-471-6251
159 Ort Lane
Merlin, OR 97532, USA

Low Power Radio Solutions

LPRS makes a number of RF receivers, transmitters, and transceivers.

www.lprs.co.uk
email: info@lprs.co.uk
+44-1993-709418
+44-1993-708575
Two Rivers Industrial Estate
Station Lane, Witney
Oxon, OX28 4BH, United Kingdom

Making Things

Making Things makes the MAKE controller, and originated the now-discontinued Teleo controllers. They do custom hardware engineering solutions.

www.makingthings.com
email: info@makingthings.com
+1-415-255-9513
1020 Mariposa Street, #2
San Francisco, CA 94110, USA

Mannings RFID Shop

Mannings sells RFID tools and bar code readers and printers. They sell the ID Innovations RFID readers.

www.rfidshop.com
email: info@manningsrfid.com
+44-1704-538-202
+44-1704-514-713
Units 1–5, Russell Road
Southport, Merseyside, England
PR9 7SY, United Kingdom

Maxim Integrated Products

Maxim makes sensors, communications chips, power management chips, and more. They also own Dallas Semiconductor. Together, they're one of the major sources for chips related to serial communication, temperature sensors, LCD control, and much more.

www.maxim-ic.com
email: info2@maxim-ic.com
+1-408-737-7600
+1-408-737-7194
120 San Gabriel Drive
Sunnyvale, CA 94086, USA

Maxstream

Maxstream makes ZigBee radios, radio modems, and Ethernet bridges.

www.maxstream.net
+1-866-765-9885
+1-801-765-9885 *International*
+1-801-765-9895
355 South 520 West Suite 180
Lindon, UT 84042, USA

Microchip

Microchip makes the PIC family of microcontrollers. They have a very wide range of microcontrollers, for just about every conceivable purpose.

www.microchip.com
+1-480-792-7200
2355 West Chandler Blvd.
Chandler, AZ, 85224-6199, USA

Mouser

Mouser is a large retailer of electronic components in the U.S. They stock most of the staple parts used in the projects in this book, like resistors, capacitors, and some sensors. They also carry the FTDI USB-to-serial cable.

www.mouser.com
email: help@mouser.com
1000 North Main Street
Mansfield, TX 76063, USA

NetMedia

NetMedia makes the BX-24 microcontroller module and the SitePlayer Ethernet module.

www.basicx.com
siteplayer.com
email: sales@netmedia.com
+1-520-544-4567
+1-520-544-0800
10940 N. Stallard Place
Tucson, AZ 85737, USA

Newark In One Electronics

Newark supplies electronics components in the U.S. Their catalog part numbers are consistent with Farnell in the Europe, so if you're working on both sides of the Atlantic, sourcing parts from Farnell and Newark can be convenient.

www.newark.com
email: somewhere@something.com
+1-773-784-5100
+1-888-551-4801
4801 N. Ravenswood
Chicago, IL 60640-4496, USA

New Micros

New Micros sells a number of microcontroller modules. They also sell a USB-XBee dongle that allows you to connect Maxstream's XBee radios to a computer really easily. Their dongles also have all the necessary pins connected for reflashing the XBee's firmware serially.

www.newmicros.com
email: nmisales@newmicros.com
+1-214-339-2204

Parallax

Parallax makes the Basic Stamp family of microcontrollers. They also make the Propeller microcontroller, and a wide range of sensors, beginners' kits, robots, and other useful tools for people interested in electronics and microcontroller projects.

www.parallax.com
email: sales@parallax.com
+1-888-512-1024 *Toll-free sales*
+1-916-624-8333
Office/international
+1-916-624-8003
599 Menlo Drive
Rocklin, California 95765, USA

Phidgets

Phidgets makes input and output modules that connect desktop and laptop computers to the physical world.

www.phidgets.com
email: sales@phidgets.com
+1-403-282-7335
+1402-282-7332
2715A 16A NW
Calgary, Alberta T2M3R7, Canada

RadioShack

You've got questions, they've got a cell phone plan for you. Despite their increasing focus on mobile phone plans, they still do carry some useful parts. Check the website for part numbers and call your local store first to see if they've got what you need. It'll save you time.

www.radioshack.com

Reynolds Electronics

Reynolds Electronics makes a number of small kits and modules for RF and infrared communications, IR remote control, and other useful add-on functions for microcontroller projects.

www.rentron.com
email: sales@rentron.com
+1-772-589-8510
+1-772-589-8620
12300 Highway A1A
Vero Beach, Florida, 32963, USA

Samtec

Samtec makes electronic connectors. They have a very wide range of connectors, so if you're looking for something odd, they probably make it.

www.samtec.com
email: info@samtec.com
+1-800-SAMTEC-9

Skyetek

Skyetek makes RFID readers, writers, and antennas.

www.skyetek.com
+1-720-565-0441
+1-720-565-8989
11030 Circle Point Road, Suite 300
Westminster, CO 80020, USA

Smarthome

Smarthome makes a wide variety of home automation devices, including cameras, appliance controllers, X10, and INSTEON devices.

www.smarthome.com
email: custsvc@smarthome.com
1-800-762-7846
+ 1-800-871-5719 *Canada*
+1-949-221-9200 *International*
16542 Millikan Avenue
Irvine, CA 92606, USA

Smart Projects/PCB Europe

Smart Projects/PCB Europe makes Arduino modules and shields, kits for building your own modules. They also make custom electronics projects.

www.pcb-europe.net/catalog
email: info@pcb-europe.net
+39-339-296-5590
Via Siccardi, 12
10034 Chivasso TO, Italy

SparkFun Electronics

SparkFun makes it easier to use all kinds of electronic components. They make breakout boards for sensors, radios, power regulators, and sell a variety of microcontroller platforms.

www.sparkfun.com
email: spark@sparkfun.com
2500 Central Avnue, Suite Q
Boulder, CO 80301, USA

Symmetry Electronics

Symmetry sells ZigBee and Bluetooth radios, serial-to-Ethernet modules, wi-fi modules, cellular modems, and other electronic communications devices.

www.semiconductorstore.com
+1-877-466-9722
+1-310-643-3470 *International*
+1-310-297-9719
5400 West Rosecrans Avenue
Hawthorne, CA 90250, USA

TI-RFID

TIRIS is Texas Instruments' RFID division. They make tags and readers for RFID in many bandwidths and protocols.

www.tiris.com
+1-800-962-RFID (7343)
+1-214-567-RFID (7343)
Radio Frequency Identification Systems
6550 Chase Oaks Blvd., MS 8470
Plano, TX 75023, USA

Trossen Robotics

Trossen Robotics sells a range of RFID supplies and robotics. They have a number of good sensors, including Interlink force-sensing resistors, linear actuators, Phidgets kits, RFID readers, and tags for most RFID ranges.

www.trossenrobotics.com
email: jenniej@trossenrobotics.com
+1-877-898-1005
+1-708-531-1614
1 Westbrook Co. Center, Suite 910
Westchester, IL 60154, USA

Uncommon Projects

Uncommon Projects make the YBox, a text overlay device that puts text from web feeds on your TV.

www.uncommonprojects.com
ybox.tv
email: info@uncommonprojects.com
68 Jay Street #206
Brooklyn New York 11201, USA

Software

Most of the software listed in this book is open source. In the following listings, anything that's not open source is noted explicitly as a commercial application. If there's no note, you can assume it's open.

Arduino
Arduino is a programming environment for AVR microcontrollers. It's based on Processing's programming interface. It runs on Mac OS X, Linux, and Windows operating systems.
www.arduino.cc

Asterisk
Asterisk is a software private branch exchange (PBX) manager for telephony. It runs on Linux and Unix operating systems.
www.asterisk.org

AVRlib
AVRlib is a library of C functions for a variety of tasks using AVR processors. It runs on Mac OS X, Linux, and Windows operating systems as a library for the avr-gcc compiler.
hubbard.engr.scu.edu/avr/avrlib/

avr-gcc
The GNU avr-gcc is a C compiler and assembler for AVR microcontrollers. It runs on Mac OS X, Linux, and Windows operating systems.
www.avrfreaks.net/AVRGCC/

CCS C
CCS C is a commercial C compiler for the PIC microcontroller. It runs on Windows and Linux operating systems.
www.ccsinfo.com

Dave's Telnet
Dave's Telnet is a telnet application for Windows.
dtelnet.sourceforge.net

Eclipse
Eclipse is an integrated development environment (IDE) for programming in many different languages. It's extensible through a plugin architecture, and there are compiler links to most major programming languages. It runs on Mac OS X, Linux, and Windows.
www.eclipse.org

Evocam
Evocam is a commercial webcam application for Mac OS X.
evological.com

Exemplar
Exemplar is a tool for authoring sensor applications through behavior rather than through programming. It runs on Mac OS X, Linux, and Windows operating systems as a plugin for Eclipse.
hci.stanford.edu/research/exemplar/

Fwink
Fwink is a webcam application for Windows.
lundie.ca/fwink

Girder
Girder is a commercial home automation application for Windows.
www.girder.nl

Java
Java is a programming language. It runs on Mac OS X, Linux, and Windows operating systems, and many embedded systems as well.
java.sun.com

Macam
Macam is a webcam driver for Mac OS X.
https://sourceforge.net/projects/webcam-osx/

Max/MSP
Max is a commercial graphic data flow authoring tool. It allows you to program by connecting graphic objects rather than writing text. Connected with Max are MSP, a realtime audio signal processing library, and Jitter, a realtime video signal processing library. It runs on Mac OS X, Linux, and Windows operating systems.
www.cycling74.com

Puredata (PD)
Puredata (PD) is a graphic data flow authoring tool. It allows you to program by connecting graphic objects rather than writing text. It runs on Mac OS X, Linux, and Windows operating systems.
puredata.info

PEAR
PEAR is the PHP Extension and Application Repository. It hosts extension libraries for the PHP scripting language, including NET_POP3 (pear.php.net/package/Net_POP3), which is used in this book.
pear.php.net

PHP
PHP is a scripting language that is especially suited for web development and can be embedded into HTML. It runs on Mac OS X, Linux, and Windows operating systems.
www.php.net

PicBasic Pro

PicBasic Pro is a commercial BASIC compiler for PIC microcontrollers. It runs on Windows.
www.melabs.com

Processing

Processing is a programming language and environment designed for the non-technical user who wants to program images, animation, and interaction. It runs on Mac OS X, Linux, and Windows.
www.processing.org

Proxi

Proxi is a free (but not open source) application for automating operating system tasks based on events. It runs on Mac OS X.
proxi.griffintechnology.com

Putty SSH

Putty is a telnet/SSH/serial port client for Windows.
www.puttyssh.org

QRcode Library

QRcode library is a set of libraries for encoding and decoding QRcode 2-D barcodes. It runs on Mac OS X, Linux, and Windows as a library for Java.
qrcode.sourceforge.jp

Dan Shiffman's Processing Libraries

Dan Shiffman has written a number of useful libraries for Processing, including the pqrcode library used in this book (www.shiffman.net/p5/pqrcode). He's also got a SFTP library (www.shiffman.net/2007/06/04/sftp-with-java-processing/) and a sudden-motion sensor library for Mac OS X (www.shiffman.net/2006/10/28/processingsms/).

Sketchtools NADA

NADA is a commercial proxy tool for connecting programming environments with hardware devices.
www.sketchtools.com

TinkerProxy

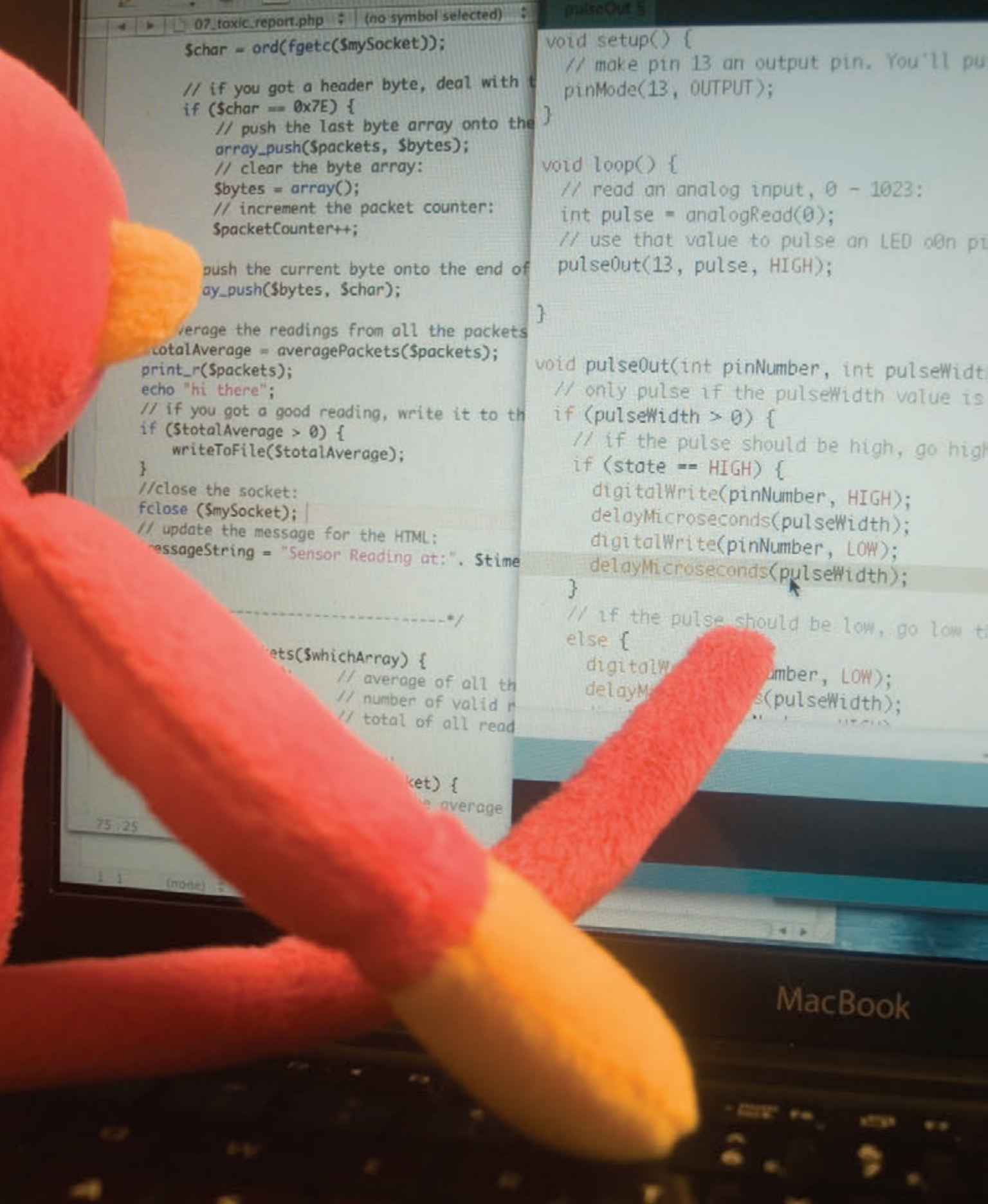
TinkerProxy is a TCP-to-serial proxy application. It runs on Windows.
tinker.it/now/category/software/

UDP Library for Processing

Hypermedia's UDP library for Processing enables you to communicate via UDP from Processing. It runs on Mac OS X, Linux, and Windows as a library for Processing.
hypermedia.loeil.org/processing/

Wiring

Wiring is a programming environment for AVR microcontrollers. It's based on Processing's programming interface. It runs on Mac OS X, Linux, and Windows operating systems.
www.wiring.org.co


07_toxic_report.php
(no symbol selected)
$char = ord(fgetc($mySocket));
// if you got a header byte, deal with t
if ($char == 0x7E) {
// push the last byte array onto the
array_push($packets, $bytes);
// clear the byte array:
$bytes = array();
// increment the packet counter:
$packetCounter++;
push the current byte onto the end of
ay_push($bytes, $char);
verage the readings from all the packets
totalAverage = averagePackets($packets);
print_r($packets);
echo "hi there";
// if you got a good reading, write it to th
if ($totalAverage > 0) {
writeToFile($totalAverage);
}
//close the socket:
fclose ($mySocket);
// update the message for the HTML:
essageString = "Sensor Reading at:". $time
*/
ets($whichArray) {
// average of all th
// number of valid r
// total of all read
ket) {
average
pulseOut 5
void setup() {
// make pin 13 an output pin. You'll pu
pinMode(13, OUTPUT);
}
void loop() {
// read an analog input, 0 - 1023:
int pulse = analogRead(0);
// use that value to pulse an LED o0n pi
pulseOut(13, pulse, HIGH);
}
void pulseOut(int pinNumber, int pulseWidt
// only pulse if the pulseWidth value is
if (pulseWidth > 0) {
// if the pulse should be high, go high
if (state == HIGH) {
digitalWrite(pinNumber, HIGH);
delayMicroseconds(pulseWidth);
digitalWrite(pinNumber, LOW);
delayMicroseconds(pulseWidth);
}
// if the pulse should be low, go low t
else {
digitalW umber, LOW);
delayM s(pulseWidth);
MacBook

Appendix C

MAKE: PROJECTS

Program Listings

This Appendix contains the complete code listings for all of the programs in the book. You can find a link to download the source code at www.makezine.com/go/MakingThingsTalk.

Chapter 1

Hello World!

Language: Processing

Prints out "Hello World!"

```
println("Hello World!\n");
```

Triangle drawing program

Language: Processing

Draws a triangle whenever the mouse button is not pressed. Erases when the mouse button is pressed.

```
// declare your variables:
float redValue = 0;    // variable to hold the red color
float greenValue = 0;  // variable to hold the green color
float blueValue = 0;   // variable to hold the blue color

// the setup() method runs once at the beginning of the program:

void setup() {
  size(320, 240);    // sets the size of the applet window
  background(0);     // sets the background of the window to black
  fill(0);           // sets the color to fill shapes with (0 = black)
  smooth();          // draw with antialiased edges
}

// the draw() method runs repeatedly, as long as the applet window
// is open.  It refreshes the window, and anything else you program
// it to do:

void draw() {

  // Pick random colors for red, green, and blue:
  redValue = random(255);
  greenValue = random(255);
  blueValue = random(255);

  // set the line color:
  stroke(redValue, greenValue, blueValue);

  // draw when the mouse is up (to hell with conventions):
  if (mousePressed == false) {
    // draw a triangle:
    triangle(mouseX, mouseY, width/2, height/2,pmouseX, pmouseY);
  }
  // erase when the mouse is down:
  else {
    background(0);
    fill(0);
  }
}
```

Hello World!

