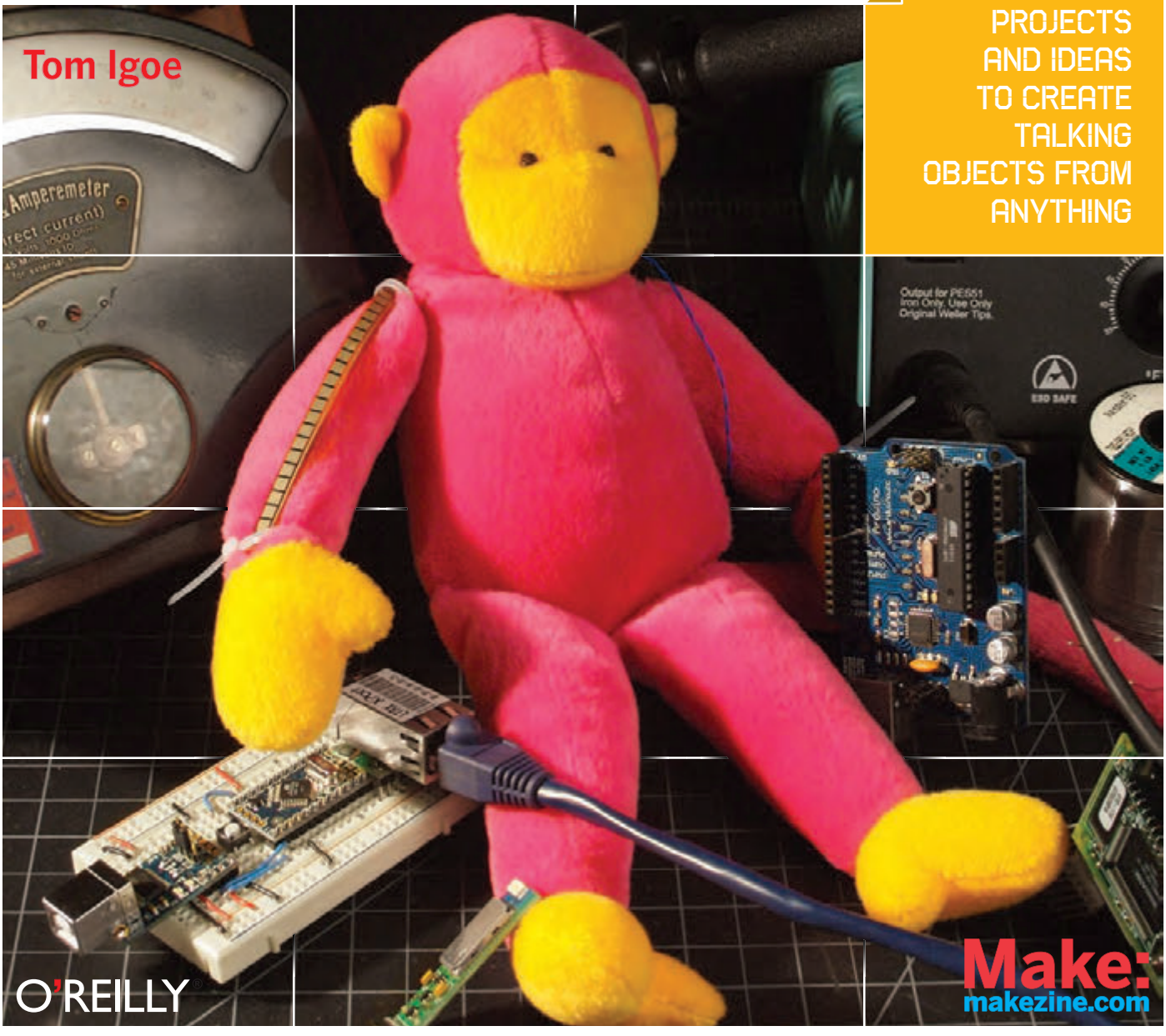
Make: PROJECTS

Making Things Talk

Practical Methods for Connecting Physical Objects

Tom Igoe

PROJECTS AND IDEAS TO CREATE TALKING OBJECTS FROM ANYTHING



O'REILLY

Make: makezine.com

Making Things Talk

First Edition

Tom Igoe

O'REILLY®

BEIJING • CAMBRIDGE • FARNHAM • KÖLN • PARIS • SEBASTOPOL • TAIPEI • TOKYO

Making Things Talk

by Tom Igoe

Published by Make:Books, an imprint of Maker Media, a division of O'Reilly Media, Inc.
1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use.
For more information, contact our corporate/institutional sales department:
800-998-9938 or *corporate@oreilly.com*.