Language: PHP

Prints out a "Hello World!" HTML page

```
<?php
echo "<html><head></head><body>\n";
echo "hello world!\n";
echo "</body></html>\n";
?>
```

Date printer

Language: PHP

Prints the date and time in an HTML page.

```
<?php
//    Get the date, and format it:
$date = date("Y-m-d h:i:s\t");

// print the beginning of an HTML page:
echo "<html><head></head><body>\n";
echo "hello world!<br>\n";
// Include the date:
echo "Today's date: $date<br>\n";
// finish the HTML:
echo "</body></html>\n";

?>
```

Blink (aka Hello World! In Arduino/Wiring):

Language: Arduino/Wiring

Blinks an LED attached to pin 13 every half second.

Connections: Pin 13: + leg of an LED (- leg goes to ground)

```
int LEDPin = 13;

void setup() {
  pinMode(LEDPin, OUTPUT);    // set pin 13 to be an output
}

void loop() {
  digitalWrite(LEDPin, HIGH);       // turn the LED on pin 13 on
  delay(500);                       // wait half a second
  digitalWrite(LEDPin, LOW);        // turn the LED off
  delay(500);                       // wait half a second
}
```

Simple Serial

Language: Arduino/Wiring

Listens for an incoming serial byte, adds one to the byte and sends the result back out serially. Also blinks an LED on pin 13 every half second.

```
int LEDPin = 13;              // you can use any digital I/O pin you want
int inByte = 0;               // variable to hold incoming serial data
long blinkTimer = 0;          // keeps track of how long since the LED
                              // was last turned off
int blinkInterval = 1000;     // a full second from on to off to on again

void setup() {
  pinMode(LEDPin, OUTPUT);    // set pin 13 to be an output
  Serial.begin(9600);         // configure the serial port for 9600 bps
                              // data rate.
}

void loop() {
  // if there are any incoming serial bytes available to read:
  if (Serial.available() > 0) {
    // then read the first available byte:
    inByte = Serial.read();
    // and add one to it, then send the result out:
    Serial.print(inByte+1, BYTE);
  }

  // Meanwhile, keep blinking the LED.
  // after a quarter of a second, turn the LED on:
  if (millis() - blinkTimer >= blinkInterval / 2) {
    digitalWrite(LEDPin, HIGH);      // turn the LED on pin 13 on
  }
  // after a half a second, turn the LED off and reset the timer:
  if (millis() - blinkTimer >= blinkInterval) {
    digitalWrite(LEDPin, LOW);        // turn the LED off
    blinkTimer = millis();            // reset the timer
  }
}
```

Chapter 2

Sensor Reader

Language: Wiring/Arduino

Reads two analog inputs and two digital inputs and outputs their values.

Connections:

- analog sensors on analog input pins 0 and 1
- switches on digital I/O pins 2 and 3

```
int leftSensor = 0;    // analog input for the left arm
int rightSensor = 1;   // analog input for the right arm
int resetButton = 2;   // digital input for the reset button
int serveButton = 3;   // digital input for the serve button

int leftValue = 0;     // reading from the left arm
int rightValue = 0;    // reading from the right arm
int reset = 0;         // reading from the reset button
int serve = 0;         // reading from the serve button

void setup() {
  // configure the serial connection:
  Serial.begin(9600);
  // configure the digital inputs:
  pinMode(resetButton, INPUT);
  pinMode(serveButton, INPUT);
}

void loop() {
  // read the analog sensors:
  leftValue = analogRead(leftSensor);
  rightValue = analogRead(rightSensor);

  // read the digital sensors:
  reset = digitalRead(resetButton);
  serve = digitalRead(serveButton);

  // print the results:
  Serial.print(leftValue, DEC);
  Serial.print(",");
  Serial.print(rightValue, DEC);
  Serial.print(",");
  Serial.print(reset, DEC);
  Serial.print(",");
  // print the last sensor value with a println() so that
  // each set of four readings prints on a line by itself:
  Serial.println(serve, DEC);
}
```

Serial String Reader

Language: Processing

Reads in a string of characters from a serial port until it gets a linefeed (ASCII 10). Then splits the string into sections separated by commas. Then converts the sections to ints, and prints them out.

```
import processing.serial.*;     // import the Processing serial library

int linefeed = 10;              // Linefeed in ASCII
Serial myPort;                  // The serial port

void setup() {
  // List all the available serial ports
  println(Serial.list());

  // I know that the first port in the serial list on my mac
  // is always my  Arduino module, so I open Serial.list()[0].
  // Change the 0 to the appropriate number of the serial port
  // that your microcontroller is attached to.
  myPort = new Serial(this, Serial.list()[0], 9600);

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil(linefeed);
}

void draw() {
  // twiddle your thumbs
}

// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():

void serialEvent(Serial myPort) {
  // read the serial buffer:
  String myString = myPort.readStringUntil(linefeed);

  // if you got any bytes other than the linefeed:
  if (myString != null) {
    myString = trim(myString);

    // split the string at the commas
    // and convert the sections into integers:
    int sensors[] = int(split(myString, ','));

    // print out the values you got:
    for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++) {
      print("Sensor " + sensorNum + ": " + sensors[sensorNum] + "\t");
    }
    // add a linefeed after all the sensor values are printed:
    println();
  }
}
```

Monski Pong

Language: Processing

Uses the values from four sensors to animate a game of pong. Expects a serial string from the serial port in the following format:

leftPaddle, rightPaddle, resetButton, serveButton, linefeed

- leftPaddle: ASCII numeric string from 0 - 1023
- rightPaddle: ASCII numeric string from 0 - 1023
- resetButton: ASCII numeric string from 0 - 1
- serveButton: ASCII numeric string from 0 - 1

```
import processing.serial.*;      // import the serial library

int linefeed = 10;                  // Linefeed in ASCII
Serial myPort;                      // The serial port

float leftPaddle, rightPaddle;   // variables for the flex sensor values
int resetButton, serveButton;    // variables for the button values
int leftPaddleX, rightPaddleX;  // horizontal positions of the paddles
int paddleHeight = 50;           // vertical dimension of the paddles
int paddleWidth = 10;            // horizontal dimension of the paddles

float leftMinimum = 250;        // minimum value of the left flex sensor
float rightMinimum = 260;       // minimum value of the right flex sensor
float leftMaximum = 450;        // maximum value of the left flex sensor
float rightMaximum = 460;       // maximum value of the right flex sensor

int ballSize = 10;      // the size of the ball
int xDirection = 1;     // the ball's horizontal direction.
                        // left is -1, right is 1.
int yDirection = 1;     // the ball's vertical direction.
                        // up is -1, down is 1.
int xPos, yPos;  // the ball's horizontal and vertical positions

boolean ballInMotion = false; // whether the ball should be moving
int leftScore = 0;
int rightScore = 0;

PFont myFont;
int fontSize = 36;

void setup() {
  // set the window size:
  size(640, 480);

  // initialize the ball in the center of the screen:
  xPos = width/2;
  yPos = height/2;

  // List all the available serial ports
  println(Serial.list());

  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);
```

```
  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil(linefeed);

  // initialize the sensor values:
  leftPaddle = height/2;
  rightPaddle = height/2;
  resetButton = 0;
  serveButton = 0;

  // initialize the paddle horizontal positions:
  leftPaddleX = 50;
  rightPaddleX = width - 50;

  // set no borders on drawn shapes:
  noStroke();

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], fontSize);
  textFont(myFont);
}

void draw() {
  background(0);
  // draw the left paddle:
  rect(leftPaddleX, leftPaddle, paddleWidth, paddleHeight);
  // draw the right paddle:
  rect(rightPaddleX, rightPaddle, paddleWidth, paddleHeight);

  // calculate the ball's position and draw it:
  if (ballInMotion == true) {
    animateBall();
  }
  // if the serve button is pressed, start the ball moving:
  if (serveButton == 1) {
    ballInMotion = true;
  }
  // if the reset button is pressed, reset the scores
  // and start the ball moving:
  if (resetButton == 1) {
    leftScore = 0;
    rightScore = 0;
    ballInMotion = true;
  }
  // print the scores:
  text(leftScore, fontSize, fontSize);
  text(rightScore, width-fontSize, fontSize);
}

// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():

void serialEvent(Serial myPort) {
  // read the serial buffer:
  String myString = myPort.readStringUntil(linefeed);
  // if you got any bytes other than the linefeed:
  if (myString != null) {

    myString = trim(myString);
    // split the string at the commas
    //and convert the sections into integers:
    int sensors[] = int(split(myString, ','));
    // if you received all the sensor strings, use them:
    if (sensors.length == 4) {
      // calculate the flex sensors' ranges:
      float leftRange = leftMaximum - leftMinimum;
      float rightRange = rightMaximum - rightMinimum;

      // scale the flex sensors' results to the paddles' range:
      leftPaddle =  height * (sensors[0] - leftMinimum) / leftRange;
      rightPaddle = height * (sensors[1] - rightMinimum) / rightRange;

      // assign the switches' values to the button variables:
      resetButton = sensors[2];
      serveButton = sensors[3];

      // print the sensor values:
      print("left: "+ leftPaddle + "\tright: " + rightPaddle);
      println("\treset: "+ resetButton + "\tserve: " + serveButton);
    }
  }
}

void animateBall() {
  // if the ball is moving left:
  if (xDirection < 0) {
    //  if the ball is to the left of the left paddle:
    if  ((xPos <= leftPaddleX)) {
      // if the ball is in between the top and bottom
      // of the left paddle:
      if((leftPaddle - (paddleHeight/2) <= yPos) &&
        (yPos <= leftPaddle + (paddleHeight /2))) {
        // reverse the horizontal direction:
        xDirection =-xDirection;
      }
    }
  }
  // if the ball is moving right:
  else {
    //  if the ball is to the right of the right paddle:
    if  ((xPos >= ( rightPaddleX + ballSize/2))) {
      // if the ball is in between the top and bottom
      // of the right paddle:
      if((rightPaddle - (paddleHeight/2) <=yPos) &&
        (yPos <= rightPaddle + (paddleHeight /2))) {
        // reverse the horizontal direction:
        xDirection =-xDirection;
      }
    }
  }
```

```
  // if the ball goes off the screen left:
  if (xPos < 0) {
    rightScore++;
    resetBall();
  }
  // if the ball goes off the screen right:
  if (xPos > width) {
    leftScore++;
    resetBall();
  }

  // stop the ball going off the top or the bottom of the screen:
  if ((yPos - ballSize/2 <= 0) || (yPos +ballSize/2 >=height)) {
    // reverse the y direction of the ball:
    yDirection = -yDirection;
  }
  // update the ball position:
  xPos = xPos + xDirection;
  yPos = yPos + yDirection;

  // Draw the ball:
  rect(xPos, yPos, ballSize, ballSize);
}

void resetBall() {
  // put the ball back in the center
  xPos = width/2;
  yPos = height/2;
}
```

Monski Pong with Handshake

Language: Processing

Uses the values from four sensors to animate a game of pong. Expects a serial string from the serial port in the following format:

leftPaddle, rightPaddle, resetButton, serveButton, linefeed

- leftPaddle: ASCII numeric string from 0 - 1023
- rightPaddle: ASCII numeric string from 0 - 1023
- resetButton: ASCII numeric string from 0 - 1
- serveButton: ASCII numeric string from 0 - 1

Uses software handshaking by sending a carriage return for the microcontroller to respond to.

```
import processing.serial.*;      // import the serial library

int linefeed = 10;               // Linefeed in ASCII
Serial myPort;                   // The serial port
boolean madeContact = false;     // whether you've made initial contact
                                 // with the microcontroller
float leftPaddle, rightPaddle;   // variables for the flex sensor values
int resetButton, serveButton;    // variables for the button values
int leftPaddleX, rightPaddleX;   // horizontal positions of the paddles

int paddleHeight = 50;           // vertical dimension of the paddles
int paddleWidth = 10;            // horizontal dimension of the paddles

float leftMinimum = 250;         // minimum value of the left flex sensor
float rightMinimum = 260;        // minimum value of the right flex sensor
float leftMaximum = 450;         // maximum value of the left flex sensor
float rightMaximum = 460;        // maximum value of the right flex sensor

int ballSize = 10;               // the size of the ball
int xDirection = 1;              // the ball's horizontal direction.
// left is -1, right is 1.
int yDirection = 1;              // the ball's vertical direction.
// up is -1, down is 1.
int xPos, yPos;                  // the ball's horizontal and vertical positions

boolean ballInMotion = false;  // whether or not the ball should be moving
int leftScore = 0;
int rightScore = 0;

PFont myFont;
int fontSize = 36;

void setup() {
  // set the window size:
  size(640, 480);

  // initialize the ball in the center of the screen:
  xPos = width/2;
  yPos = height/2;

  // List all the available serial ports
  println(Serial.list());
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil(linefeed);

  // initialize the sensor values:
  leftPaddle = height/2;
  rightPaddle = height/2;
  resetButton = 0;
  serveButton = 0;

  // initialize the paddle horizontal positions:
  leftPaddleX = 50;
  rightPaddleX = width - 50;

  // set no borders on drawn shapes:
  noStroke();

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], fontSize);
  textFont(myFont);
}
```

```
void draw() {
  // if you've never gotten a string from the microcontroller,
  // keep sending carriage returns to prompt for one:
  if (madeContact == false) {
    myPort.write('\r');
  }
  background(0);
  // draw the left paddle:
  rect(leftPaddleX, leftPaddle, paddleWidth, paddleHeight);

  // draw the right paddle:
  rect(rightPaddleX, rightPaddle, paddleWidth, paddleHeight);

  // calculate the ball's position and draw it:
  if (ballInMotion == true) {
    animateBall();
  }
  // if the serve button is pressed, start the ball moving:
  if (serveButton == 1) {
    ballInMotion = true;
  }
  // if the reset button is pressed, reset the scores
  // and start the ball moving:
  if (resetButton == 1) {
    leftScore = 0;
    rightScore = 0;
    ballInMotion = true;
  }
  // print the scores:
  text(leftScore, fontSize, fontSize);
  text(rightScore, width-fontSize, fontSize);
}


// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():

void serialEvent(Serial myPort) {
  // if serialEvent occurs at all, contact with the microcontroller
  // has been made:
  madeContact = true;

  // read the serial buffer:
  String myString = myPort.readStringUntil(linefeed);

  // if you got any bytes other than the linefeed:
  if (myString != null) {

    myString = trim(myString);
    // split the string at the commas
    //and convert the sections into integers:
    int sensors[] = int(split(myString, ','));
    // if you received all the sensor strings, use them:
    if (sensors.length == 4) {
      // calculate the flex sensors' ranges:
      float leftRange = leftMaximum - leftMinimum;
      float rightRange = rightMaximum - rightMinimum;

      // scale the flex sensors' results to the paddles' range:
      leftPaddle =  height * (sensors[0] - leftMinimum) / leftRange;
      rightPaddle = height * (sensors[1] - rightMinimum) / rightRange;

      // assign the switches' values to the button variables:
      resetButton = sensors[2];
      serveButton = sensors[3];

      // print the sensor values:
      print("left: "+ leftPaddle + "\tright: " + rightPaddle);
      println("\treset: "+ resetButton + "\tserve: " + serveButton);

      // send out the serial port to ask for data:
      myPort.write('\r');
    }
  }
}
void animateBall() {
  // if the ball is moving left:
  if (xDirection < 0) {
    //  if the ball is to the left of the left paddle:
    if  ((xPos <= leftPaddleX)) {
      // if the ball is in between the top and bottom
      // of the left paddle:
      if((leftPaddle - (paddleHeight/2) <= yPos) &&
        (yPos <= leftPaddle + (paddleHeight /2))) {
        // reverse the horizontal direction:
        xDirection =-xDirection;
      }
    }
  }
  // if the ball is moving right:
  else {
    //  if the ball is to the right of the right paddle:
    if  ((xPos >= ( rightPaddleX + ballSize/2))) {
      // if the ball is in between the top and bottom
      // of the right paddle:
      if((rightPaddle - (paddleHeight/2) <=yPos) &&
        (yPos <= rightPaddle + (paddleHeight /2))) {

        // reverse the horizontal direction:
        xDirection =-xDirection;
      }
    }
  }

  // if the ball goes off the screen left:
  if (xPos < 0) {
    rightScore++;
    resetBall();
  }
```

```
  // if the ball goes off the screen right:
  if (xPos > width) {
    leftScore++;
    resetBall();
  }


  // stop the ball going off the top or the bottom of the screen:
  if ((yPos - ballSize/2 <= 0) || (yPos +ballSize/2 >=height)) {
    // reverse the y direction of the ball:
    yDirection = -yDirection;
  }
  // update the ball position:
  xPos = xPos + xDirection;
  yPos = yPos + yDirection;

  // Draw the ball:
  rect(xPos, yPos, ballSize, ballSize);
}

void resetBall() {
  // put the ball back in the center
  xPos = width/2;
  yPos = height/2;
}
```

Chapter 3

Modified Date page

Language: PHP

Prints the date. But no HTML.

```
<?php
//    Get the date, and format it:
$date = date("Y-m-d h:i:s\t");
// Include the date:
echo "< $date >\n";
?>
```

Parameter reader

Language: PHP

Prints any parameters sent in using an HTTP GET command.

```
<?php
// print the beginning of an HTML page:
echo "<html><head></head><body>\n";

// print out all the variables:
foreach ($_REQUEST as $key => $value)
   {
       echo "$key: $value<br>\n";
   }
// finish the HTML:
echo "</body></html>\n";
?>
```

Age checker

Language: PHP

Expects two parameters from the HTTP request:

- name (a text string)
- age (an integer)

Prints a personalized greeting based on the name and age.

```
<?php
// print the beginning of an HTML page:
echo "<html><head></head><body>\n";

// read all the parameters and assign them to local variables:
foreach ($_REQUEST as $key => $value) {
      if ($key == "name") {
          $name = $value;
      }
      if ($key == "age") {
          $age = $value;
      }
   }
```

```
if ($age < 21) {
    echo "<p> $name, You're not old enough to drink.</p>\n";
} else {
    echo "<p> Hi $name. You're old enough to have a drink, but do ";
    echo "so responsibly.</p>\n";
}
// finish the HTML:
echo "</body></html>\n";
?>
```

Analog sensor reader

Language: Arduino/Wiring

Reads an analog input on Analog in 0, prints the result as an ASCII-formatted decimal value.

Connections:

- FSR analog sensor on Analog in 0

```
int sensorValue;            // outgoing ADC value
void setup()
{
  // start serial port at 9600 bps:
  Serial.begin(9600);
}

void loop()
{
  sensorValue = analogRead(0); // read analog input.

  // send analog value out in ASCII decimal format:
  Serial.println(sensorValue, DEC);

  delay(10);  // wait 10ms for next reading.
}
```

Serial String Reader

Language: Processing

Reads in a string of characters until it gets a linefeed (ASCII 10). Then converts the string into a number. Then graphs it.

```
import processing.serial.*;

int graphPosition = 0;  // the horizontal position of the latest
                        // line to be drawn on the graph
int linefeed = 10;      // linefeed in ASCII
Serial myPort;          // The serial port
int sensorValue = 0;    // the value from the sensor
void setup() {
  size(400,300);
  // List all the available serial ports
  println(Serial.list());

  // I know that the first port in the serial list on my Mac
  // is always my Arduino, so I open Serial.list()[0].
  // Open whatever port is the one you're using (the output
  // of Serial.list() can help; the are listed in order
  // starting with the one that corresponds to [0]).
  myPort = new Serial(this, Serial.list()[0], 9600);

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil(linefeed);
}

void draw() {
  // twiddle your thumbs
}

// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():
void serialEvent(Serial myPort) {
  // read the serial buffer:
  String myString = myPort.readStringUntil(linefeed);

  // if you got any bytes other than the linefeed:
  if (myString != null) {
    // trim the carriage return and convert the string to an integer:
    sensorValue = int(trim(myString));

    println(sensorValue); // print it.
    drawGraph();
  }
}
void drawGraph() {
  // adjust this formula so that lineHeight is always less than
  // the height of the window:
  int lineHeight = sensorValue /2;

  stroke(0,255,0); // draw the line.
  line(graphPosition, height, graphPosition, height - lineHeight);
  // at the edge of the screen, go back to the beginning:
  if (graphPosition >= width) {
    graphPosition = 0;
    background(0);
  }
  else {
    graphPosition++;
  }
}
```

Cat graphing program

Language: Processing

Reads in a string of characters until it gets a linefeed (ASCII 10). Then converts the string into a number. Then graphs it. If the number has changed significantly, and there hasn't been a big change in more than a minute, the program prints a text string in place of an email message.

```
import processing.serial.*;

int linefeed = 10;          // linefeed in ASCII
Serial myPort;              // The serial port
int sensorValue = 0;        // the value from the sensor
int graphPosition = 0;      // the horizontal position of the latest
                            // line to be drawn on the graph
int prevSensorValue = 0;    // the previous sensor reading
boolean catOnMat = false;   // whether or not the cat's on the mat
int threshold = 320;        // above this number, the cat is on the mat.
```

» Change this number to reflect the threshold of your own sensor.

» Adjust this to an acceptable frequency for sending emails.

```
int timeThreshold = 1;    // minimum number of minutes between emails
int timeLastSent[] = {
  hour(), minute() - 1 }; // time the last message was sent

void setup() {
  size(400,300);
  // List all the available serial ports
  println(Serial.list());
  // I know that the first port in the serial list on my Mac
  // is always my  Arduino, so I open Serial.list()[0].
  // Open whatever port is the one you're using (the output
  // of Serial.list() can help; the are listed in order
  // starting with the one that corresponds to [0]).
  myPort = new Serial(this, Serial.list()[0], 9600);

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil(linefeed);
  println(hour() + ":" + minute());
}

void draw() {
  if (sensorValue > threshold ) {
    // if the last reading was less than the threshold,
    // then the cat just got on the mat.
    if (prevSensorValue <= threshold) {
      delay(100);
      if (sensorValue > threshold) {
        catOnMat = true;
        sendMail();
      }
    }
  } else {
    // if the sensor value is less than the threshold,
    // and the previous value was greater, then the cat
    // just left the mat
    if (prevSensorValue >= threshold) {
      catOnMat = false;
    }
  }
  // save the sensor value as the previous value
  // so you can take new readings:
  prevSensorValue = sensorValue;
}

// serialEvent method is run automatically by the Processing applet
// whenever the buffer reaches the byte value set in the bufferUntil()
// method in the setup():
void serialEvent(Serial myPort) {
  // read the serial buffer:
  String myString = myPort.readStringUntil(linefeed);
  // if you got any bytes other than the linefeed:
  if (myString != null) {
    // trim the carriage return and convert the string to an integer:
    sensorValue = int(trim(myString));
    drawGraph(); // call this method instead of println()
  }
}

void drawGraph() {
  int lineHeight = sensorValue /2;
  // draw the line:
  if (catOnMat) {
    stroke(0,255,0); // draw green
  }
  else {
    stroke(255,0,0); // draw red
  }
  line(graphPosition, height, graphPosition, height - lineHeight);
  // at the edge of the screen, go back to the beginning:
  if (graphPosition >= width) {
    graphPosition = 0;
    background(0);
  }
  else {
    graphPosition++;
  }
}

void sendMail() {
  // calculate the current time in minutes:
  int[] presentTime = { hour(), minute() };

  // print the current time and the last time you sent a message:
  print(sensorValue + "\t");
  print( presentTime[0] + ":" + presentTime[1] +"\t");
  println(timeLastSent[0] + ":" + timeLastSent[1]);
```

```
  // if you're still in the same hour as the last message,
  // then make sure at least the minimum number of minutes has passed:
  if (presentTime[0] == timeLastSent[0]) {
    if (presentTime[1] - timeLastSent[1] >= timeThreshold) {
      println("This is where you'd send a mail.");
      // take note of the time this message was sent:
      timeLastSent[0] = hour();
      timeLastSent[1] = minute();
    }
  }

  // if the hour has changed since the last message,
  // then the difference in minutes is a bit more complex.
  // Use != rather than > to make sure the shift
  // from 23:59 to 0:00 is covered as well:
  if (presentTime[0] != timeLastSent[0]) {
    // calculate the difference in minutes:
    int minuteDifference = (60 - timeLastSent[1]) + presentTime[1];

    if (minuteDifference >= timeThreshold) {
      println("This is where you'd send a mail.");

      // take note of the time this message was sent:
      timeLastSent[0] = hour();
      timeLastSent[1] = minute();
    }
  }
}
```

Cat On Mat

Language: PHP

Expects a parameter called SensorValue, an integer. Prints a custom message depending on the value of SensorValue.

```
<?php

$threshold = 320;    // minimum sensor value to trigger action

// print the beginning of the HTML page:
echo "<html><head></head><body>\n";

// read all the parameters and assign them to local variables:
foreach ($_REQUEST as $key => $value) {
      if ($key == "sensorValue") {
          $sensorValue = $value;
      }
   }
// Respond depending on the sensor value:
if ($sensorValue > $threshold) {
    echo "<p> The cat is on the mat.</p>\n";
} else {
    echo "<p> the cat is not on the mat.</p>\n";
}
// finish the HTML:
echo "</body></html>\n";
?>
```

Mail sender

Language: PHP

Expects a parameter called SensorValue, an integer. Sends an email if sensorValue is above a threshold value. This is an extension of the previous program. The previous one didn't actually send mail, but this one does.

```
<?php

$threshold = 320;    // minimum sensor value to trigger action.
```

▸▸ Change this number to reflect the threshold of your own sensor.

```
// print the beginning of an HTML page:
echo "<html><head></head><body>\n";

// read all the parameters and assign them to local variables:
foreach ($_REQUEST as $key => $value)
   {
      if ($key == "sensorValue") {
          $sensorValue = $value;
      }
   }

if ($sensorValue > $threshold) {
    $messageString =
      " The cat is on the mat at http://www.example.com/catcam.";
    echo $messageString;
    send_mail("yourname@example.com", "the cat", $messageString);
} else {
    echo "<p> the cat is not on the mat.</p>\n";
}
// finish the HTML:
echo "</body></html>\n";

end;

// end of the main script. Anything after here won't get run
// unless it's called in the code above this line

////////////////////////////////////////////

function send_mail($to, $subject, $message) {
    $from = "cat@example.com";
    mail($to, $subject, $message, "From: $from");
}
?>
```

HTTP sender

Language: Processing

Uses the Processing net library to make an HTTP request.

```
import processing.net.*;        // gives you access to the net library

Client client;                  // a new net client
boolean requestInProgress;      // whether a net request is in progress
String responseString = "";     // string of text received by client
void setup()
{
  // Open a connection to the host:
  client = new Client(this, "example.com", 80);

  // Send the HTTP GET request:
  client.write(
    "GET  /catcam/cat-script.php?sensorValue=321 HTTP/1.0\r\n");
  client.write("HOST: example.com\r\n\r\n");
  // note that you've got a request in progress:
  requestInProgress = true;
}

void draw()
{
  // available() returns how many bytes have been received by the client:
  if (client.available() > 0) {
    // read a byte, convert it to a character, and add it to the string:
    responseString +=char(client.read());

    // add to a line of |'s on the screen (crude progress bar):
    print("|");
  }
  // if there's no bytes available, either the response
  // hasn't started yet, or it's done:
  else {
  // if responseString is longer than 0 bytes, the response has started:
    if(responseString.length() > 0 )  {
      // you've got some bytes, but now there's no more to read. Stop:
      if(requestInProgress == true) {
        // print the response:
        println(responseString);
        // note that the request is over:
        requestInProgress = false;
        // reset the string  for future requests:
        responseString = "";
      }
    }
  }
}
```

Cat graphing and email program

Language: Processing

Reads in a string of characters until it gets a linefeed (ASCII 10). Then converts the string into a number. Then graphs it. If the number has changed significantly, and there hasn't been a big change in more than a minute, the program calls a PHP script to send an email message.

```
import processing.serial.*;
import processing.net.*;  // gives you access to the net library

int linefeed = 10;      // linefeed in ASCII
Serial myPort;          // The serial port
int sensorValue = 0;    // the value from the sensor
int graphPosition = 0;  // the horizontal position of the latest
                        // line to be drawn on the graph

int prevSensorValue = 0;    // the previous sensor reading
boolean catOnMat = false;   // whether or not the cat's on the mat
int threshold = 330;        // above this number, the cat is on the mat.

int timeThreshold = 1;    // minimum number of minutes between emails
int timeLastSent[] = {hour(), minute()}; // time last message was sent

// HTTP client variables:
Client client;                    // a new net client
boolean requestInProgress = false; // whether a net request is in progress
String responseString = "";       // string of text received by client

void setup() {
  size(400,300);
  // List all the available serial ports
  println(Serial.list());

  // I know that the first port in the serial list on my mac
  // is always my  Arduino, so I open Serial.list()[0].
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);
  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil(linefeed);
  println(hour() + ":" + minute());
}

void draw() {
  if (sensorValue > threshold ) {
    // if the last reading was less than the threshold,
    // then the cat just got on the mat.
    if (prevSensorValue <= threshold) {
      delay(100);
      if (sensorValue > threshold) {
        catOnMat = true;
        sendMail();
      }
    }
  }
```

```
  else {
    // if the sensor value is less than the threshold,
    // and the previous value was greater, then the cat
    // just left the mat
    if (prevSensorValue >= threshold) {
      catOnMat = false;
    }
  }
  // save the sensor value as the previous value
  // so you can take new readings:
  prevSensorValue = sensorValue;

  if (requestInProgress == true) {
    checkNetClient();
  }
}

// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():

void serialEvent(Serial myPort) {
  // read the serial buffer:
  String myString = myPort.readStringUntil(linefeed);
  // if you got any bytes other than the linefeed:
  if (myString != null) {
    //trim off the carriage return and convert the string to an integer:
    sensorValue = int(trim(myString)) /2 -100;
    // print it:
    // println(sensorValue);
    drawGraph();
  }
}

void drawGraph() {
  int lineHeight = sensorValue /2;
  // draw the line:
  if (catOnMat) {
    // draw green:
    stroke(0,255,0);
  }
  else {
    // draw red:
    stroke(255,0,0);
  }
  line(graphPosition, height, graphPosition, height - lineHeight);
  // at the edge of the screen, go back to the beginning:
  if (graphPosition >= width) {
    graphPosition = 0;
    background(0);
  }
  else {
    graphPosition++;
  }
}
```

```
void sendMail() {
  // calculate the current time in minutes:
  int[] presentTime = { hour(), minute() };

  // print the current time and the last time you sent a message:
  print(sensorValue + "\t");
  print( presentTime[0] + ":" + presentTime[1] +"\t");
  println(timeLastSent[0] + ":" + timeLastSent[1]);

  // if you're still in the same hour as the last message,
  // then make sure at least the minimum number of minutes has passed:
  if (presentTime[0] == timeLastSent[0]) {
    if (presentTime[1] - timeLastSent[1] >= timeThreshold) {
      println("This is where you'd send a mail.");
      makeHTTPCall();
      // take note of the time this message was sent:
      timeLastSent[0] = hour();
      timeLastSent[1] = minute();
    }
  }

  // if the hour has changed since the last message,
  // then the difference in minutes is a bit more complex.
  // Use !+ rather than > to make sure the shift
  // from 23:59 to 0:00 is covered as well:
  if (presentTime[0] != timeLastSent[0]) {
    // calculate the difference in minutes:
    int minuteDifference = (60 - timeLastSent[1]) + presentTime[1];

    if (minuteDifference >= timeThreshold) {
      println("This is where you'd send a mail.");
      makeHTTPCall();
      // take note of the time this message was sent:
      timeLastSent[0] = hour();
      timeLastSent[1] = minute();
    }
  }
}

void makeHTTPCall() {
  if (requestInProgress == false) {
    // Open a connection to the host:
    client = new Client(this, "example.com", 80);

    // form the request string:
    String requestString = "/cat-script.php?sensorValue=" +
      sensorValue;
    println(requestString);
    // Send the HTTP GET request:
    client.write("GET  " + requestString + " HTTP/1.1\n");
    client.write("HOST: example.com\n\n");
    // note that you've got a request in progress:
    requestInProgress = true;
  }
}
```

```
void checkNetClient() {
  // available() returns how many bytes have been received by the client:
  if (client.available() > 0) {
    // read a byte, convert it to a character, and add it to the string:
    responseString +=char(client.read());

    // add to a line of |'s on the screen (crude progress bar):
    print("|");
  }
  // if there's no bytes available, either the response hasn't
  // started yet, or it's done:
  else {
    // if responseString is longer than 0 bytes, the response has started:
    if(responseString.length() > 0 )  {
      // you've got some bytes, but now there's no more to read. Stop:
      if(requestInProgress == true) {
        // print the response:
        println(responseString);
        // note that the request is over:
        requestInProgress = false;
        // reset the string  for future requests:
        responseString = "";
      }
    }
  }
}
```

Chapter 4

AIRNow Web Page Scraper

Language: PHP

Reads a web page and returns one line from it.

```
<?php
    // url of the air quality index page for New York City:
    $url =
      'http://airnow.gov/index.cfm?action=airnow.showlocal&cityid=164';
    // open the file at the URL for reading:
    $filePath = fopen ($url, "r");

    // as long as you haven't reached the end of the file:
    while (!feof($filePath))
    {
        // read one line at a time, and strip all HTML and
        // PHP tags from the line:
        $line = fgetss($filePath, 4096);
        echo $line;
    }
    fclose($filePath); // close the file at the URL, you're done!
?>
```

Air Quality meter

Language: Wiring/Arduino (*pin numbers defined for Arduino*)

Microcontroller is connected to a Lantronix serial-to-ethernet device. This program connects to a HTTP server through the Lantronix module, makes a HTTP GET request for a PHP script, and parses the returned string. Lantronix device communicates at 9600-8-n-1 non-inverted (true) serial. Lantronix serial settings:

- Baudrate 9600, I/F Mode 4C, Flow 00
- Port 10001
- Remote IP Addr: --- none ---, Port 00000
- Connect Mode : D4
- Disconn Mode : 00
- Flush Mode : 00

```
// Defines for the  program's status (used for status variable):
#define disconnected 0
#define connecting 1
#define connected 2
#define requesting 3
#define reading 4
#define requestComplete 5

// Defines for I/O pins:
#define connectedLED 2   // indicates when there's a TCP connection
#define requestingLED 3  // indicates a HTTP request has been made
#define readingLED 4     // indicates device is reading HTTP results
```

```
#define requestCompleteLED 5  // indicates a successful read
#define programResetLED 6     // indicates reset of Arduino
#define deviceResetPin 7      // resets Lantronix Device
#define meterPin 11           // controls VU meter

// defines for voltmeter:
#define meterMax 130          // max value on the meter
#define meterScale 150        // my meter reads 0 - 150

// variables:
int inByte= -1;              // incoming byte from serial RX
char inString[32];           // string for incoming serial data
int stringPos = 0;           // string index counter
int status = 0;              // Lantronix device's connection status
long lastCompletionTime = 0; // counter for delay after last completion

void setup() {
  // set all status LED pins and Lantronix device reset pin:
  pinMode(connectedLED, OUTPUT);
  pinMode(requestingLED, OUTPUT);
  pinMode(requestCompleteLED, OUTPUT);
  pinMode(programResetLED, OUTPUT);
  pinMode(deviceResetPin, OUTPUT);
  pinMode(meterPin, OUTPUT);

  // start serial port, 9600 8-N-1:
  Serial.begin(9600);
  //reset Lantronix device:
  resetDevice();
  // blink reset LED:
  blink(3);
}

void loop() {
  stateCheck();
  setLEDs();
}

/*
  Check the status of the connection and take appropriate action:
 */
void stateCheck() {
  switch (status) {
  case disconnected:
    // attempt to connect to the server:
    deviceConnect();
    break;
  case connecting:
    // until you get a C, keep trying to connect:
    // read the serial port:
    if (Serial.available()) {
      inByte = Serial.read();
      if (inByte == 'C') {  // 'C' in ascii
        status = connected;
      } else {
        // if you got anything other than a C, try again:
        deviceConnect();
      }
    }
    break;
  case connected:
    // send HTTP GET request for CGI script:
    httpRequest();
    break;
  case requesting:
    lookForData();
    break;
  case reading:
    readData();
    break;
  case requestComplete:
    waitForNextRequest();
  }
}

/*
  Set the indicator LEDs according to the state of the program
 */
void setLEDs() {
  /*
   Except for the disconnected and connecting states,
   all the states of the program have corresponding LEDS.
   so you can use a for-next loop to set them  by
   turning them all off except for the one that has
   the same number as the current program state:
   */
  for (int thisLED = 2; thisLED <= 5; thisLED++) {
    if (thisLED == status) {
      digitalWrite(thisLED, HIGH);
    }
    else {
      digitalWrite(thisLED, LOW);
    }
  }
}

/*
  Command the Lantronix device to connect to the server
 */
void deviceConnect() {
  //   fill in your server's numerical address below:
  Serial.print("C82.165.199.35/80\n");
  status = connecting;
}
```

```
/*
  Send a HTTP GET request
 */
void httpRequest() {
  // make sure you've cleared the last byte
  // from the last request:
  inByte = -1;
  // reset the string position counter:
  stringPos = 0;
  //  Make HTTP GET request. Fill in the path to your version
  //  of the CGI script:
  Serial.print("GET /~myaccount/scraper.php HTTP/1.0\n");
  //  Fill in your server's name:
  Serial.print("HOST:example.com\n\n");
  // update the state of the program:
  status = requesting;
}

/*
  Read the results sent by the server until you get a < character.
 */
void lookForData() {
  // wait for bytes from server:
  if (Serial.available()) {
    inByte = Serial.read();
    // If you get a "<", what follows is the air quality index.
    // You need to read what follows the <.
    if (inByte == '<') {
      stringPos = 0;
      status = reading;
    }
  }
}
/*
  read the number from the server into an array, terminating with a > character.
 */
void readData() {
  if (Serial.available()) {
    inByte = Serial.read();
    // Keep reading until you get a ">":
    if (inByte != '>') {
      // save only ASCII numeric characters (ASCII 0 - 9):
      if ((inByte >= '0') && (inByte <= '9')){
        inString[stringPos] = inByte;
        stringPos++;
      }
    }
    // if you get a ">", you've reached the end of the AQI reading:
    else {
      interpretResults();
    }
  }
}
```

```
/*
  convert the input string to an integer.
 */
void interpretResults() {
  // convert the string to a numeric value:
  int airQuality = atoi(inString);
  setMeter(airQuality); // set the meter appropriately.
  lastCompletionTime = millis();
  status = requestComplete;
}

/*
  scale the number from the request to the meter's range and set the meter.
 */
void setMeter(int desiredValue) {
  int airQualityValue = 0;
  // if the value won't peg the meter, convert it
  // to the meter scale and send it out:
  if (desiredValue <= meterScale) {
    airQualityValue = (desiredValue * meterMax /meterScale) ;
    analogWrite(meterPin, airQualityValue);
  }
}

/*
  Wait two minutes before initiating a new request.
 */
void waitForNextRequest() {
  if (millis() - lastCompletionTime >= 120000) {
    resetDevice(); // reset Lantronix device before next request.
    status = disconnected;
  }
}

/*
  Take the Lantronix device's reset pin low to reset it
 */
void resetDevice() {
  digitalWrite(deviceResetPin, LOW);
  delay(50);
  digitalWrite(deviceResetPin, HIGH);
  delay(2000); // pause to let Lantronix device boot up.
}
/*
   Blink the reset LED.
 */
void blink(int howManyTimes) {
  int i;
  for (i=0; i< howManyTimes; i++) {
    digitalWrite(programResetLED, HIGH);
    delay(200);
    digitalWrite(programResetLED, LOW);
    delay(200);
  }
}
```

SoftwareSerial example

Language: Wiring/Arduino

This program uses the SoftwareSerial library to send serial messages on pins 8 and 9.

```
// include the SoftwareSerial library so you can use its functions:
#include <SoftwareSerial.h>

#define rxPin 8
#define txPin 9

// set up a new serial port
SoftwareSerial mySerial =  SoftwareSerial(rxPin, txPin);

void setup()  {
  // define pin modes for tx, rx, led pins:
  pinMode(rxPin, INPUT);
  pinMode(txPin, OUTPUT);
  // set the data rate for the SoftwareSerial port
  mySerial.begin(9600);
}

void loop() {
  // print out a debugging message:
  mySerial.println("Hello from SoftwareSerial");
  delay(100);
}
```

SoftwareSerial example

Language: Wiring/Arduino

This program is a variation on the networked air quality meter. It uses the SoftwareSerial library to send serial messages on pins 8 and 9.

```
// include the SoftwareSerial library so you can use its functions:
#include <SoftwareSerial.h>

#define rxPin 8
#define txPin 9

// Defines go here

// variables go here

// set up a new serial port
SoftwareSerial mySerial =  SoftwareSerial(rxPin, txPin);

void setup() {
  // the rest of the setup() code goes here

  // define pin modes for SoftwareSerial tx, rx pins:
  pinMode(rxPin, INPUT);
  pinMode(txPin, OUTPUT);
  // set the data rate for the SoftwareSerial port
  mySerial.begin(9600);

  // print out a debugging message:
  mySerial.println("All set up");

}

void loop() {
  stateCheck();
  setLEDs();
}

void stateCheck() {

  // the rest of stateCheck() code goes here
}

void setLEDs() {
  // setLEDs() code goes here
}

void deviceConnect() {
  // print out a debugging message:
  mySerial.println("connect");

  //   the rest of deviceConnect() code goes here
}

void httpRequest() {
  // print out a debugging message:
  mySerial.println("request");

  // the rest of httpRequest() code goes here
}

void lookForData() {

  // wait for bytes from server:
  if (Serial.available()) {
    inByte = Serial.read();
    mySerial.print(inByte, BYTE);

    // the rest of lookForData() code goes here
}

void readData() {
  if (Serial.available()) {
    inByte = Serial.read();
    mySerial.print(inByte, BYTE);

    // the rest of readData() code goes here
}
```

```
void interpretResults() {
  // print out a debugging message:
  mySerial.println("interpret");

  // the rest of interpretResults() code goes here

  mySerial.println("wait"); // print out a debugging message.
}

void setMeter(int desiredValue) {
  mySerial.println("set"); // print out a debugging message.

  // the rest of setMeter() code goes here
}

void resetDevice() {
  mySerial.println("reset"); // print out a debugging message.

  // the rest of resetDevice() code goes here
}
/*
   Blink the reset LED.
 */
void blink(int howManyTimes) {
  int i;
  for (i=0; i< howManyTimes; i++) {
    digitalWrite(programResetLED, HIGH);
    delay(200);
    digitalWrite(programResetLED, LOW);
    delay(200);
  }
}
```

Lantronix serial-to-Ethernet HTTP request tester

Language: Processing

This program sends serial messages to a Lantronix serial-to-Ethernet device to get it to connect to a remote webserver and make an HTTP request. To use this program, connect your PC to the Lantronix module's serial port as you did when you were configuring the Lantronix module earlier.

```
// include the serial library
import processing.serial.*;

Serial myPort;       // Serial object
int step = 0;        // which step in the process you're on
char linefeed = 10;  // ASCII linefeed character

void setup()
{
  // get the list of serial ports:
  println(Serial.list());
  // open the serial port appropriate to your computer:
  myPort = new Serial(this, Serial.list()[2], 9600);
  // configure the serial object to buffer text until it receives a
  // linefeed character:
  myPort.bufferUntil(linefeed);
}

void draw()
{
  //no action in the draw loop
}

void serialEvent(Serial myPort) {
  // print any string that comes in serially to the monitor pane
  print(myPort.readString());
}

void keyReleased() {
  // if any key is pressed, take the next step:
  switch (step) {
  case 0:
    // open a connection to the server in question:
    myPort.write("C208.201.239.37/80\r");
    // add one to step so that the next keystroke causes the next step:
    step++;
    break;
  case 1:
    // send a HTTP GET request
    myPort.write("GET /~igoe/index.html HTTP/1.0\n");
    myPort.write("HOST:example.com\n\n");
    step++;
    break;
  }
}
```

Test Server Program

Language: Processing

Creates a server that listens for clients and prints what they say. It also sends the last client anything that's typed on the keyboard.

```
// include the net library:
import processing.net.*;

int port = 8080;        // the port the server listens on
Server myServer;        // the server object
Client  thisClient;     // incoming client object

void setup()
{
  myServer = new Server(this, port); // Start the server
}

void draw()
{
  // get the next client that sends a message:
  Client speakingClient = myServer.available();

  // if the message is not null, display what it sent:
  if (speakingClient !=null) {
    String whatClientSaid = speakingClient.readString();
    // print who sent the message, and what they sent:
    println(speakingClient.ip() + "\t" + whatClientSaid);
  }
}

// ServerEvent message is generated when a new client
// connects to the server.
void serverEvent(Server myServer, Client someClient) {
  println("We have a new client: " + someClient.ip());
  thisClient = someClient;
}

void keyReleased() {
  // only send if there's a client to send to:
  if (thisClient != null) {
    // if return is pressed, send newline and carriage feed:
    if (key == '\n') {
      thisClient.write("\r\n");
    }
    // send any other key as is:
    else {
      thisClient.write(key);
    }
  }
}
```

Chapter 5

Pong client

Language: Wiring/Arduino

This program enables an Arduino to control one paddle in a networked Pong game. This listing uses the readSensors() method from the seesaw client in project #7.

```
// Defines for the Lantronix device's status (used for staus variable):
#define disconnected 0
#define connected 1
#define connecting 2

// Defines for I/O pins:
#define connectButtonPin 2
#define rightLED 3
#define leftLED 4
#define connectionLED 5
#define connectButtonLED 6
#define deviceResetPin 7
// variables:
int inByte= -1;                  // incoming byte from serial RX
int status = disconnected;       // Lantronix device's connection status

// variables for the sensors:
byte connectButton = 0;          // state of the exit button
byte lastConnectButton = 0;      // previous state of the exit button
/*
 When the connect button is pressed, or the accelerometer
 passes the left or right threshold, the client should send a message
 to the server.  The next two variables get filled with a value
 when either of those conditions is met.  Otherwise, these
 variables are set to 0.
 */
byte paddleMessage = 0;       // message sent to make a paddle move
byte connectMessage = 0;      // message sent to connect or disconnect

void setup() {
  // set the modes of the various I/O pins:
  pinMode(connectButtonPin, INPUT);
  pinMode(rightLED, OUTPUT);
  pinMode(leftLED, OUTPUT);
  pinMode(connectionLED, OUTPUT);
  pinMode(connectButtonLED, OUTPUT);
  pinMode(deviceResetPin, OUTPUT);

  // start serial port, 9600 8-N-1:
  Serial.begin(9600);

  // reset the Lantronix device:
  resetDevice();
  // blink the exit button LED to signal that we're ready for action:
  blink(3);
}
```

```
void loop() {
  // read the inputs:
  readSensors();
  // set the indicator LEDS:
  setLeds();
  // check the state of the client and take appropriate action:
  stateCheck();
}
void readSensors() {
  // thresholds for the accelerometer values:
  int leftThreshold = 500;
  int rightThreshold = 420;

  // read the X axis of the accelerometer:
  int x = analogRead(0);
  // let the analog/digital converter settle:
  delay(10);

  // if the accelerometer has passed either threshold,
  // set paddleMessage to the appropriate message, so it can
  // be sent by the main loop:
  if (x > leftThreshold) {
    paddleMessage = 'l';
  } else if (x < rightThreshold) {
    paddleMessage = 'r';
  } else {
    paddleMessage = 0;
  }
// read the connectButton, look for a low-to-high change:
  connectButton = digitalRead(connectButtonPin);
  connectMessage = 0;
  if (connectButton == HIGH ) {
    if (connectButton != lastConnectButton) {
      // turn on the exit button LED to let the user
      // know that they hit the button:
      digitalWrite(connectButtonLED, HIGH);
      connectMessage = 'x';
    }
  }
  // save the state of the exit button for next time you check:
  lastConnectButton = connectButton;
}
void setLeds() {
  // this should happen no matter what state the client is in,
  // to give local feedback every time a sensor senses a change

  // set the L and R LEDs if the sensor passes the appropriate
threshold:
  switch (paddleMessage) {
  case 'l':
    digitalWrite(leftLED, HIGH);
    digitalWrite(rightLED, LOW);
    break;
  case 'r':
    digitalWrite(rightLED, HIGH);
    digitalWrite(leftLED, LOW);
    break;
  case 0:
    digitalWrite(rightLED, LOW);
    digitalWrite(leftLED, LOW);
  }

  // set the connect button LED based on the connectMessage:
  if (connectMessage !=0) {
    digitalWrite(connectButtonLED, HIGH);
  }
  else {
    digitalWrite(connectButtonLED, LOW);
  }

  // set the connection LED based on the client's status:
  if (status == connected) {
    // turn on the connection LED:
    digitalWrite(connectionLED, HIGH);
  }
  else {
    // turn off the connection LED:
    digitalWrite(connectionLED, LOW);
  }
}
void stateCheck() {
  // Everything in this method depends on the client's status:
  switch (status) {
  case connected:
    // if you're connected, listen for serial in:
    while (Serial.available() > 0) {
      // if you get a 'D', it's from the Lantronix device,
      // telling you that it lost the connection:
      if (Serial.read() == 'D') {
        status = disconnected;
      }
    }

    // if there's a paddle message to send, send it:
    if (paddleMessage != 0) {
      Serial.print(paddleMessage);
      // reset paddleMessage to 0 once you've sent the message:
      paddleMessage = 0;
    }
    // if there's a connect message to send, send it:
    if (connectMessage != 0) {
      // if you're connected, disconnect:
      Serial.print(connectMessage);
      // reset connectMessage to 0 once you've sent the message:
      connectMessage = 0;
    }
    break;
```

```
  case disconnected:
    // if there's a connect message, try to connect:
    if (connectMessage !=0 ) {
      deviceConnect();
      // reset connectMessage to 0 once you've sent the message:
      connectMessage = 0;
    }
    break;
    // if you sent a connect message but haven't connected yet,
    // keep trying:
  case connecting:
    // read the serial port:
    if (Serial.available()) {
      inByte = Serial.read();
      // if you get a 'C' from the Lantronix device,
      // then you're connected to the server:
      if (inByte == 'C') {
        status = connected;
      }
      else {
        // if you got anything other than a C, try again:
        deviceConnect();
      }
    }
    break;
  }
}
void deviceConnect() {
  /*
     send out the server address and
   wait for a "C" byte to come back.
   fill in your personal computer's numerical address below:
   */
  Serial.print("C192.168.1.20/8080\n\r");
  status = connecting;
}

// Take the Lantronix device's reset pin low to reset it:
void resetDevice() {
  digitalWrite(deviceResetPin, LOW);
  delay(50);
  digitalWrite(deviceResetPin, HIGH);
  // pause to let Lantronix device boot up:
  delay(2000);
}

// Blink the connect button LED:
void blink(int howManyTimes) {
  for (int i=0; i< howManyTimes; i++) {
    digitalWrite(connectButtonLED, HIGH);
    delay(200);
    digitalWrite(connectButtonLED, LOW);
    delay(200);
  }
}
```