Print History
September 2007
First Edition

Publisher: Dale Dougherty
Associate Publisher and Executive Editor: Dan Woods
Editor: Brian Jepson
Copy Editor: Nancy Kotary
Creative Director: Daniel Carter
Designer: Katie Wilson
Production Manager: Terry Bronson
Indexer: Patti Schiendelman
Cover Photograph: Tom Igoe

ISBN-10: 0-596-51051-9
ISBN-13: 978-0-596-51051-0

Contents

Preface

A few years ago, Neil Gershenfeld wrote a smart book called *When Things Start to Think*. In it, he discussed a world in which everyday objects and devices are endowed with computational power: in other words, today. He talked about the implications of devices that exchange information about our identities, abilities, and actions. It's a good read, but I think he got the title wrong. I would have called it *When Things Start to Gossip*. Because let's face it, even the most exciting thoughts are worthwhile only once you start to talk to someone else about them. This is a book about learning to make things that have computational power talk to each other, and about giving people the ability to use those things to communicate with each other.

For a couple of decades now, computer scientists have used the term object-oriented programming to refer to a style of software development in which programs and subprograms are thought of as objects. Like physical objects, they have properties and behaviors. They inherit these properties from the prototypes from which they descend. The canonical form of any object in software is the code that describes its type. Software objects make it easy to recombine objects in novel ways. You can reuse a software object, if you know its interface, the collection of properties and methods that its creator allows you access to (and documents, so that you know how to use them). It doesn't matter how a software object does what it does, as long as it does it consistently. Software objects are most effective when they're easy to understand and when they work well with other objects.

In the physical world, we're surrounded by all kinds of electronic objects: clock radios, toasters, mobile phones, music players, children's toys, and more. It can take a lot of work and a significant amount of knowledge to make a useful electronic gadget. It can take almost as much knowledge to make those gadgets talk to each other in useful ways. But that doesn't have to be the case. Electronic devices can be — and often are — built up from modules with simple, easy-to-understand interfaces. As long as you understand the interfaces, you can make anything from them. Think of it as object-oriented hardware. Understanding the ways in which things talk to each other is central to making this work. It doesn't matter whether the object is a toaster, an email program on your laptop, or a networked database. All of these objects can be connected if you can figure out how they communicate. This book is a guide to some of the tools for making those connections.

x

Who This Book Is For

This book is written for people who want to make things talk to other things. Maybe you're a science teacher who wants to show your students how to monitor weather conditions at several locations around your school district simultaneously, or a sculptor who wants to make a whole room of choreographed mechanical sculptures. You might be an industrial designer who needs to be able to build quick mockups of new products, modeling both their forms and their functions. Maybe you're a cat owner, and you'd like to be able to play with your cat while you're away from home. It's a primer for people with little technical training and a lot of interest. It's for people who want to get projects done.

The main tools in this book are personal computers, web servers, and microcontrollers, the tiny computers inside everyday appliances. Over the past decade, microcontrollers and the programming tools for them have gone from being arcane items to common, easy-to-use tools. Elementary school students are using the tools that graduate students were baffled by only a decade ago. During that time, my colleagues and I have taught people from diverse backgrounds (few of them computer programmers) how to use these tools to increase the range of physical actions that computers can sense, interpret, and respond to.

In recent years, there's been a rising interest among people using microcontrollers to make their devices not only sense and control the physical world, but also talk to other things about what they're sensing and controlling. If you've built something with a Basic Stamp or a Lego Mindstorms kit, and wanted to make that thing communicate with other things you or others have built, this book is for you. It is also useful for software programmers familiar with networking and web services who want an introduction to embedded network programming.

If you're the type of person who likes to get down to the very core of a technology, you may not find what you're looking for in this book. There aren't detailed code samples for Bluetooth or TCP/IP stacks, nor are there circuit diagrams for Ethernet controller chips. The

components used here strike a balance between simplicity, flexibility, and cost. They use object-oriented hardware, requiring relatively little wiring or code. They're designed to get you to the end goal of making things talk to each other as fast as possible.

What You Need to Know

In order to get the most from this book, you should have a basic knowledge of electronics and programming microcontrollers, some familiarity with the Internet, and access to both.

Many people whose experience of programming begins with microcontrollers can do wonderful things with some sensors and a couple of servomotors, but may not have done much in the way of communication between the microcontroller and other programs on a personal computer. Similarly, many experienced network and multimedia programmers have never experimented with hardware of any sort, including microcontrollers. If you're either of these people, this book is for you. Because the audience of this book is diverse, you may find some of the introductory material a bit simple, depending on which background you're coming from. If so, feel free to skip past the stuff you know and get to the meaty parts.

If you've never used a microcontroller, you'll need a little background before starting with this book. My previous book, *Physical Computing: Sensing and Controlling the Physical World with Computers*, co-authored with Dan O'Sullivan, introduces the fundamentals of electronics, microcontrollers, and physical interaction design for beginning readers.

You should also have a basic understanding of computer programming before reading much further. If you've never done any programming, check out the Processing programming environment at www.processing.org. Processing is a simple language designed for nonprogrammers to learn how to program, yet it's powerful enough to do a number of advanced tasks. It will be used throughout this book whenever graphic interface programming is needed.

There are code examples in a few different programming languages in this book. They're all fairly simple examples, however, so if you don't want to work in the languages provided, you can rewrite them in your favorite language using the comments in these examples.

Contents of This Book

This book is composed of explanations of the concept that underlie networked objects, followed by recipes to illustrate each set of concepts. Each chapter contains instructions on how to build working projects that make use of the new ideas introduced in that chapter.

In Chapter 1, you'll encounter the major programming tools in the book, and get to "Hello World!" on each of them.

Chapter 2 introduces the most basic concepts needed to make things talk to each other. It covers the characteristics that need to be agreed upon in advance, and how keeping those things separate in your mind helps troubleshooting. You'll build a simple project that features one-to-one serial communication between a microcontroller and a personal computer using Bluetooth radios as an example of modem communication. You'll learn about data protocols, modem devices, and address schemes.

Chapter 3 introduces a more complex network: the Internet. It covers the basic devices that hold it together, and the basic relationships between devices. You'll see the messages that underlie some of the most common tasks you do on the Internet every day, and learn how to send those messages. You'll write your first set of programs to allow you to send data across the Net based on a physical activity in your home.

In Chapter 4, you'll build your first embedded device. You'll get more experience with command-line connections to the Net, and you'll connect a microcontroller to a web server without using a desktop or laptop computer as an intermediary.

Chapter 5 takes the Net connection a step further by explaining socket connections, which allow for longer interaction. In this chapter, you'll learn how to write a server program of your own that you can connect to from an embedded device, a personal computer, or anything else connected to the Net. You'll connect to this server program from the command line and from a microcontroller, in order to understand how devices of different types can connect to each other through the same server.

Chapter 6 introduces wireless communication. You'll learn some of the characteristics of wireless, along with its possibilities and limitations. Several short examples in this chapter enable you to say "Hello World!" over the air in a number of ways.

Chapter 7 offers a contrast to the socket connections of Chapter 5, introducing message-based protocols like UDP on the Internet and ZigBee and 802.15.4 for wireless networks. Instead of using the client-server model used in the earlier chapters, here you'll learn how to design conversations where each object in a network is equal to the others, exchanging information one message at a time.

Chapter 8 is about location. It introduces a few tools to help you locate things in physical space, and some thoughts on the relationship between physical location and network relationships.

Chapter 9 deals with identification in physical space and network space. In that chapter, you'll learn a few techniques for generating unique network identities based on physical characteristics. You'll also learn a bit about how a networked device's characteristics can be determined.

In the appendices, you'll find a few extra pieces that weren't appropriate to the main chapters, but that are very useful nonetheless. You'll also find a list of hardware and software resources for networked projects. In the final appendix, you'll find code listings for all of the programs in the book.
x

On Buying Parts

You'll need a lot of parts for all of the projects in this book. As a result, you'll learn about a lot of vendors. Because there are no large electronics parts retailers in my city, I buy parts online all the time. If you're lucky enough to live in an area where you can buy from a brick-and-mortar store, good for you! If not, get to know some of these vendors.

Some of them, like Jameco (www.jameco.com), Digi-Key (www.digikey.com), and Newark (www.newarkinone.com; their sister company in Europe is Farnell, www.farnell.com), are general electronics parts retailers, and sell many of the same things as each other. A full list of suppliers is listed in Appendix B. If a part is commonly found at many retailers, it is noted. Other parts are specialty parts, available from only one or two vendors. I've noted that too. Feel free to use substitute parts for things you are familiar with.

Because it's easy to order goods online, you might be tempted to communicate with vendors entirely through their websites. Don't be afraid to pick up the phone as well. Particularly when you're new to this type of project, it helps to talk tosomeone about what you're ordering, and to ask questions. You're likely to find helpful people at the end of the phone line for most of the retailers listed here. In Appendix B, I've listed phone numbers wherever possible. Use them.
x

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code.

For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Making Things Talk: Practical Methods for Connecting Physical Objects*, by Tom Igoe. Copyright 2007 O'Reilly Media, 978-0-596-51051-0." If you feel that your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.
x

Using Circuit Examples

In building the projects in this book, you're going to break things and void warranties. If you're averse to this, put this book down and walk away. This is not a book for those who are squeamish about taking things apart without knowing whether they'll go back together again.

Even though we want you to be adventurous, we also want you to be safe. Please don't take any unnecessary risks in building the projects that follow. Every set of instructions is written with safety in mind. Ignore the safety instructions at your own peril. Be sure you have the appropriate level of knowledge and experience to get the job done in a safe manner.

Please keep in mind that the projects and circuits shown in this book are for instructional purposes only. Details like power conditioning, automatic resets, RF shielding, and other things that make an electronic product certifiably ready for market are not included here. If you're designing real products to be used by people other than yourself, please do not rely on this information alone.

Technology, and the laws and limitations imposed by manufacturers and content owners, are constantly changing. Thus, some of the projects described may not work, may be inconsistent with current laws or user agreements, or may damage or adversely affect some equipment.

Your safety is your own responsibility, including proper use of equipment and safety gear, and determining whether you have adequate skill and experience. Power tools, electricity, and other resources used for these projects are dangerous, unless used properly and with adequate precautions, including safety gear. Some illustrative photos do not depict safety precautions or equipment, in order to show the project steps more clearly. These projects are not intended for use by children.