Pong Server

Language: Processing

This program listens for TCP socket connections and uses the data from the incoming connections in a networked multiplayer version of pong.

```
// include the net library:
import processing.net.*;

// variables for keeping track of clients:
int port = 8080;                  // the port the server listens on
Server myServer;                  // the server object
ArrayList playerList = new ArrayList(); // list of clients

// Variables for keeping track of the game play and graphics:
int ballSize = 10;              // the size of the ball
int ballDirectionV = 2;         // the ball's horizontal direction.
                                // left is negative, right is positive
int ballDirectionH = 2;         // the ball's vertical direction.
                                // up is negative, down is positive
int ballPosV, ballPosH;         // the ball's vertical/horizontal
                                // and vertical positions
boolean ballInMotion = false;   // whether the ball should be moving

int topScore, bottomScore;      // scores for the top team and
                                // the bottom teams
int paddleHeight = 10;          // vertical dimension of the paddles
int paddleWidth = 80;           // horizontal dimension of the paddles
int nextTopPaddleV;             // paddle positions for the next player
                                // to be created
int nextBottomPaddleV;

boolean gameOver = false;           // whether a game is in progress
float delayCounter = millis();      // a counter for the delay after
                                    // a game is over
long gameOverDelay = 4000;          // pause after each game
long pointDelay = 2000;             // pause after each point

void setup()  {
  // set up all the pong details:
  pongSetup();
  // Start the server:
  myServer = new Server(this, port);
}

void pongSetup() {
  // set the window size:
  size(480, 640);
  // set the frame rate:
  frameRate(90);

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 18);
  textFont(myFont);
```

```
  // set the default font settings:
  textFont(myFont, 18);
  textAlign(CENTER);

  // initalize paddle positions for the first player.
  // these will be incremented with each new player:
  nextTopPaddleV = 50;
  nextBottomPaddleV = height - 50;

  // initialize the ball in the center of the screen:
  ballPosV = height / 2;
  ballPosH = width / 2;

  // set no borders on drawn shapes:
  noStroke();
  // set the rectMode so that all rectangle dimensions
  // are from the center of the rectangle (see Processing reference):
  rectMode(CENTER);
}

void draw() {
  pongDraw();
  listenToClients();
}

// The ServerEvent message is generated when a new client
// connects to the server.
void serverEvent(Server someServer, Client someClient) {
  boolean isPlayer = false;

  if (someClient != null) {
    // iterate over the playerList:
    for (int p = 0; p < playerList.size(); p++) {
      // get the next object in the ArrayList and convert it
      // to a Player:
      Player thisPlayer = (Player)playerList.get(p);

      // if thisPlayer's client matches the one that generated
      // the serverEvent, then this client is already a player:
      if (thisPlayer.client == someClient) {
        // we already have this client
        isPlayer = true;
      }
    }

    // if the client isn't already a Player, then make a new Player
    // and add it to the playerList:
    if (!isPlayer) {
      makeNewPlayer(someClient);
    }
  }
}

void makeNewPlayer(Client thisClient) {
  // paddle position for the new Player:
  int h = width/2;
  int v = 0;

  /*
  Get the paddle position of the last player on the list.
   If it's on top, add the new player on the bottom, and vice versa.
   If there are  no other players, add the new player on the top.
   */
  // get the size of the list:
  int listSize = playerList.size() - 1;
  // if there are any other players:
  if  (listSize >= 0) {
    // get the last player on the list:
    Player lastPlayerAdded = (Player)playerList.get(listSize);
    // is the last player's on the top, add to the bottom:
    if (lastPlayerAdded.paddleV == nextTopPaddleV) {
      nextBottomPaddleV = nextBottomPaddleV - paddleHeight * 2;
      v = nextBottomPaddleV;
    }
    // is the last player's on the bottom, add to the top:
    else if (lastPlayerAdded.paddleV == nextBottomPaddleV) {
      nextTopPaddleV = nextTopPaddleV + paddleHeight * 2;
      v = nextTopPaddleV;
    }
  }
  // if there are no players, add to the top:
  else {
    v = nextTopPaddleV;
  }

  // make a new Player object with the position you just calculated
  // and using the Client that generated the serverEvent:
  Player newPlayer = new Player(h, v, thisClient);
  // add the new Player to the playerList:
  playerList.add(newPlayer);
  // Announce the new Player:
  print("We have a new player: ");
  println(newPlayer.client.ip());
  newPlayer.client.write("hi\r\n");
}

void listenToClients() {
  // get the next client that sends a message:
  Client speakingClient = myServer.available();
  Player speakingPlayer = null;

  // iterate over the playerList to figure out whose
  // client sent the message:
  for (int p = 0; p < playerList.size(); p++) {
    // get the next object in the ArrayList and convert it
    // to a Player:
    Player thisPlayer = (Player)playerList.get(p);
```

```
    // compare the client of thisPlayer to the client that sent a
    // message. If they're the same, then this is the Player we want:
    if (thisPlayer.client == speakingClient) {
      speakingPlayer = thisPlayer;
    }
  }

  // read what the client sent:
  if (speakingPlayer != null) {
    int whatClientSaid = speakingPlayer.client.read();
    /*
     There a number of things it might have said that we care about:
     x = exit
     l = move left
     r = move right
     */
    switch (whatClientSaid) {
      // If the client says "exit", disconnect it
    case 'x':
      // say goodbye to the client:
      speakingPlayer.client.write("bye\r\n");
      // disconnect the client from the server:
      println(speakingPlayer.client.ip() + "\t left");
      myServer.disconnect(speakingPlayer.client);
      // remove the client's Player from the playerList:
      playerList.remove(speakingPlayer);
      break;
    case 'l':
      // if the client sends an "l", move the paddle left
      speakingPlayer.movePaddle(-10);
      break;
    case'r':
      // if the client sends a "r", move the paddle right
      speakingPlayer.movePaddle(10);
      break;
    }
  }
}

void pongDraw() {
  background(0);
  // draw all the paddles
  for (int p = 0; p < playerList.size(); p++) {
    Player thisPlayer = (Player)playerList.get(p);
    // show the paddle for this player:
    thisPlayer.showPaddle();
  }

  // calculate ball's position:
  if (ballInMotion) {
    moveBall();
  }
  // Draw the ball:
  rect(ballPosH, ballPosV, ballSize, ballSize);

  // show the score:
  showScore();

  // if the game is over, show the winner:
  if (gameOver) {
    textSize(24);
    gameOver = true;
    text("Game Over", width/2, height/2 - 30);
    if (topScore > bottomScore) {
      text("Top Team Wins!", width/2, height/2);
    }
    else {
      text("Bottom Team Wins!", width/2, height/2);
    }
  }
  // pause after each game:
  if (gameOver && (millis() > delayCounter + gameOverDelay)) {
    gameOver = false;
    newGame();
  }
  // pause after each point:
  if (!gameOver && !ballInMotion && (millis() >
    delayCounter + pointDelay)) {

    // make sure there are at least two players:
    if (playerList.size() >=2) {
      ballInMotion = true;
    }
    else {
      ballInMotion = false;
      textSize(24);
      text("Waiting for two players", width/2, height/2 - 30);
    }
  }
}

void moveBall() {
  // Check to see if the ball contacts any paddles:
  for (int p = 0; p < playerList.size(); p++) {
    // get the player to check:
    Player thisPlayer = (Player)playerList.get(p);

    // calculate the horizontal edges of the paddle:
    float paddleRight = thisPlayer.paddleH + paddleWidth/2;
    float paddleLeft = thisPlayer.paddleH - paddleWidth/2;
    // check whether the ball is in the horizontal range of the paddle:
    if ((ballPosH >= paddleLeft) && (ballPosH <= paddleRight)) {

      // calculate the vertical edges of the paddle:
      float paddleTop = thisPlayer.paddleV - paddleHeight/2;
      float paddleBottom = thisPlayer.paddleV + paddleHeight/2;
```

```
      // check to see if the ball is in the
      // horizontal range of the paddle:
      if ((ballPosV >= paddleTop) && (ballPosV <= paddleBottom)) {
        // reverse the ball vertical direction:
        ballDirectionV = -ballDirectionV;
      }
    }
  }

  // if the ball goes off the screen top:
  if (ballPosV < 0) {
    bottomScore++;
    ballDirectionV = int(random(2) + 1) * -1;
    resetBall();
  }
  // if the ball goes off the screen bottom:
  if (ballPosV > height) {
    topScore++;
    ballDirectionV = int(random(2) + 1);
    resetBall();
  }

  // if any team goes over 5 points, the other team loses:
  if ((topScore > 5) || (bottomScore > 5)) {
    delayCounter = millis();
    gameOver = true;
  }

  // stop the ball going off the left or right of the screen:
  if ((ballPosH - ballSize/2 <= 0) || (ballPosH +ballSize/2 >=width)) {
    // reverse the y direction of the ball:
    ballDirectionH = -ballDirectionH;
  }
  // update the ball position:
  ballPosV = ballPosV + ballDirectionV;
  ballPosH = ballPosH + ballDirectionH;
}

void newGame() {
  gameOver = false;
  topScore = 0;
  bottomScore = 0;
}

public void showScore() {
  textSize(24);
  text(topScore, 20, 40);
  text(bottomScore, 20, height - 20);
}
```

```
void resetBall() {
  // put the ball back in the center
  ballPosV = height/2;
  ballPosH = width/2;
  ballInMotion = false;
  delayCounter = millis();
}

public class Player {
  // declare variables that belong to the object:
  float paddleH, paddleV;
  Client  client;

  public Player (int hpos, int vpos, Client someClient) {
    // initialize the localinstance variables:
    paddleH = hpos;
    paddleV = vpos;
    client = someClient;
  }

  public void movePaddle(float howMuch) {
    float newPosition = paddleH + howMuch;
    // constrain the paddle's position to the width of the window:
    paddleH = constrain(newPosition, 0, width);
  }

  public void showPaddle() {
    rect(paddleH, paddleV, paddleWidth, paddleHeight);
    // display the address of this player near its paddle
    textSize(12);
    text(client.ip(), paddleH, paddleV - paddleWidth/8 );
  }
}
```

Chapter 6

IR transmit example

Language: Wiring/Arduino

This program reads an analog input on pin 0 and sends the result out as an ASCII-encoded string. The TX line of the microcontroller is connected to a Rentron TX-IRHS IR transmitter which can transmit at 19200 bps.

```
void setup(){
  //Open the serial port at 19200 bps:
  Serial.begin(19200);
}
void loop(){
  // Read the analog input:
  int analogValue = analogRead(0);
  // send the value out via the transmitter:
  Serial.println(analogValue, DEC);
  // delay 10ms to allow the analog-to-digital receiver to settle:
  delay(10);
}
```

RF Transmitter

Language: Wiring/Arduino

This program reads an analog input on pin 0 and sends the result out as an ASCII-encoded string. The TX line of the microcontroller is connected to an RF transmitter that is capable of reading at 2400 bps.

```
void setup(){
  // Open the serial port at 2400 bps:
  Serial.begin(2400);
}
void loop(){
  // Read the analog input:
  int analogValue = analogRead(0);
  // send the value out via the transmitter:
  Serial.println(analogValue, DEC);
  // delay 10ms to allow the analog-to-digital receiver to settle:
  delay(10);
}
```

RF Receive

Language: Processing

This program listens for data coming in through a serial port. It reads a string and throws out any strings that contain values other than ASCII numerals, linefeed, or carriage return, or that are longer than four digits. This program is designed to work with a Laipac RF serial receiver connected to the serial port, operating at 2400 bps.

```
import processing.serial.*;

Serial myPort;            // the serial port
int incomingValue = 0;    // the value received in the serial port

void setup() {
  // list all the available serial ports:
  println(Serial.list());

  // open the appropriate serial port.  On my computer, the RF
  // receiver is connected to a USB-to-serial adaptor connected to
  // the first port in the list. It may be on a different port on
  // your machine:
  myPort = new Serial(this, Serial.list()[0], 2400);
  // tell the serial port not to generate a serialEvent
  //until a linefeed is received:
  myPort.bufferUntil('\n');
}

void draw() {
  // set the background color according to the incoming value:
  background(incomingValue/4);
}

// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():
void serialEvent(Serial myPort) {
  boolean validString = true;  // whether the string you got is valid
  String errorReason = "";     // a string that tells what went wrong

  // read the serial buffer:
  String myString = myPort.readStringUntil('\n');

  // make sure you have a valid string:
  if (myString != null) {
    // trim off the whitespace (linefeed, carriage return) characters:
    myString = trim(myString);

    // check for garbage characters:
    for (int charNum = 0; charNum < myString.length(); charNum++) {
      if (myString.charAt(charNum) < '0' ||
          myString.charAt(charNum) > '9') {
```

```
        // you got a garbage byte; throw the whole string out
        validString = false;
        errorReason =
          "Received a byte that's not a valid ASCII numeral.";
      }
    }
    // check to see that the string length is appropriate:
    if (myString.length() > 4) {
      validString = false;
      errorReason = "Received more than 4 bytes.";
    }
    // if all's good, convert the string to an int:
    if (validString == true) {
      incomingValue = int(trim(myString));
      println("Good value: " + incomingValue);
    } else {
      // if the data is bad, say so:
      println("Error: Data is corrupted. " + errorReason);
    }
  }
}
```

XBee terminal

Language: Processing

This program is a basic serial terminal program. It replaces newline characters from the keyboard with return characters. You need it to talk to XBee radios with Linux/Unix/Mac OS X because the XBees don't send newline characters back.

```
import processing.serial.*;

Serial myPort;           // the serial port you're using
String portnum;          // name of the serial port
String outString = "";   // the string being sent out the serial port
String inString = "";    // the string coming in from the serial port
int receivedLines = 0;   // how many lines have been received
int bufferedLines = 10;  // number of incoming lines to keep

void setup() {

  size(400, 300);        // window size
  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 14);
  textFont(myFont);

  // list all the serial ports:
  println(Serial.list());
  // based on the list of serial ports printed from the
  // previous command, change the 0 to your port's number:
  portnum = Serial.list()[0];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 9600);
}

void draw() {
  background(0); // clear the screen.
  // print the name of the serial port:
  text("Serial port: " + portnum, 10, 20);
  // Print out what you get:
  text("typed: " + outString, 10, 40);
  text("received:\n" + inString, 10, 80);
}

// This method responds to key presses when the
// program window is active:
void keyPressed() {
  switch (key) {
    // In Unix/Linux/OS X, if the user types return, a linefeed is
    // returned. But the XBee wants a carriage return:
  case '\n':
    myPort.write(outString + "\r");
    outString = "";
    break;
  case 8:    // backspace
    // delete the last character in the string:
    outString = outString.substring(0, outString.length() -1);
    break;
  case '+':  // we have to send the + signs even without a return:
    myPort.write(key);
    // add the key to the end of the string:
    outString += key;
    break;
  case 65535:  // If the user types the shift key, don't type anything:
    break;
  default:     // any other key typed, add it to outString:
    // add the key to the end of the string:
    outString += key;
    break;
  }
}

// this method runs when bytes show up in the serial port:
void serialEvent(Serial myPort) {
  // read the next byte from the serial port:
  int inByte = myPort.read();
  // add it to  inString:
  inString += char(inByte);
  if (inByte == '\r') {
    // if the byte is a carriage return, print
    // a newline and carriage return:
    inString += '\n';
    // count the number of newlines:
    receivedLines++;
    // if there are more than 10 lines, delete the first one:
    if (receivedLines >  bufferedLines) {
      deleteFirstLine();
    }
  }
}
```

```
// deletes the top line of inString so that it all fits on the screen:
void deleteFirstLine() {
  // find the first newline:
  int firstChar = inString.indexOf('\n');
  // delete it:
  inString= inString.substring(firstChar+1);
}
```

XBee Analog Duplex sender

Language: Wiring/Arduino

This sketch configures an XBee radio via the serial port, sends the value of an analog sensor out, and listens for input from the radio, using it to set the value of a PWM output. Thanks to Robert Faludi for the critique and improvements.

```
#define sensorPin 0         // input sensor
#define txLed 2             // LED to indicate outgoing data
#define rxLed 3             // LED to indicate incoming data
#define analogLed 9         // LED that changes brightness with
                            // incoming value
#define threshold 10        // how much change you need to see on
                            // the sensor before sending
int lastSensorReading = 0;  // previous state of the switch
int inByte= -1;             // incoming byte from serial RX
char inString[6];           // string for incoming serial data
int stringPos = 0;          // string index counter

void setup() {
  // configure serial communications:
  Serial.begin(9600);

  // configure output pins:
  pinMode(txLed, OUTPUT);
  pinMode(rxLed, OUTPUT);
  pinMode (analogLed, OUTPUT);

  // set XBee's destination address:
  setDestination();
  // blink the TX LED to indicate the main program's about to start:
  blink(3);
}
void setDestination() {
  // put the radio in command mode:
  Serial.print("+++");
  // wait for the radio to respond with "OK\r"
  char thisByte = 0;
  while (thisByte != '\r') {
    if (Serial.available() > 0) {
      thisByte = Serial.read();
    }
  }
  // set the destination address, using 16-bit addressing.
  // if you're using two radios, one radio's destination
  // should be the other radio's MY address, and vice versa:
  Serial.print("ATDH0, DL5678\r");
  // set my address using 16-bit addressing:
  Serial.print("ATMY1234\r");
  // set the PAN ID. If you're working in a place where many people
  // are using XBees, you should set your own PAN ID distinct
  // from other projects.
  Serial.print("ATID1111\r");
  // put the radio in data mode:
  Serial.print("ATCN\r");
}
// Blink the tx LED:
void blink(int howManyTimes) {
  for (int i=0; i< howManyTimes; i++) {
    digitalWrite(txLed, HIGH);
    delay(200);
    digitalWrite(txLed, LOW);
    delay(200);
  }
}

void loop() {
  // listen for incoming serial data:
  if (Serial.available() > 0) {
    // turn on the RX LED whenever you're reading data:
    digitalWrite(rxLed, HIGH);
    handleSerial();
  }
  else {
    // turn off the receive LED when there's no incoming data:
    digitalWrite(rxLed, LOW);
  }
  // listen to the potentiometer:
  char sensorValue = readSensor();

  // if there's something to send, send it:
  if (sensorValue > 0) {
    //light the tx LED to say you're sending:
    digitalWrite(txLed, HIGH);
    Serial.print(sensorValue, DEC );
    Serial.print("\r");

    // turn off the tx LED:
    digitalWrite(txLed, LOW);
  }
}
void handleSerial() {
  inByte = Serial.read();
  // save only ASCII numeric characters (ASCII 0 - 9):
  if ((inByte >= '0') && (inByte <= '9')){
    inString[stringPos] = inByte;
    stringPos++;
  }
```

```
  // if you get an ASCII carriage return:
  if (inByte == '\r') {
    // convert the string to a number:
    int brightness = atoi(inString);
    // set the analog output LED:
    analogWrite(analogLed, brightness);

    // put zeroes in the array
    for (int c = 0; c < stringPos; c++) {
      inString[c] = 0;
    }
    // reset the string pointer:
    stringPos = 0;
  }
}

char readSensor() {
  char message = 0;
  // read the sensor:
  int sensorReading = analogRead(sensorPin);

  // look for a change from the last reading
  // that's greater than the threshold:
  if (abs(sensorReading - lastSensorReading) > threshold) {
    message = sensorReading/4;
    lastSensorReading = sensorReading;
  }
  return message;
}
```

BlueRadios Master Connection

Language: Wiring/Arduino

This program assumes that the microcontroller is connected to a BlueRadios bluetooth radio, and that the radio is in master mode. When the program starts, it releases the CTSpin pin of the radio, so the radio can send data to the microcontroller. Then it sends a connect message and listens. If more than 5 seconds passes, it attempts to connect again. If it receives a comma, which only appears in the CONNECT,<address> string, it assumes the radio is connected and starts sending data. If it receives an S, it assumes the radio is disconnected and stops sending.

```
#define sensorPin 0         // input sensor
#define txLed 2             // LED to indicate outgoing data
#define rxLed 3             // LED to indicate incoming data
#define CTSpin 4            // Clear-to-send pin
#define analogLed 9         // LED that will change brightness with
                            // incoming value
#define threshold 10        // how much change you need to see on the
                            // sensor before sending

byte lastSensorReading = 0;  // previous state of the pot

long lastConnectTry;         // milliseconds elapsed since the last
                             // connection attempt
long connectTimeout = 5000;  // milliseconds to wait between
                             // connection attempts
int inByte= -1;              // incoming byte from serial RX
char inString[6];            // string for incoming serial data
int stringPos = 0;           // string index counter

// address of the remote BT radio.
char remoteAddress[13] = "112233445566";
byte connected = false;      // whether you're connected or not

void setup() {
  // configure serial communications:
  Serial.begin(9600);

  // configure output pins:
  pinMode(txLed, OUTPUT);
  pinMode(rxLed, OUTPUT);
  pinMode (analogLed, OUTPUT);
  pinMode(CTSpin, OUTPUT);

  // set CTS low so BlueSMiRF can send you serial data:
  digitalWrite(CTSpin, LOW);

  // Attempt a connection:
  BTConnect();

  // blink the tx LED to say you're done with setup:
  blink(3);
}

void BTConnect() {
  Serial.print("+++\r");
  delay(250);
  Serial.print("ATDH\r");
  Serial.print("ATDM");
  Serial.print(remoteAddress);
  Serial.print(",1101\r");
}

int readSensor() {
  int message = 0;
  // read the sensor:
  int sensorReading = analogRead(sensorPin);

  // look for a change from the last reading
  // that's greater than the threshold:
  if (abs(sensorReading - lastSensorReading) > threshold) {
    message = sensorReading/4;
    lastSensorReading = sensorReading;
  }
  return message;
}
```