Use of the instructions and suggestions in this book is at your own risk. O'Reilly Media, Inc., disclaims all responsibility for any resulting damage, injury, or expense. It is your responsibility to make sure that your activities comply with applicable laws, including copyright.

Acknowledgments

This book is the product of many conversations and collaborations. It would not have been possible without the support and encouragement of my own network.

The Interactive Telecommunications Program in the Tisch School of the Arts at New York University has been my home for the past decade or more. It is a lively and warm place to work, crowded with many talented people. This book grew out of a class called Networked Objects that I have taught there for several years. I hope that the ideas herein represent the spirit of the place, and give you a sense of my own enjoyment working there.

Red Burns, the department's chair and founder, has supported me since I first entered this field. She's indulged my many flights of fancy, and brought me firmly down to earth when needed. She has challenged me on every project to make sure that I use technology not for its own sake, but always in the service of empowering people.

Dan O'Sullivan introduced me to physical computing and then generously allowed me to share in teaching it and shaping its role at ITP. He's been a great advisor and collaborator, and offered constant feedback as I worked. Most of the chapters started with a rambling conversation with Dan. His fingerprints are all over this book, and it's a better book for it.

Clay Shirky, Daniel Rozin, and Dan Shiffman have also been close advisors on this project. Clay's watched indulgently as the pile of parts mounted in our office and interrupted his own writing to offer opinions on my ideas as they came up. Daniel Rozin has and offered valuable critical insight as well, and his ideas are heavily influential in this book. Dan Shiffman read many drafts and offered great feedback. He also contributed many great code samples and libraries.

Fellow faculty members Marianne Petit, Nancy Hechinger, and Jean-Marc Gauthier have been supportive throughout the writing, offering encouragement and inspiration, covering departmental duties for me, and offering inspiration through their work.

The rest of the faculty and staff at ITP have also made this possible. George Agudow, Edward Gordon, Midori Yasuda, Megan Demarest, Nancy Lewis, Robert Ryan, John Duane, Marlon Evans, Tony Tseng, and Gloria Sed have tolerated all kinds of insanity in the name of physical computing and networked objects, and made things possible for me and the other faculty and students. Research residents Carlyn Maw, Todd Holoubek, John Schimmel, Doria Fan, David Nolen, Peter Kerlin, and Michael Olson have assisted both faculty and students over the past few years to realize projects that have influenced the ones you see in these chapters, both in their own classes and in general. Faculty members Patrick Dwyer, Michael Schneider, Greg Shakar, Scott Fitzgerald, Jamie Allen, Shawn Van Every, James Tu, and Raffi Krikorian have used the tools from this book in their classes, or have lent techniques of their own to the projects described here.

The students of ITP have pushed the boundaries of possibility in this area, and their work is reflected in many of the projects. I have cited specifics where they come up, but in general I'd like to thank all the students who've taken the Networked Objects class over the years, as they've helped me to understand what this is all about. Those from the 2006 and 2007 classes have been particularly influential, as they've had to learn the stuff from early drafts of this book, and have caught several important mistakes in the manuscript.

A few people have contributed significant amounts of code, ideas, or labor to this book. Geoff Smith gave me the original title for the course, Networked Objects, and introduced me to the idea of object-oriented hardware. John Schimmel showed me how to get a microcontroller to make HTTP calls. Dan O'Sullivan's server code was the root of all of my server code. All of my Processing code is more readable because of Dan Shiffman's coding style advice. Robert Faludi contributed many pieces of code, made the XBee examples in this book simpler to read, and corrected errors in many of them. Max Whitney helped me get Bluetooth exchanges working, and to get the cat bed finished (despite her allergies!). Dennis Crowley made the possibilities and limitations of 2D barcodes clear to me. Chris Heathcote heavily influenced my ideas on location. Durrell Bishop helped me to think about identity. Mike Kuniavsky and the folks at the Sketching in Hardware workshops in 2006 and 2007 helped me to see this work as part of a larger community, and introduced me to a lot

of new tools. Noodles the cat put up with all manner of silliness in order to finish the cat bed and its photos. No animals were harmed in the making of this book, though one was bribed with catnip.

Casey Reas and Ben Fry have made the software side of this book possible by creating Processing. Without Processing, the software side of networked objects was much more painful. Without Processing, there would be no simple, elegant programming interface for Arduino and Wiring. The originators of Arduino and Wiring have made the hardware side of this book possible. Massimo Banzi, Gianluca Martino, David Cuartielles, and David Mellis on Arduino, Hernando Barragán on Wiring, and Nicholas Zambetti bridging the two. I have been lucky to work with them.

Though I've tried to use and cite many hardware vendors in this book, special mention must be made of Nathan Seidle at SparkFun. This book would not be what it is without him. While I've been talking about object-oriented hardware for years, Nathan and the folks at SparkFun have been quietly making it a reality.

Thanks also to the support team at Lantronix. Their products are good and their support is excellent. Garry Morris, Gary Marrs, and Jenny Eisenhauer have answered countless emails and phone calls from me helpfully and cheerfully.