» Replace with the address of your remote radio

```
void blink(int howManyTimes) {
  for (int i=0; i< howManyTimes; i++) {
    digitalWrite(txLed, HIGH);
    delay(200);
    digitalWrite(txLed, LOW);
    delay(200);
  }
}

void loop() {
  if (Serial.available() > 0) {
    // signal that there's incoming data using the rx LED:
    digitalWrite(rxLed, HIGH);
    // do something with the incoming byte:
    handleSerial();
    // turn the rx LED off.
    digitalWrite(rxLed, LOW);
  }

  // if you're not connected and 5 seconds have passed in that state,
  // make an attempt to connect to the other radio:
  if (!connected && millis() - lastConnectTry > connectTimeout) {
    BTConnect();
    lastConnectTry = millis();
  }
}

void handleSerial() {
  inByte = Serial.read();
  delay(2);
  // comma comes only in the CONNECT,<address> message:
  if (inByte == ',') {
    // send an initial message:
    sendData();
    // update the connection status:
    connected = true;
  }

  //S comes only in the DISCONNECT message:
  if (inByte == 'S') {
    // turn off the analog LED:
    analogWrite(analogLed, 0);
    connected = false;
  }
  //R comes only in the NO CARRIER and NO ANSWER messages:
  if (inByte == 'R') {
    // turn off the analog LED:
    analogWrite(analogLed, 0);
    connected = false;
  }

  if (connected) {
    // save only ASCII numeric characters (ASCII 0 - 9):
    if ((inByte >= '0') && (inByte <= '9')){
      inString[stringPos] = inByte;
      stringPos++;
    }
    // if you get an asterisk, it's the end of a string:
    if (inByte == '*') {
      // convert the string to a number:
      int brightness = atoi(inString);
      // set the analog output LED:
      analogWrite(analogLed, brightness);

      // put zeroes in the array
      for (int c = 0; c < stringPos; c++) {
        inString[c] = 0;
      }
      // reset the string pointer:
      stringPos = 0;
      // Since this  byte (*) is the end of an incoming string,
      // send out your reading in response:
      sendData();
    }
  }
}

void sendData() {
  // indicate that we're sending using the tx LED:
  digitalWrite(txLed, HIGH);
  Serial.print(readSensor(), DEC);
  // string termination:
  Serial.print("*");
  // turn off the tx LED:
  digitalWrite(txLed, LOW);
}
```

Chapter 7

Lantronix UDP Device Query

Language: Processing

Sends out a UDP broadcast packet to query a subnet for Lantronix serial-to-ethernet devices. Lantronix devices are programmed to respond to UDP messages received on port 30718. If a Lantronix device receives the string 0x00 0x00 0x00 0xF6, it responds with a UDP packet containing the status message on port 30718. When the program starts, press any key on the keyboard and watch the message pane for responses. See the Lantronix integration guide from www.lantronix.com for the details. This program uses the Hypermedia UDP library available at hypermedia.loeil.org/processing/

```
// import UDP library
import hypermedia.net.*;

UDP udp;                        // define the UDP object
int queryPort = 30718;          // the port number for the device query

void setup() {
  // create a new connection to listen for
  // UDP datagrams on query port;
  udp = new UDP(this, queryPort);

  // listen for incoming packets:
  udp.listen( true );
}

void draw() {
  // twiddle your thumbs.  Everything is event-generated.
}

/*
 send the query message when any key is pressed:
 */
void keyPressed() {
  byte[] queryMsg = new byte[4];
  queryMsg[0] = 0x00;
  queryMsg[1] = 0x00;
  queryMsg[2] = 0x00;
  // because 0xF6 (decimal value 246) is greater than 128
  // you have to explicitly convert it to a byte:
  queryMsg[3] = byte(0xF6);

  // send the message
  udp.send( queryMsg, "255.255.255.255", queryPort );
  println("UDP Query sent");
}
```

```
/*
  listen for responses via UDP
 */
void receive( byte[] data, String ip, int port ) {
  String inString = new String(data);  // incoming data as a string
  int[] intData = int(data);           // data converted to ints
  int i = 0;                           // counter
  // print the result:
  println( "response from "+ip+" on port "+port );

  // parse the response for the appropriate data.
  // if the fourth byte is <F7>, we got a status reply:
  print("Received response: ");
  println(hex(intData[3],2));
  if (intData[3] == 0xF7) {
    // MAC address of the sender is bytes 24 to 30 (the end):
    print("MAC Addr: ");
    for (i=24; i < intData.length; i++) {
      print(" " + hex(intData[i], 2));
    }
  }
  // print 2 lines to separate messages from multiple responders:
  print("\n\n");
}
```

Toxic Report

Language: PHP

This program opens a socket connection to an Xport and reads bytes from the socket. It then sorts the bytes into packets, interprets the packets, reports the results, and saves them to a data log file.

```
<?php
// Global variables.
// These can be used by any of the script's functions:

global $ip, $port, $packetsToRead, $timeStamp, $messageString;

$ip = "192.168.1.236";  // IP address to connect to

$port = 10001;            // port number of IP.
$packetsToRead = 10;      // total number of packets to read
$totalAverage = 0;        // the summary of all sensor readings
$packetCounter = 0;       // counter for packets as you read them
$bytes = array();         // array for bytes when you're reading them
$packets = array();       // array to hold the arrays of bytes

// $messageString is used to return messages for printing in the HTML:
$messageString = "No Sensor Reading Taken";
```

Fill in the IP address of your Xport here

```
// Get the time and date:
$timeStamp =  $date = date("m-d-Y H:i:s");

//if a filled textbox was submitted, get the values:
if ((isset($_POST["ip"])) && (isset($_POST["port"])) &&
    (isset($_POST["packetsToRead"]))) {
        $ip = $_POST["ip"];
        $port = $_POST["port"];
        $packetsToRead = $_POST["packetsToRead"];
}

// open a socket to the Xport:
$mySocket = fsockopen ($ip, $port, $errorno, $errorstr, 30);
if (!$mySocket) {
    //if the socket didn't open, return an error message
    return "Error $errorno: $errorstr<br>";
} else {
    // if the socket exists, read packets until you've reached
    // $packetsToRead:
    while ($packetCounter < $packetsToRead) {
        // read a character from the socket connection,
        // and convert it to a numeric value using ord(),
        $char = ord(fgetc($mySocket));

        // if you got a header byte, deal with the last array
        // of bytes first:
        if ($char == 0x7E) {
            // push the last byte array onto the end
            // of the packet array:
            array_push($packets, $bytes);
            $bytes = array(); // clear the byte array.
            // increment the packet counter:
            $packetCounter++;
        }
        // push the current byte onto the end of the byte array:
        array_push($bytes, $char);
    }
    // average the readings from all the packets to get a final
    // sensor reading:
    $totalAverage = averagePackets($packets);

    // update the message for the HTML:
    $messageString =
      "Sensor Reading at:". $timeStamp . ": " . $totalAverage;

    // if you got a good reading, write it to the datalog file:
    if ($totalAverage > 0) {
        writeToFile($totalAverage);
    }
    //close the socket:
    fclose ($mySocket);
}
```

```
function averagePackets($whichArray) {
    $packetAverage = 0;        // average of all the sensor readings
    $validReadings = 0;        // number of valid readings
    $readingsTotal = 0;        // total of all readings, for averaging

  // iterate over the packet array:
    foreach ($whichArray as $thisPacket) {
        // parse each packet to get the average sensor reading:
        $thisSensorReading = parsePacket($thisPacket);

        if ($thisSensorReading > 0 && $thisSensorReading < 1023) {
            // if the sensor reading is valid, add it to the total:
            $readingsTotal = $readingsTotal + $thisSensorReading;
            // increment the total number of valid readings:
            $validReadings++;
        }
    }
    if ($validReadings > 0) {
        // round the packet average to 2 decimal points:
        $packetAverage = round($readingsTotal / $validReadings, 2);
        return $packetAverage;
    } else {
        return -1;
    }
}

function parsePacket($whichPacket) {
    $adcStart = 11;                // ADC reading starts at 12th byte
    $numSamples = $whichPacket[8]; // number of samples in the packet
    $total = 0;                    // sum of ADC readings for averaging
    // if you got all the bytes, find the average ADC reading:
    if( count($whichPacket) == 22) {
        // read the address. It's a two-byte value, so you
        // add the two bytes as follows:
        $address = $whichPacket[5] + $whichPacket[4] * 256;

        // read $numSamples 10-bit analog values, two at a time
        // because each reading is two bytes long:
        for ($i = 0; $i < $numSamples * 2;  $i=$i+2) {
            // 10-bit value = high byte * 256 + low byte:
            $thisSample = ($whichPacket[$i + $adcStart] * 256) +
              $whichPacket[($i + 1) + $adcStart];
            // add the result to the total for averaging later:
            $total = $total + $thisSample;
        }
        // average the result:
        $average = $total / $numSamples;
        return $average;
    }  else {
        return -1;
    }
}
```

```
function writeToFile($whichReading) {
    global $timeStamp, $messageString;

    // combine the reading and the timestamp:
    $logData = "$timeStamp $whichReading\n";
    $myFile = "datalog.txt";    // name of the file to write to:

    // check to see that the file exists and is writable:
    if (is_writable($myFile)) {
        // try to write to the file:
       if (!($fh = fopen($myFile, "a"))) {
            $messageString = "Couldn't open file $myFile";
        } else {
            // if you could open the file but not write to it, say so:
            if (!fwrite($fh, $logData)) {
                $messageString = "Couldn't write to $myFile";
            }
        }
    } else {
        //if it's not writeable:
       $messageString = "The file $myFile is not writable";
    }
}

?>

    <html>
    <head>
   </head>

    <body>
        <h2>
        <?=$messageString?>
        </h2>
        <hr>

        <form name="message" method="post" action="toxic_report.php">
            IP Address: <input type="text" name="ip" value="<?=$ip?>"
                        size="15" maxlength="15">
            Port: <input type="text" name="port" value="<?=$port?>"
                   size="5" maxlength="5">  <br>
            Number of readings to take: <input type="text"
              name="packetsToRead"
              value="<?=$packetsToRead?>" size="6">
            <input type="submit" value="Send It">
        </form>

    </body>
</html>
```

Lantronix UDP Tester

Language: Processing

Sends and receives UDP messages from Lantronix serial-to-ethernet devices. Sends a serial message to a Lantronix device connected to the serial port when you type "s". Sends a UDP message to the Lantronix device when you type "u". Listens for both UDP and serial messages and prints them out.

```
// import UDP library
import hypermedia.net.*;
// import serial library:
import processing.serial.*;

UDP udp;                    // define the UDP object
int queryPort = 10002;      // the port number for the device query
Serial myPort;
String xportIP = "192.168.1.20";  // fill in your Xport's IP here
int xportPort = 10001;              // the Xport's receive port
String inString = "";               // incoming serial string

void setup() {
  // create a new connection to listen for
  // UDP datagrams on query port;
  udp = new UDP(this, queryPort);
  // listen for incoming packets:
  udp.listen( true );

  println(Serial.list());
  // make sure the serial port chosen here is the one attached
  // to your Xport:
  myPort = new Serial(this, Serial.list()[0], 9600);
}

//process events
void draw() {
  background(0,0,255); // a nice blue background.
}

/*
 send messages when s or u key is pressed:
 */
void keyPressed() {
  switch (key) {
  case 'u':
    udp.send("Hello UDP!\r\n", xportIP, xportPort);
    break;
  case 's':
    String messageString = "Hello Serial!";
    for (int c = 0; c < messageString.length(); c++) {
      myPort.write(messageString.charAt(c));
    }
    break;
  }
}
```

```
/*
  listen for UDP responses
 */
void receive( byte[] data, String ip, int port ) {
  String inString = new String(data); // incoming data as a string
  println( "received "+inString +"  from "+ip+" on port "+port );

  // print a couple of blank lines to separate messages
  // from multiple responders:
  print("\n\n");
}

/*
  listen for serial responses
 */
void serialEvent(Serial myPort) {
  // read any incoming bytes from the serial port and print them:
  char inChar = char(myPort.read());

  // if you get a linefeed, the string is ended; print it:
  if (inChar == '\n') {
    println("received " + inString + "  in the serial port\r\n");
    // empty the string for the next message:
    inString = "";
  }
  else {
    inString += inChar; // add the latest byte to inString.
  }
}
```

XBee Packet Reader and Graphing Program

Language: Processing

Reads a packet from an XBee radio via UDP and parses it. Graphs the results over time. The packet should be 22 bytes long, made up of the following:

- byte 1: 0x7E, the start byte value
- byte 2-3: packet size, a 2-byte value (not used here)
- byte 4: API identifier value, a code that says what this response is (not used here)
- byte 5-6: Sender's address
- byte 7: RSSI, Received Signal Strength Indicator (not used here)
- byte 8: Broadcast options (not used here)
- byte 9: Number of samples to follow
- byte 10-11: Active channels indicator (not used here)
- byte 12-21: 5 10-bit values, each ADC samples from the sender

```
import hypermedia.net.*;
import processing.serial.*;

UDP udp;                        // define the UDP object
int queryPort = 10002;          // the port number for the device query
int hPos = 0;                   // horizontal position on the graph
int fontSize = 14;              // size of the text font

void setup() {
  // set the window size:
  size(400,300);
  // create a font with the second font available to the system:
  PFont myFont = createFont(PFont.list()[1], fontSize);
  textFont(myFont);

  // create a new connection to listen for
  // UDP datagrams on query port:
  udp = new UDP(this, queryPort);

  // listen for incoming packets:
  udp.listen( true );

  // show the initial time and date:
  background(0);
  eraseTime(hPos, 0);
  drawTime(hPos, 0);
}

void draw() {
  // nothing happens here.  It's all event-driven
  // by the receive() method.
}

/*
  listen for UDP responses
 */
void receive( byte[] data, String ip, int port ) {
  int[] inString = int(data); // incoming data converted to string
  parseData(inString);
}

/*
 Once you've got a packet, you need to extract the useful data.
 This method gets the address of the sender and the 5 ADC readings.
 It then averages the ADC readings and gives you the result.
 */
void parseData(int[] thisPacket) {
  int adcStart = 11;    // ADC reading starts at byte 12
  int numSamples = thisPacket[8];        // number of samples in packet
  int[] adcValues = new int[numSamples]; // array to hold
                                         // the 5 readings

  int total = 0;                 // sum of all the ADC readings
  int rssi = 0;                  // the received signal strength

  // read the address. It's a two-byte value, so you
  // add the two bytes as follows:
  int address = thisPacket[5] + thisPacket[4] * 256;
```

```
  // read the received signal strength:
  rssi = thisPacket[6];

  // read <numSamples> 10-bit analog values, two at a time
  // because each reading is two bytes long:
  for (int i = 0; i < numSamples * 2;  i=i+2) {
    // 10-bit value = high byte * 256 + low byte:
    int thisSample = (thisPacket[i + adcStart] * 256) +
      thisPacket[(i + 1) + adcStart];
    // put the result in one of 5 bytes:
    adcValues[i/2] = thisSample;
    // add the result to the total for averaging later:
    total = total + thisSample;
  }
  // average the result:
  int average = total / numSamples;
  // draw a line on the graph:
  drawGraph(average/4);
  eraseTime (hPos - 1, fontSize * 2);
  drawTime(hPos, fontSize * 2);
}
/*
  update the graph
 */
void drawGraph(int graphValue) {
  // draw the line:
  stroke(0,255,0);
  line(hPos, height, hPos, height - graphValue);
  // at the edge of the screen, go back to the beginning:
  if (hPos >= width) {
    hPos = 0;
    //wipe the screen:
    background(0);
    // wipe the old date and time, and draw the new:
    eraseTime(hPos, 0);
    drawTime(hPos, 0);
  }
  else {
    // increment the horizontal position to draw the next line:
    hPos++;
  }
}
/*
  Draw a black block over the previous date and time strings
 */

void eraseTime(int xPos, int yPos) {
  // use a rect to block out the previous time, rather than
  // redrawing the whole screen, which would mess up the graph:
  noStroke();
  fill(0);
  rect(xPos,yPos, 120, 80);
  // change the fill color for the text:
  fill(0,255,0);
}

/*
  print the date and the time
 */
void drawTime(int xPos, int yPos) {
  // set up an array to get the names of the months
  // from their numeric values:
  String[] months = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
    "Sep", "Oct", "Nov", "Dec"                };

  String date = "";          // string to hold the date
  String time = "";          // string to hold the time

  // format the date string:
  date += day();
  date += " ";
  date += months[month() -1];
  date += " ";
  date += year();

  // format the time string:
  time += hour();
  time += ":";
  if (minute() < 10) {
    time += "0";
    time += minute();
  }
  else {
    time +=minute();
  }
  time += ":";
  if (second() < 10) {
    time += "0";
    time += second();
  }
  else {
    time +=second();
  }

  // print both strings:
  text(date, xPos, yPos + fontSize);
  text(time, xPos, yPos + (2 * fontSize));
}
```

Chapter 8

Sharp GP2D12 IR ranger reader

Language: Wiring/Arduino

Reads the value from a Sharp GP2D12 IR ranger and sends it out serially.

```
int sensorPin = 0;     // Analog input pin
int sensorValue = 0;   // value read from the pot

void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  sensorValue = analogRead(sensorPin); // read the pot value

  // the sensor actually gives results that aren't linear.
  // this formula converts the results to a linear range.
  int range = (6787 / (sensorValue - 3)) - 4;

  Serial.println(range, DEC);    // print the sensor value
  delay(10);  // wait 10 milliseconds before the next loop
}
```

SRF02 sensor reader

Language: Wiring/Arduino

Reads data from a Devantech SRF02 ultrasonic sensor. Should also work for the SRF08 and SRF10 sensors as well.

Sensor connections:

- SDA - Analog pin 4
- SCL - Analog pin 5

```
// include Wire library to read and write I2C commands:
#include <Wire.h>

// the commands needed for the SRF sensors:
#define sensorAddress 0x70
#define readInches 0x50
// use these as alternatives if you want centimeters or microseconds:
#define readCentimeters 0x51
#define readMicroseconds 0x52
// this is the memory register in the sensor that contains the result:
#define resultRegister 0x02

void setup()
{
  Wire.begin();  // start the I2C bus.
  // open the serial port:
  Serial.begin(9600);
}

void loop()
{
  // send the command to read the result in inches:
  sendCommand(sensorAddress, readInches);
  // wait at least 70 milliseconds for a result:
  delay(70);
  // set the register that you want to read the result from:
  setRegister(sensorAddress, resultRegister);

  // read the result:
  int sensorReading = readData(sensorAddress, 2);
  // print it:
  Serial.print("distance: ");
  Serial.print(sensorReading);
  Serial.println(" inches");
  // wait before next reading:
  delay(70);
}
/*
  SendCommand() sends commands in the format that the SRF sensors
expect
 */
void sendCommand (int address, int command) {
  // start I2C transmission:
  Wire.beginTransmission(address);
  // send command:
  Wire.send(0x00);
  Wire.send(command);
  // end I2C transmission:
  Wire.endTransmission();
}
/*
  setRegister() tells the SRF sensor to change the address
  pointer position
 */
void setRegister(int address, int thisRegister) {
  // start I2C transmission:
  Wire.beginTransmission(address);
  // send address to read from:
  Wire.send(thisRegister);
  // end I2C transmission:
  Wire.endTransmission();
}
/*
readData() returns a result from the SRF sensor
 */
int readData(int address, int numBytes) {
  int result = 0;        // the result is two bytes long

  // send I2C request for data:
  Wire.requestFrom(address, numBytes);
```

```
  // wait for two bytes to return:
  while (Wire.available() < 2 )   {
    // wait for result
  }
  // read the two bytes, and combine them into one int:
  result = Wire.receive() * 256;
  result = result + Wire.receive();
  // return the result:
  return result;
}
```

XBee Signal Strength Reader

Language: Processing

Reads a packet from an XBee radio and parses it. The packet should be 22 bytes long. It should be made up of the following:

- byte 1: 0x7E, the start byte value
- byte 2-3: packet size, a 2-byte value (not used here)
- byte 4: API identifier value, a code that says what this response is (not used here)
- byte 5-6: Sender's address
- byte 7: RSSI, Received Signal Strength Indicator (not used here)
- byte 8: Broadcast options (not used here)
- byte 9: Number of samples to follow
- byte 10-11: Active channels indicator (not used here)
- byte 12-21: 5 10-bit values, each ADC samples from the sender

```
import processing.serial.*;

Serial XBee ;                   // input serial port from the XBee
Radio
int[] packet = new int[22];     // with 5 samples, the XBee packet is
                                // 22 bytes long
int byteCounter;                // keeps track of where you are in
                                // the packet
int rssi = 0;                   // received signal strength
int address = 0;                // the sending XBee 's address
Serial myPort;                  // The serial port
int fontSize = 18;              // size of the text on the screen
int lastReading = 0;            // value of the previous incoming byte

void setup () {
  size(400, 300);        // window size

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], fontSize);
  textFont(myFont);

  // get a list of the serial ports:
  println(Serial.list());

  // open the serial port attached to your XBee radio:
  XBee = new Serial(this, Serial.list()[0], 9600);
}

void draw() {
  // if you have new data and it's valid (>0), graph it:
  if ((rssi > 0 ) && (rssi != lastReading)) {
    // set the background:
    background(0);
    // set the bar height and width:
    int rectHeight = rssi;
    int rectWidth = 50;
    // draw the rect:
    stroke(23, 127, 255);
    fill (23, 127, 255);
    rect(width/2 - rectWidth, height-rectHeight, rectWidth, height);
    // write the number:
    text("XBee Radio Signal Strength test", 10, 20);
    text("From: " + hex(address), 10, 40);
    text ("RSSI: -" + rssi + " dBm", 10, 60);
    // save the current byte for next read:
    lastReading = rssi;
  }
}

void serialEvent(Serial XBee ) {
  // read a byte from the port:
  int thisByte = XBee .read();
  // if the byte = 0x7E, the value of a start byte, you have
  // a new packet:
  if (thisByte == 0x7E) {   // start byte
    // parse the previous packet if there's data:
    if (packet[2] > 0) {
      parseData(packet);
    }
    // reset the byte counter:
    byteCounter = 0;
  }
  // put the current byte into the packet at the current position:
  packet[byteCounter] = thisByte;
  //  increment the byte counter:
  byteCounter++;
}

/*
 Once you've got a packet, you need to extract the useful data.
 This method gets the address of the sender and RSSI.
 */
void parseData(int[] thisPacket) {
  // read the address. It's a two-byte value, so you
  // add the two bytes as follows:
  address = thisPacket[5] + thisPacket[4] * 256;
  // get RSSI:
  rssi = thisPacket[6];
}
```

GPS parser

Language: Processing

This program takes in NMEA 0183 serial data and parses out the date, time, latitude, and longitude using the GPRMC sentence.

```
// import the serial library:
import processing.serial.*;

Serial myPort;           // The serial port
float latitude = 0.0;    // the latitude reading in degrees
String northSouth;       // north or south?
float longitude = 0.0;   // the longitude reading in degrees
String eastWest;         // east or west?
float heading = 0.0;     // the heading in degrees

int hrs, mins, secs;      // time units
int thisDay,  thisMonth, thisYear;

void setup() {
  size(300, 300);        // window size

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 14);
  textFont(myFont);

  // settings for drawing arrow:
  noStroke();
  smooth();

  // List all the available serial ports
  println(Serial.list());

  // I know that the first port in the serial list on my mac
  // is always my  Keyspan adaptor, so I open Serial.list()[0].
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 4800);

  // read bytes into a buffer until you get a carriage
  // return (ASCII 13):
  myPort.bufferUntil('\r');
}

void draw() {
  background(0);
  // make the text white:
  fill(255);

  // print the date and time from the GPS sentence:
  text(thisMonth+ "/"+ thisDay+ "/"+ thisYear , 50, 30);
  text(hrs+ ":"+ mins+ ":"+ secs + " GMT  ", 50, 50);
  // print the position from the GPS sentence:
  text(latitude + " " + northSouth + ", " +longitude +" "+ eastWest,
       50, 70);
  text("heading " + heading + " degrees", 50,90);

  // draw an arrow using the heading:
  drawArrow(heading);
}


void serialEvent(Serial myPort) {
  // read the serial buffer:
  String myString = myPort.readStringUntil('\n');

  // if you got any bytes other than the linefeed, parse it:
  if (myString != null) {
    parseString(myString);
  }
}

void parseString (String serialString) {
  // split the string at the commas:
  String items[] = (split(serialString, ','));

  // if the first item in the sentence is the
  // identifier, parse the rest
  if (items[0].equals("$GPRMC")) {
    // get time, date, position, course, and speed
    getRMC(items);
  }
}


void getRMC(String[] data) {
  // move the items from the string into the variables:
  int time = int(data[1]);
  // first two digits of the time are hours:
  hrs = time/10000;
  // second two digits of the time are minutes:
  mins = (time%10000)/100;
  // last two digits of the time are seconds:
  secs = (time%100);

  // if you have a valid reading, parse the rest of it:
  if (data[2].equals("A")) {
    latitude = float(data[3])/100.0;
    northSouth = data[4];
    longitude = float(data[5])/100.0;
    eastWest = data[6];
    heading = float(data[8]);
    int date = int(data[9]);
    // last two digits of the date are year.  Add the century too:
    thisYear = date%100 + 2000;
    // second two digits of the date are month:
    thisMonth =  (date%10000)/100;
    // first two digits of the date are day:
    thisDay = date/10000;
  }
}
```

```
void drawArrow(float angle) {
  // move whatever you draw next so that (0,0) is centered
  // on the screen:
  translate(width/2, height/2);

  // draw a circle in light blue:
  fill(80,200,230);
  ellipse(0,0,50,50);
  // make the arrow black:
  fill(0);
  // rotate using the heading:
  rotate(radians(angle));

  // draw the arrow.  center of the arrow is at (0,0):
  triangle(-10, 0, 0, -20, 10, 0);
  rect(-2,0, 4,20);
}
```

CMPS03 compass reader

Language: Wiring/Arduino

Reads data from a Devantech CMPS03 compass sensor.

Sensor connections:

- SDA - Analog pin 4
- SCL - Analog pin 5

```
// include Wire library to read and write I2C commands:
#include <Wire.h>

// the commands needed for the SRF sensors:
#define sensorAddress 0x60
// this is the memory register in the sensor that contains the result:
#define resultRegister 0x02


void setup() {
  // start the I2C bus
  Wire.begin();
  // open the serial port:
  Serial.begin(9600);
}

void loop() {
  // send the command to read the result in inches:
  setRegister(sensorAddress, resultRegister);
  // read the result:
  int bearing = readData(sensorAddress, 2);
  // print it:
  Serial.print("bearing: ");
  Serial.print(bearing/10);
  Serial.println(" degrees");
  // wait before next reading:
  delay(70);
}
```

```
/*
  setRegister() tells the SRF sensor to change the address
  pointer position
 */
void setRegister(int address, int thisRegister) {
  // start I2C transmission:
  Wire.beginTransmission(address);
  // send address to read from:
  Wire.send(thisRegister);
  // end I2C transmission:
  Wire.endTransmission();
}
/*
 readData() returns a result from the SRF sensor
 */

int readData(int address, int numBytes) {
  int result = 0;        // the result is two bytes long

  // send I2C request for data:
  Wire.requestFrom(address, numBytes);
  // wait for two bytes to return:
  while (Wire.available() < 2 )   {
    // wait for result
  }
  // read the two bytes, and combine them into one int:
  result = Wire.receive() * 256;
  result = result + Wire.receive();
  // return the result:
  return result;
}
```

Accelerometer reader

Language: Wiring/Arduino

Reads 2 axes of an accelerometer and sends the values out the serial port

```
int accelerometer[2];     // variable to hold the accelerometer values

void setup() {
  // open serial port:
  Serial.begin(9600);
  // send out some initial data:
  Serial.println("0,0,");
}

void loop() {
  // read 2 channels of the accelerometer:
  for (int i = 0; i < 2; i++) {
    accelerometer[i] = analogRead(i);
    // delay to allow analog-to-digital converter to settle:
    delay(10);
  }
```

```
  // if there's serial data in, print sensor values out:
  if (Serial.available() > 0) {
    // read incoming data to clear serial input buffer:
    int inByte = Serial.read();
    for (int i = 0; i <  2; i++) {
      // values as ASCII strings:
      Serial.print(accelerometer[i], DEC);
      // print commas in between values:
      Serial.print(",");
    }
    // print \r and \n after values are sent:
    Serial.println();
  }
}
```

Accelerometer Tilt

Language: Processing

Takes the values in serially from an accelerometer attached to a microcontroller and uses them to set the attitude of a disk on the screen.

```
import processing.serial.*;      // import the serial lib

int graphPosition = 0;           // horizontal position of the graph
int[] vals = new int[2];         // raw values from the sensor
int[] maximum = new int[2];      // maximum value sensed
int[] minimum = new int[2];      // minimum value sensed
int[] range = new int[2];        // total range sensed
float[] attitude = new float[2]; // the tilt values
float position;                  // position to translate to
Serial myPort;                   // the serial port
boolean madeContact = false;     // whether there's been serial data in

void setup () {
  // draw the window:
  size(400, 400, P3D);
  // set the background color:
  background(0);
  // set the maximum and minimum values:
  for (int i = 0; i < 2; i++) {
    maximum[i] = 600;
    minimum[i] = 200;
    // calculate the total current range:
    range[i] = maximum[i] - minimum[i];
  }
  position = width/2; // calculate position.

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 18);
  textFont(myFont);
  // List all the available serial ports
  println(Serial.list());
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);
  // only generate a serial event when you get a return char:
  myPort.bufferUntil('\r');

  // set the fill color:
  fill(90,250,250);
}

void draw () {
  // clear the screen:
  background(0);

  // print the values:
  text(vals[0] + " " + vals[1], -30, 10);

  // if you've never gotten a string from the microcontroller,
  // keep sending carriage returns to prompt for one:
  if (madeContact == false) {
    myPort.write('\r');
  }

  // set the attitude:
  setAttitude();
  // draw the plane:
  tilt();
}
void setAttitude() {
  for (int i = 0; i < 2; i++) {
    // calculate the current attitude as a percentage of 2*PI,
    // based on the current range:
    attitude[i] = (2*PI) * float(vals[i] -
      minimum[i]) /float(range[i]);
  }
}

void tilt() {
  // translate from origin to center:
  translate(position, position, position);

  // X is front-to-back:
  rotateX(-attitude[1]);
  // Y is left-to-right:
  rotateY(-attitude[0] - PI/2);

  // set the fill color:
  fill(90,250,250);
  // draw the rect:
  ellipse(0, 0, width/4, width/4);
  // change the fill color:
  fill(0);
  // Draw some text so you can tell front from back:
  // print the values:
  text(vals[0] + " " + vals[1], -30, 10,1);
}
```

```
// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():
void serialEvent(Serial myPort) {
  // if serialEvent occurs at all, contact with the microcontroller
  // has been made:
  madeContact = true;
  // read the serial buffer:
  String myString = myPort.readStringUntil('\n');

  // if you got any bytes other than the linefeed:
  if (myString != null) {
    myString = trim(myString);
    // split the string at the commas
    //and convert the sections into integers:
    int sensors[] = int(split(myString, ','));
    // if you received all the sensor strings, use them:
    if (sensors.length >= 2) {
      vals[0] = sensors[0];
      vals[1] = sensors[1];

      // send out the serial port to ask for data:
      myPort.write('\r');
    }
  }
}
```

Chapter 9

Color Recognition Using a Webcam

Language: Processing

Reads an image from a camera and looks for a blob of a particular color. Click on a color in the image to choose the color to track.

```
import processing.serial.*;      // import the serial lib

int graphPosition = 0;           // horizontal position of the graph
int[] vals = new int[2];         // raw values from the sensor
int[] maximum = new int[2];      // maximum value sensed
int[] minimum = new int[2];      // minimum value sensed
int[] range = new int[2];        // total range sensed
float[] attitude = new float[2]; // the tilt values
float position;                  // position to translate to

Serial myPort;                   // the serial port
boolean madeContact = false;     // whether there's been serial data in

void setup () {
  // draw the window:
  size(400, 400, P3D);
  // set the background color:
  background(0);
  // set the maximum and minimum values:
  for (int i = 0; i < 2; i++) {
    maximum[i] = 600;
    minimum[i] = 200;
    // calculate the total current range:
    range[i] = maximum[i] - minimum[i];
  }
  // calculate position:
  position = width/2;

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 18);
  textFont(myFont);

  // List all the available serial ports
  println(Serial.list());
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);
  // only generate a serial event when you get a return char:
  myPort.bufferUntil('\r');

  fill(90,250,250); // set the fill color.
}

void draw () {
  // clear the screen:
  background(0);
```

```
  // print the values:
  text(vals[0] + " " + vals[1], -30, 10);

  // if you've never gotten a string from the microcontroller,
  // keep sending carriage returns to prompt for one:
  if (madeContact == false) {
    myPort.write('\r');
  }

  // set the attitude:
  setAttitude();
  // draw the plane:
  tilt();
}
void setAttitude() {
  for (int i = 0; i < 2; i++) {
    // calculate the current attitude as a percentage of 2*PI,
    // based on the current range:
    attitude[i] = (2*PI) * float(vals[i] -
      minimum[i]) /float(range[i]);
  }
}

void tilt() {
  // translate from origin to center:
  translate(position, position, position);

  // X is front-to-back:
  rotateX(-attitude[1]);
  // Y is left-to-right:
  rotateY(-attitude[0] - PI/2);

  // set the fill color:
  fill(90,250,250);
  // draw the rect:
  ellipse(0, 0, width/4, width/4);
  // change the fill color:
  fill(0);
  // Draw some text so you can tell front from back:
  // print the values:
  text(vals[0] + " " + vals[1], -30, 10,1);
}

// serialEvent  method is run automatically by the Processing applet
// whenever the buffer reaches the  byte value set in the bufferUntil()
// method in the setup():
void serialEvent(Serial myPort) {
  // if serialEvent occurs at all, contact with the microcontroller
  // has been made:
  madeContact = true;
  // read the serial buffer:
  String myString = myPort.readStringUntil('\n');
  // if you got any bytes other than the linefeed:
  if (myString != null) {
    myString = trim(myString);
    // split the string at the commas
    //and convert the sections into integers:
    int sensors[] = int(split(myString, ','));
    // if you received all the sensor strings, use them:
    if (sensors.length >= 2) {
      vals[0] = sensors[0];
      vals[1] = sensors[1];

      // send out the serial port to ask for data:
      myPort.write('\r');
    }
  }
}
```

QRcode 2D Barcode Reader

Language: Processing

Uses the qrcode library from www.shiffman.net/p5/pqrcode based on a Java library from qrcode.sourceforge.jp. To use this, generate images from a QRcode generator such as qrcode.kaywa.com and put them in this sketch's data folder. Press spacebar to read from the camera, generate an image, and scan for barcodes. Press `f` to read from a file and scan. Press `s` for camera settings.
— by Tom Igoe / Daniel Shiffman

```
import processing.video.*;
import pqrcode.*;

Capture video;                              // Video capture object
String statusMsg = "Waiting for an image";  // a string for messages:

// Decoder object from pqrdecoder library
Decoder decoder;

// make sure to generate your own image here:
String testImageName = "qrcode.png";

void setup() {
  size(400, 320);
  video = new Capture(this, width, height-20, 30);
  // Create a decoder object
  decoder = new Decoder(this);

  // Create a font with a font available to the system:
  PFont myFont = createFont(PFont.list()[2], 14);
  textFont(myFont);
}
```

```
void draw() {
  background(0);

  // Display video
  image(video, 0, 0);
  // Display status
  text(statusMsg, 10, height-4);

  // If we are currently decoding
  if (decoder.decoding()) {
    // Display the image being decoded
    PImage show = decoder.getImage();
    image(show,0,0,show.width/4,show.height/4);
    statusMsg = "Decoding image";
    // fancy code for drawing dots as a progress bar:
    for (int i = 0; i < (frameCount/2) % 10; i++)
    {
      statusMsg += ".";
    }
  }
}

void captureEvent(Capture video) {
  video.read();
}

void keyReleased() {
  String code = "";
  // Depending on which key is hit, do different things:
  switch (key) {
  case ' ':        // Spacebar takes a picture and tests it:
    // copy it to the PImage savedFrame:
    PImage savedFrame = createImage(video.width,video.height,RGB);
    savedFrame.copy(video, 0,0,video.width,video.height,0,0,
      video.width,video.height);
    savedFrame.updatePixels();
    // Decode savedFrame
    decoder.decodeImage(savedFrame);
    break;
  case 'f':    // f runs a test on a file
    PImage preservedFrame = loadImage(testImageName);
    // Decode file
    decoder.decodeImage(preservedFrame);
    break;
  case 's':      // s opens the settings for this capture device:
    video.settings();
    break;
  }
}

// When the decoder object finishes
// this method will be invoked.
void decoderEvent(Decoder decoder) {
  statusMsg = decoder.getDecodedString();
}
```