I have drawn ideas from many colleagues from around the world in these projects through conversations in workshops and visits. Thanks to the faculty and students I've worked with at the Royal College of Art's Interaction Design program, UCLA's Digital Media | Arts program, the Interaction Design program at the Oslo School of Architecture and Design, Interaction Design Institute Ivrea, and the Copenhagen Institute of Interaction Design.

Many networked object projects have inspired this writing. Thanks to those whose work illustrates the chapters: Tuan Anh T. Nguyen, Joo Youn Paek, Doria Fan, Mauricio Melo, and Jason Kaufman, Tarikh Korula and Josh Rooke-Ley of Uncommon Projects, Jin-Yo Mok, Alex Beim, Andrew Schneider, Gilad Lotan and Angela Pablo, Mouna Andraos and Sonali Sridhar, Frank Lantz and Kevin Slavin of Area/Code, and Sarah Johansson.

Working for MAKE has been a great experience. Dale Dougherty has been encouraging of all of my ideas, patient with my delays, and indulgent when I wanted to try new things. He's never said no without offering an acceptable alternative (and often a better one). Brian Jepson has gone above and beyond the call of duty as an editor, building all of the projects, suggesting modifications, debugging code, helping with photography and illustrations, and being endlessly encouraging. It's an understatement to say that I couldn't have done this without him. I could not have asked for a better editor. Thanks to Nancy Kotary for her excellent copyedit of the manuscript. Katie Wilson made this book far better looking and readable that I could have hoped for. Thanks also to Tim Lillis for the illustrations. Thanks to all of the MAKE team.

Thanks to my agents: Laura Lewin, who got the ball rolling; Neil Salkind, who picked it up from her; and the whole support team at Studio B. Thanks finally to my family and friends who listened to me rant enthusiastically or complain bitterly as this book progressed. Much love to you all.
X

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a website for this book, where we list errata, examples, and any additional information. You can access this page at: www.makezine.com/go/MakingThingsTalk

To comment or ask technical questions about this book, send email to: bookquestions@oreilly.com

Maker Media is a division of O'Reilly Media devoted entirely to the growing community of resourceful people who believe that if you can imagine it, you can make it. Consisting of *MAKE Magazine*, *CRAFT Magazine*, Maker Faire, and the Hacks series of books, Maker Media encourages the Do-It-Yourself mentality by providing creative inspiration and instruction.

For more information about Maker Media, visit us online:
MAKE: www.makezine.com
CRAFT: www.craftzine.com
Maker Faire: www.makerfaire.com
Hacks: www.hackszine.com


FOCUS
GAIN

1

MAKE: PROJECTS

The Tools

This book is a cookbook of sorts, and this chapter covers the staple ingredients. The concepts and tools you'll use in every chapter are introduced here. There's enough information on each tool to get you to the point where you can make the tool say **"Hello World!"** Chances are you've used some of the tools in this chapter before, or other tools just like them. Skip past the things you know and jump into learning the tools that are new to you. You may want to explore some of the less-familiar tools on your own to get a sense of what they can do. The projects in the following chapters only scratch the surface of what's possible for most of these tools. References for further investigation are provided.

Happy Feedback Machine by Tuan Anh T. Nguyen
The main pleasure of interacting with this piece comes from the feel of flipping the switches and turning the knobs. The lights and sounds produced as a result are secondary, and most people who play with it remember the feel of it rather than its behavior.

It Starts with the Stuff You Touch

All of the objects that you'll encounter in this book, tangible or intangible, will have certain behaviors. Software objects will send and receive messages, store data, or both. Physical objects will move, light up, or make noise. The first question to ask about any of them is: what does it do? The second is: how do I make it do what it's supposed to do? Or, more simply, what is its interface?

An object's interface is made up of three elements. First, there's the physical interface. This is the stuff you touch. The knobs, switches, keys, and other sensors that make up the physical interface react to your actions. The connectors that join objects are also part of the physical interface. Many of the projects in this book will show you how to build physical interfaces. Every network of objects begins and ends with a physical interface. Even though some objects in a network (software objects) have no physical interface, people build their mental models of how a system works based on the physical interface. A computer is much more than the keyboard, mouse, and screen, but that's what we think of it as, because that's what we see and touch. You can build all kinds of wonderful functions into your system, but if those functions aren't apparent in the things people get to see, hear, and touch, your wonderful functions will never get used. Remember the lesson of the VCR clock that constantly blinks 12:00 because no one can be bothered to learn how to set it: if the physical interface isn't good, the rest of the system suffers.

Second, there's the software interface, the commands that you send to the object to make it respond. In some projects, you'll invent your own software interface, and in others, you'll rely on existing interfaces to do the work for you. The best software interfaces have simple, consistent functions that result in predictable outputs. Unfortunately, not all software interfaces are as simple as you'd like them to be, so be prepared to have to experiment a little to get some software objects to do what you think they should do. When you're learning a new software interface, it helps to approach it mentally in the same way you do with a physical interface. Don't try to use all the functions at once. Learn what each function does on its own before you try to use them all together. You don't learn to play the piano by starting with a Bach fugue — you start one note at a time. Likewise, you don't learn a software interface by writing a full application with it — you learn it one function at a time. There are many projects in this book; if you find any of their software functions confusing, write a simple program that demonstrates just that function, then return to the project.

Finally, there's the electrical interface, the pulses of electrical energy sent from one device to another to be interpreted as information. Unless you're designing new objects or the connections between them, you never have to deal with this interface. When you're designing new objects or the networks that connect them, however, you have to know and understand a few things about the electrical interface, so that you know how to match up objects that might have slight differences in their electrical interfaces.
X

It's About Pulses

In order to communicate with each other, objects use communications protocols. A protocol is a series of mutually agreed-upon standards for communication between two or more objects.

Serial protocols like RS-232, USB, and IEEE 1394 (also known as FireWire and i.Link) connect computers to printers, hard drives, keyboards, mice, and other peripheral devices. Network protocols like Ethernet and TCP/IP connect multiple computers to each other through network hubs, routers, and switches. A communications protocol usually defines the rate at which messages are exchanged, the arrangement of data in the messages, and the grammar of the exchange. If it's a protocol for physical objects, it will also specify the electrical characteristics, and sometimes even the physical shape of the connectors. Protocols don't specify what happens between objects, however. The commands to make an object do something rely on protocols in the same way that clear instructions rely on good grammar. You can't give good instructions if you can't form a good sentence.

One thing that all communications protocols share, from the simplest chip-to-chip message to the most complex network architecture, is this: it's all about pulses of energy. Digital devices exchange information by sending timed pulses of energy across a shared connection. The USB connection from your mouse to your computer uses two wires for transmission and reception, sending timed pulses of electrical energy across those wires. Likewise, wired network connections are made up of timed pulses of electrical energy sent down the wires. For longer distances and higher bandwidth, the electrical wires may be replaced with fiber optic cables carrying timed pulses of light. In cases where a physical connection is inconvenient or impossible, the transmission can be sent using pulses of radio energy between radio transceivers (a transceiver is two-way radio, capable of transmitting and receiving). The meaning of data pulses is independent of the medium that's carrying them. You can use the same sequence of pulses whether you're sending them across wires, fiber optic cables, or radios. If you keep in mind that all of the communication you're dealing with starts with a series of pulses, and that somewhere there's a guide explaining the sequence of those pulses, you can work with any communication system you come across.
X

Computers of all Shapes and Sizes

You'll encounter at least four different types of computers in this book, grouped according to their physical interfaces. The most familiar of these is the personal computer. Whether it's a desktop or a laptop machine, it's got a keyboard, a screen, and a mouse, and you probably use it just about every working day. These three elements: the keyboard, the screen, and the mouse — make up its physical interface.

The second type of computer you'll encounter in this book, the microcontroller, has no physical interface that humans can interact with directly. It's just an electronic chip with input and output pins that can send or receive electrical pulses. Using a microcontroller is a three-stage process:

1. You connect sensors to the inputs to convert physical energy like motion, heat, and sound into electrical energy.
2. You attach motors, speakers, and other devices to the outputs to convert electrical energy into physical action.
3. Finally, you write a program to determine how the input changes affect the outputs.

In other words, the microcontroller's physical interface is whatever you make of it.

The third type of computer in this book, the network server, is basically the same as a desktop computer, and may even have a keyboard, screen, and mouse. Even though it can do all the things you expect of a personal computer, its primary function is to send and receive data over a network. Most people using servers don't think of them as physical things, because they only interact with them over a network, using their local computers as physical interfaces to the server. A server's most important interface for most users' purposes is its software interface.

The fourth group of computers is a mixed bag: mobile phones, music synthesizers, and motor controllers, to name a few. Some of them will have fully developed physical interfaces, some of them will have minimal physical interfaces but detailed software interfaces, and most will have a little of both. Even though you don't normally think of these devices as computers, they are. When you think of them as programmable objects, with interfaces that you can manipulate, it's easier to figure out how they can all communicate with each other, regardless of their end function.

X

Good Habits

Networking objects is a bit like love. The fundamental problem in both is that when you're sending a message, you never really know whether the receiver understands what you're saying, and there are a thousand ways for your message to get lost or garbled in transmission.

You may know why you feel the way you do, but your partner doesn't. All he or she has to go on are the words you say and the actions you take. Likewise, you may know exactly what message your local computer is sending, how it's sending it, and what all the bits mean, but the remote computer has no idea what they mean unless you program it to understand them. All it has to go on are the bits it receives. If you want reliable, clear communications (in love or networking), there are a few simple things you have to do:

- Listen more than you speak.
- Never assume that what you said is what they heard.
- Agree on how you're going to say things in advance.
- Ask politely for clarification when messages aren't clear.

Listen More Than You Speak

The best way to make a good first impression, and to maintain a good relationship, is to be a good listener. Listening is more difficult than speaking. You can speak anytime you want to, but you never know when the other person is going to say something, so you have to listen all the time. In networking terms, this means that you should write your programs such that they're listening for new messages most of the time, and sending messages only when necessary. It's often easier to send out messages all the time rather than figure out when it's appropriate, but it can lead to all kinds of problems. It usually doesn't take a lot of work to limit your sending, and the benefits far outweigh the costs.