Parallax RFID Reader

Language: Processing

Reads data serially from a Parallax RFID reader.

```
// import the serial library:
import processing.serial.*;

Serial myPort;      // the serial port you're using
String tagID = "";  // the string for the tag ID

void setup() {
  size(600,200);
  // list all the serial ports:
  println(Serial.list());

  // based on the list of serial ports printed from the
  // previous command, change the 0 to your port's number:
  String portnum = Serial.list()[0];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 2400);
  // incoming string from reader will have 12 bytes:
  myPort.buffer(12);

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 24);
  textFont(myFont);
}

void draw() {
  // clear the screen:
  background(0);
  // print the string to the screen:
  text(tagID, width/4, height/2 - 24);
}

/*
 this method reads bytes from the serial port
 and puts them into the tag string.
 It trims off the \r and \n
 */
void serialEvent(Serial myPort) {
  tagID = trim(myPort.readString());
}
```

ID Innovations RFID Reader

Language: Processing

Reads data serially from an ID Innovations ID12 RFID reader.

```
// import the serial library:
import processing.serial.*;

Serial myPort;      // the serial port you're using
```

```
String tagID = "";  // the string for the tag ID

void setup() {
  size(600,200);
  // list all the serial ports:
  println(Serial.list());

  // based on the list of serial ports printed from the
  // previous command, change the 0 to your port's number:
  String portnum = Serial.list()[0];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 9600);
  // incoming string from reader will have 16 bytes:
  myPort.buffer(16);

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 24);
  textFont(myFont);
}

void draw() {
  // clear the screen:
  background(0);
  // print the string to the screen:
  text(tagID, width/4, height/2 - 24);
}

/*
 this method reads bytes from the serial port
 and puts them into the tag string
 */
void serialEvent(Serial myPort) {
  // get the serial input buffer in a string:
  String inputString = myPort.readString();
  // filter out the tag ID from the string:
  tagID = parseString(inputString);
}

/*
  This method reads a string and looks for the 10-byte
 tag ID. It assumes it should get a STX byte (0x02)
 at the beginning and an ETX byte (0x03) at the end
 */
String parseString(String thisString) {
  String tagString = "";    // string to put the tag ID into

  // first character of the input:
  char firstChar = thisString.charAt(0);
  // last character of the input:
  char lastChar = thisString.charAt(thisString.length() -1);

  // if the first char is STX (0x02) and the last char
  // is ETX (0x03), then put the next ten bytes
  // into the tag string:
  if ((firstChar == 0x02) && (lastChar == 0x03)) {
    tagString = thisString.substring(1, 11);
  }
  return tagString;
}
```

ASPX RW-210 RFID Reader

Language: Processing

Reads data serially from an ASPX RW-210 RFID RFID reader.

```
// import the serial library:
import processing.serial.*;

Serial myPort;      // the serial port you're using
String tagID = "";  // the string for the tag ID

void setup() {
  size(600,200);
  // list all the serial ports:
  println(Serial.list());

  // based on the list of serial ports printed from the
  //previous command, change the 0 to your port's number:
  String portnum = Serial.list()[0];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 19200);
  // incoming string from reader will have 12 bytes:
  myPort.buffer(12);

  // create a font with the third font available to the system:
  PFont myFont = createFont(PFont.list()[2], 24);
  textFont(myFont);

  // send the continual read command:
  myPort.write(0xFB);
}

void draw() {
  // clear the screen:
  background(0);
  // print the string to the screen:
  text(tagID, width/8, height/2 - 24);
}

/*
  this method reads bytes from the serial port
 and puts them into the tag string
 */
 void serialEvent(Serial myPort) {
  int thisByte = 0;
  tagID = "";
  while(myPort.available() > 0) {
    int newByte = myPort.read();
```

```
    tagID += hex(newByte, 2);
    tagID += " ";
  }
}
```

Microcontroller RFID Reader

Language: Wiring/Arduino

Reads data serially from a Parallax or ID Innovations ID12 RFID reader.

```
#define tagLength 10    // each tag ID contains 10 bytes
#define startByte 0x0A  // for the ID Innovations reader, use 0x02
#define endByte 0x0D    // for the ID Innovations reader, use 0x03
#define dataRate 2400   // for the ID Innovations reader, use 9600


char tagID[tagLength];            // array to hold the tag you read
int tagIndex = 0;                 // counter for number of bytes read
int tagComplete = false;          // whether the whole tag's been read

void setup() {
  // begin serial:
  Serial.begin(dataRate);
}

void loop() {
  // read in and parse serial data:
  if (Serial.available() > 0) {
    readByte();
  }
  if(tagComplete == true) {
    Serial.println(tagID);
  }
}

/*
  This method reads the bytes, and puts the
 appropriate ones in the tagID
 */
void readByte() {
  char thisChar = Serial.read();
Serial.print(thisChar, HEX);
  switch (thisChar) {
  case startByte:     // start character
    // reset the tag index counter
    tagIndex = 0;
    break;
  case endByte:    // end character
    tagComplete = true;  // you have the whole tag
    break;
  default: // any other character
    tagComplete = false;  // there are still more bytes to read
    // add the byte to the tagID
    if (tagIndex < tagLength) {
      tagID[tagIndex] = thisChar;
      // increment the tag byte counter
      tagIndex++;
    }
    break;
  }
}
```

X10 test

Language: Arduino

Sends out basic X10 messages from an Arduino module using a PL513 ot TW523 X10 module.

```
// include the X10 library files:
#include <x10.h>
#include <x10constants.h>

#define zcPin 9         // the zero crossing detect pin
#define dataPin 8       // the X10 data out pin
#define repeatTimes 1   // how many times to repeat each X10 message
                        // in an electrically noisy environment, you
                        // can set this higher.

// set up a new x10  library instance:
x10 myHouse =  x10(zcPin, dataPin);

void setup() {
  // Turn off all lights:
  myHouse.write(A, ALL_UNITS_OFF,repeatTimes);
}


void loop() {
        // Turn on first module:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
        delay(500);
        // turn on second module:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        delay(500);
}
```

RFID–to-X10 translator

Language: Arduino

Reads RFID tags and sends X10 messages in response to the tags.

```
// include the X10 library files:
#include <x10.h>
#include <x10constants.h>

#define zcPin 9         // the zero crossing detect pin
#define dataPin 8       // the X10 data out pin
#define repeatTimes 1   // how many times to repeat each X10 message
                        // in an electrically noisy environment, you
                        // can set this higher.

#define tagLength 10    // each tag ID contains 10 bytes
#define startByte 0x0A  // for the ID Innovations reader, use 0x02
#define endByte 0x0D    // for the ID Innovations reader, use 0x03
#define dataRate 2400   // for the ID Innovations reader, use 9600

// set up a new x10  library instance:
x10 myHouse =  x10(zcPin, dataPin);

char tagID[tagLength];          // array to hold the tag you read
int tagIndex = 0;               // counter for number of bytes read
int tagComplete = false;        // whether the whole tag's been read
char tagOne[] = "0415AB6FB7"; // put the values for your tags here
char tagTwo[] = "0415AB5DAF";
char lastTag = 0;               // value of the last tag read

void setup() {
  Serial.begin(dataRate); // begin serial.
  // Turn off all lights:
  myHouse.write(A, ALL_LIGHTS_OFF,repeatTimes);
}
void loop() {
  // read in and parse serial data:
  if (Serial.available() > 0) {
    readByte();
  }
  // if you've got a complete tag, compare your tag
  // to the existing values:
  if (tagComplete == true) {
    if (compareTags(tagID, tagOne) == true) {
      if (lastTag != 1) {
        // if the last tag wasn't this one,
        // send commands:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
        // note that this was the last tag read:
        lastTag = 1;
      }
    }
    if (compareTags(tagID, tagTwo) == true) {
      if (lastTag != 2) {
        // if the last tag wasn't this one,
        // send commands:
        myHouse.write(A, UNIT_1,repeatTimes);
        myHouse.write(A, OFF,repeatTimes);
        myHouse.write(A, UNIT_2,repeatTimes);
        myHouse.write(A, ON,repeatTimes);
        // note that this was the last tag read:
        lastTag = 2;
      }
    }
  }
}

/*
  This method compares two char arrays byte by byte:
 */
char compareTags(char* thisTag, char* thatTag) {
  char match = true;  // whether they're the same
    for (int i = 0; i < tagLength; i++) {
    // if any two bytes don't match, the whole thing fails:
    if (thisTag[i] != thatTag[i]) {
      match = false;
    }
  }
  return match;
}

/*
  This method reads the bytes, and puts the
 appropriate ones in the tagID
 */
void readByte() {
  char thisChar = Serial.read();

  switch (thisChar) {
  case startByte:     // start character
    // reset the tag index counter
    tagIndex = 0;
    break;
  case endByte:    // end character
    tagComplete = true;  // you have the whole tag
    break;
  default:    // any other character
    tagComplete = false;  // there are still more bytes to read
    // add the byte to the tagID
    if (tagIndex < tagLength) {
      tagID[tagIndex] = thisChar;
      // increment the tag byte counter
      tagIndex++;
    }
    break;
  }
}
```

HTTP Environment Variable Printer

Language: PHP

Prints out the HTTP environment variables.

```
<?php
foreach ($_REQUEST as $key => $value)
   {
       echo "$key: $value<br>\n";
   }
   foreach ($_SERVER as $key => $value)
   {
       echo "$key: $value<br>\n";
   }
?>
```

IP geocoder

Language: PHP

Uses a client's IP address to get a latitude and longitude. Uses the client's user agent to format the response.

```
<?php
// initialize variables:
    $lat = 0;
    $long = 0;
    $ipAddress = "0.0.0.0";
    $country = "unknown";



    // Check to see what type of client this is:
    $userAgent  = getenv('HTTP_USER_AGENT');
    // Get the client's IP address:
    $ipAddress = getenv('REMOTE_ADDR');

    // use http://www.hostIP.info to get the latitude and longitude
    // from the IP address.  First, format the HTTP request string:
    $IpLocatorUrl =
      "http://api.hostip.info/get_html.php?&position=true&ip=";
    $IpLocatorUrl = $IpLocatorUrl.$ipAddress;

    // make the HTTP request:
    $filePath = fopen ($IpLocatorUrl, "r");

    // as long as you haven't reached the end of the incoming text:
    while (!feof($filePath)) {
        // read one line at a time, strip all HTML tags from the line:
        $line = fgetss($filePath, 4096);
        // break each line into fragments at the colon:
        $fragments = explode(":", $line);
        switch ($fragments[0]) {
            // if the first fragment is "country", the second
            // is the country name:
            case "Country":
                // trim any whitespace:
                $country = trim($fragments[1]);
                break;
            // if the first fragment is "Latitude", the second
            // is the latitude:
            case "Latitude":
                // trim any whitespace:
                $lat = trim($fragments[1]);
                break;
            // if the first fragment is "Longitude", the second
            //is the longitude:
            case "Longitude":
                // trim any whitespace:
                $long = trim($fragments[1]);
                break;
        }
    }
    // close the connection:
    fclose($filePath);

    // decide on the output based on the client type:
    switch ($userAgent) {
        case "lantronix":
            // Lantronix device wants a nice short answer:
            echo "<$lat,$long,$country>\n";
            break;
        case "processing":
            // Processing does well with lines:
            echo "Latitude:$lat\nLongitude:$long\nCountry:$country\n\
n";
            break;
        default:
            // other clients can take a long answer:
            echo <<<END
<html>
<head></head>

<body>
    <h2>Where You Are:</h2>
        Your country: $country<br>
        Your IP: $ipAddress<br>
        Latitude: $lat<br>
        Longitude: $long<br>
</body>
</html>
END;
    }
?>
```

Mail Reader

Language: PHP

Reads email in PHP. Requires the **pwds.php** file listed below.

```
<?php
include('POP3.php');

// keep your personal info in a separate file:
@include_once("pwds.php");

// New instance of the Net_POP3 class:
$pop3 =& new Net_POP3();

// Connect to the mail server:
$pop3->connect($host , $port);

// Send login info:
$pop3->login($user , $pass , 'APOP');

// Get a count of the number of new messages waiting:
$numMsgs = $pop3->numMsg();

echo "<pre>\n";
echo "Checking mail...\n";
echo "Number of messages: $numMsgs\n";

// Get the headers of the first message:
echo($pop3->getRawHeaders(1));
echo "\n\n\n";
echo "</pre>\n";

// disconnect:
$pop3->disconnect();
?>
```

pwds.php. This will contain your username and password info. You want to keep it separate from the main PHP file so you can protect it.

```
<?php
$user='username';        // your mail login
$pass='password';        // exactly as you normally type it
$host='pop.example.com'; // usually pop.yourmailserver.com
$port="110";             // this won't work on gmail.com and
                         // other servers using SSL
?>
```

RFID mail reader

Language: PHP

Parses a POP email box for a specific message from an Xport. The message looks like this:

From: myAccountName@myMailhost.com
Subject: Notification: Tag one
Date: June 21, 2007 6:11:59 PM EDT
To: myAccountName@myMailhost.com

```
<?php
include('POP3.php');

// keep your personal info in a separate file:
@include_once("pwds.php");

echo "Checking mail...";

// New instance of the Net_POP3 class:
$pop3 =& new Net_POP3();

// Connect to the mail server:
$pop3->connect($host , $port);

// Send login info:
$pop3->login($user , $pass , 'APOP');

// Get a count of the number of new messages waiting:
$numMsgs = $pop3->numMsg();

echo "<pre>\n";
echo "Number of messages: $numMsgs\n";

// iterate over the messages:
for ($thisMsg = 1; $thisMsg <= $numMsgs; $thisMsg++) {
    // parse the headers for each message into
    // an array called $header:
    $header = $pop3->getParsedHeaders($thisMsg);

    // print the subject header:
    $subject = $header["Subject"];
    // look for the word "Notification" before a colon
    // in the subject:
    $words = explode(":", $subject);

    // only do the rest if this mail message is a notification:
    if ($words[0] == "Notification"){
        // get the second half of the subject; that's the tag ID:
        $idTag = $words[1];
        // print it;
        echo "$idTag showed up at address\t";
```

```
        /*
            the IP address is buried in the "Received" header.
            That header is an array. The second element contains
            who it's from.  In that string, the IP is the first
            thing contained in square brackets. So:
        */

        // get the stuff in the right array element after the
        // opening square bracket:
        $receivedString = explode("[", $header["Received"][1]);
        // throw away the stuff after the closing bracket:
        $recdString2 = explode("]", $receivedString[1]);
        // what's left is the IP address:
        $ipAddress = $recdString2[0];

        // print the IP address:
        echo "$ipAddress at \t";

        // print the date header:
        $date = $header["Date"];
        echo "$date\t";
        echo "\n";
    }

}

echo "That's all folks";
echo "</pre>";

// disconnect:
$pop3->disconnect();
?>
```

Index

T

U

V

Hardware/General

Make: PROJECTS

Making Things Talk

Building electronic projects that interact with the physical world is good fun. But when devices that you've built start to talk to each other, things really start to get interesting. Making Things Talk demonstrates that once you figure out how objects communicate — whether they're microcontroller-powered devices, email programs, or networked databases — you can get them to interact.

Microcontrollers, personal computers, and web servers talking to each other.

This book is perfect for people with little technical training but a lot of interest. Maybe you're a science teacher who wants to show students how to monitor weather conditions at several locations at once, or a sculptor who wants to stage a room of choreographed mechanical sculptures.

Whether you need to plug some sensors in your home to the Internet or create a device that can interact wirelessly with other creations, Making Things Talk explains exactly what you need.

Through twenty-six simple projects, Making Things Talk shows how to get your creations to talk with one another by forming networks of smart devices that carry on conversations with you and your environment. Here are just a few of the projects:

Blink
Your very first program.

Monski pong
Control a video game with a fluffy pink monkey.

Networked Air Quality Meter
Download and display the latest report for your city.

XBee Toxic Sensor
Use ZigBee, sensors, and a cymbal monkey to warn of toxic vapors.

Bluetooth GPS
Build a battery-powered GPS that reports its location over Bluetooth.

RFID Reader Bowl
Turn your lights off when you leave the home or office.

You will:

- **Make your pet's bed send you email.**
- **Make your own game controllers that communicate over a network.**
- **Use ZigBee, Bluetooth, Infrared, and plain old radio to transmit sensor data wirelessly.**
- **Work with three easy-to-program, open source environments: Arduino/Wiring, Processing, and PHP.**
- **Write programs to send data across the Internet based on physical activity in your home, office, or backyard.**

Tom Igoe teaches courses in physical computing and networking at the Interactive Telecommunications Program in the Tisch School of the Arts at New York University. In his teaching and research, he explores ways to allow digital technologies to sense and respond to a wider range of human physical expression. He co-authored *Physical Computing: Sensing and Controlling the Physical World with Computers* with Dan O'Sullivan, which has been adopted by numerous digital art and design schools around the world. He is a contributor to MAKE magazine and a collaborator on the Arduino open source micro-controller project. He hopes someday to work with monkeys, as well.

Make:
makezine.com

US $29.99 CAN $35.99
ISBN–10: 0-596-51051-9
ISBN–13: 978-0-596-51051-0

www.oreilly.com
O'REILLY®