Never Assume

What you say is not always what the other person hears. Sometimes it's a matter of misinterpretation, and other times, you may not have been heard clearly. If you assume that the message got through and continue on obliviously, you're in for a world of hurt. Likewise, you may be tempted to work out all the logic of your system, and all the steps of your messages before you start to connect things together, then build it, then test it all at once. Avoid that temptation.

It's good to plan the whole system out in advance, but build it and test it in baby steps. Most of the errors that occur in building these projects occur in the communication between objects. Always send a quick "Hello World!" message from one object to the others and make sure that the message got there intact before you proceed to the more complex details. Keep that "Hello World!" example on hand for testing when communication fails.

Getting the message wrong isn't the only wrong step you can make. Most of the projects in this book involve building the physical, software, and electrical elements of the interface. One of the most common mistakes people make when developing hybrid projects like these is to assume that the problems are all in one place. Quite often, I've sweated over a bug in the software transmission of a message, only to find out later that the receiving device wasn't even connected, or wasn't ready to receive messages. Don't assume that communication errors are in the element of the system with which you're most familiar.

They're most often in the element with which you're least familiar, and therefore are avoiding. When you can't get a message through, think about every link in the chain from sender to receiver, and check every one. Then check the links you overlooked.

Agree on How You Say Things

In good relationships, you develop a shared language based on shared experience. You learn the best ways to say things so that your partner will be most receptive, and you develop shorthand for expressing things that you repeat all the time. Good data communications also rely on shared ways of saying things, or protocols. Sometimes you make up a protocol yourself for all the objects in your system, and other times you have to rely on existing protocols. If you're working with a previously established protocol, make sure you understand what all the parts are before you start trying to interpret it. If you have the luxury of making up your own protocol, make sure you've considered the needs of both the sender and receiver when you define it. For example, you might decide to use a protocol that's easy to program on your web server, but turns out to be impossible to handle on your microcontroller. A little thought to the strengths and weaknesses on both sides of the transmission and a little compromise before you start to build will make things flow much more smoothly.

Ask Politely for Clarification

Messages get garbled in countless ways. Sometimes you hear one thing; it may not make much sense, but you act on it ... only to find out that your partner said something entirely different from what you thought. It's always best to ask nicely for clarification to avoid making a stupid mistake. Likewise, in network communications, it's wise to check that any messages you receive make sense. When they don't, ask for a repeat transmission. It's also wise to check that a message was sent, rather than assume. Saying nothing can be worse than saying something wrong. Minor problems can become major when no one speaks up to acknowledge that there's a problem. The same thing can occur in network communications. One device may wait forever for a message from the other side, not knowing that the remote device is unplugged, or perhaps it didn't get the initial message. When no response is forthcoming, send another message. Don't resend it too often, and give the other party time to reply before resending. Acknowledging messages may seem like a luxury, but it can save a whole lot of time and energy when you're building a complex system.

X

Tools

As you'll be working with the physical, software, and electrical interfaces of objects, the tools you'll need are physical tools, software, and (computer) hardware.

Physical Tools

If you've worked with electronics or microcontrollers before, chances are you have your own hand tools already. Figure 1-1 shows the ones used most frequently in this book. They're common tools, and can be obtained from many vendors. A few are listed in Table 1-1.

In addition to hand tools, there are some common electronic components that you'll use all the time. They're listed as well, with part numbers from the retailers featured most frequently in this book. Not all retailers will carry all parts, so there are many gaps in the table.

NOTE: You'll find a number of component suppliers in this book. I buy from different vendors depending on who's got the best and the least expensive version of each part. Sometimes it's easier to buy from a vendor that you know carries what you need rather than search through the massive catalog of a vendor who might carry it cheaper. Feel free to substitute your favorite vendors. A list of vendors can be found in Appendix B.

Figure 1-1. See list at right for number references.

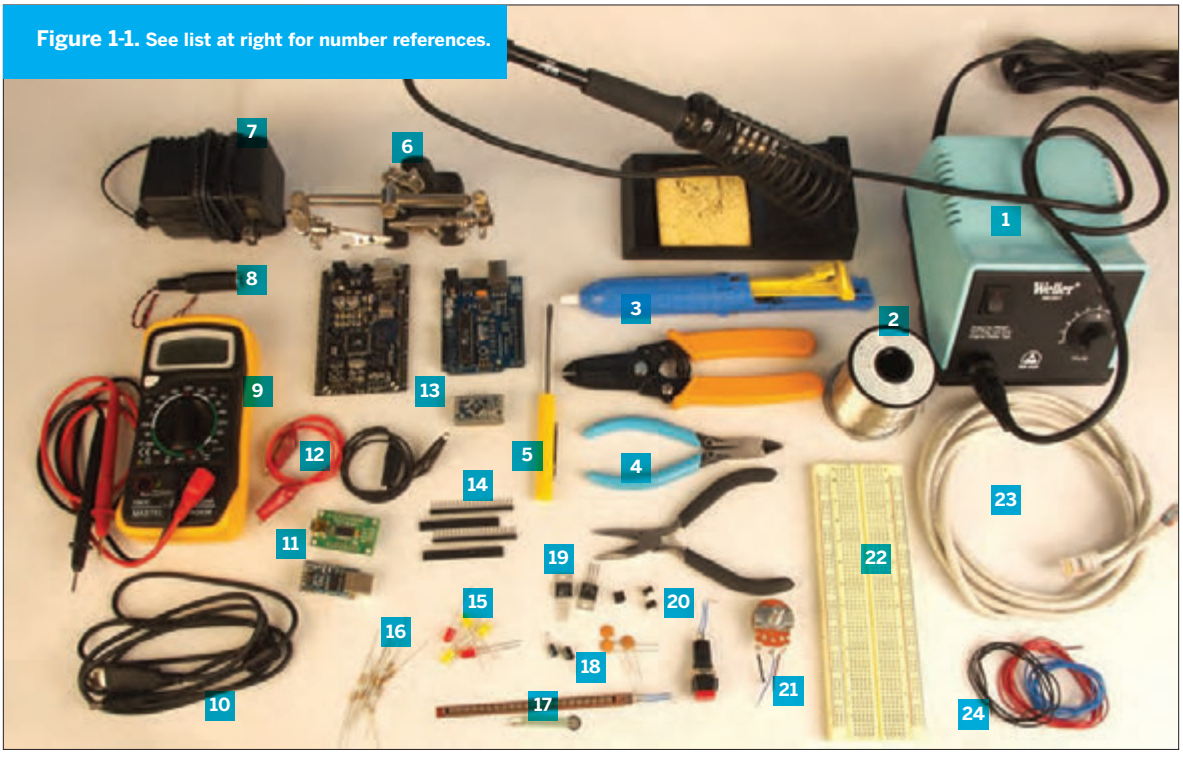


Table 1-1. Common tools for electronic and microcontroller work.

D Digi-Key (digikey.com)
I Images SI (imagesco.com)
J Jameco (jameco.com)
S SparkFun Electronics (sparkfun.com)

RESISTORS

100Ω	**D** 100QBK-ND, **J** 690620
220Ω	**D** 220QBK-ND, **J** 690700
470Ω	**D** 470QBK-ND, **J** 690785
1K	**D** 1.0KQBK, **J** 29663
10K	**D** 10KQBK-ND, **J** 29911
22K	**D** 22KQBK-ND, **J** 30453
100K	**D** 100KQBK-ND, **J** 29997
1M	**D** 1.0MQBK-ND, **J** 29698

CAPACITORS

0.1µF ceramic	**D** 399-4151-ND, **J** 15270
1µF electrolytic	**D** P10312-ND, **J** 94161
10µF electrolytic	**D** P11212-ND, **J** 29891, **S** COM-00523
100µF electrolytic	**D** P10269-ND, **J** 158394, **S** COM-00096

VOLTAGE REGULATORS

3.3V	**D** 576-1134-ND, **J** 242115, **S** COM-00526
5V	**D** LM7805CT-ND, **J** 51262, **S** COM-00107

ANALOG SENSORS

Flex sensors	**J** 150551, **I** FLX-01
FSRs	**P** 30056, **I** FSR-400, 402, 406, 408

LED

T1, Green clear	**D** 160-1144-ND, **J** 34761
T1, Red, clear	**D** 160-1665-ND, **J** 94511

TRANSISTORS

2N2222A	**J** 38236
TIP120	**J** 32993

DIODES

1N4004-R	**D** 1N4004-E3 or 23GI-ND, **J** 35992
3.3V zener (1N5226)	**D** 1N5226B-TPCT-ND, **J** 743488

PUSHBUTTONS

PCB	**D** SW400-ND, **J** 119011, **S** COM-00097
Panel Mount	**D** GH1344-ND, **J** 164559PS

SOLDERLESS BREADBOARDS

various	**D** 438-1045-ND, **J** 20723, 20600, **S** PRT-00137

HOOKUP WIRE

red	**J** 36856, **S** PRT-08023
black	**J** 36792, **S** PRT-08022
blue	**J** 36767
yellow	**S** PRT-08024

POTENTIOMETER

10K	**D** 29081

HEADER PINS

straight	**D** A26509-20-ND, **J** 103377, **S** PRT-00116
right angle	**D** S1121E-36-ND, **S** PRT-00553

HEADERS

female	**S** PRT-00115

BATTERY SNAP

9V	**D** 2238K-ND, **J** 101470PS, **S** PRT-00091

Handy hand tools for networking objects.

1 Soldering iron Middle-of-the-line is best here. Cheap soldering irons die fast, but a mid-range iron like the Weller WLC-100 work great for small electronic work. Avoid the Cold Solder irons. They solder by creating a spark, and that spark can damage static-sensitive parts like microcontrollers. Jameco (jameco.com): 146595; RadioShack: 640-2802 and 640-2078

2 Solder 21-23 AWG solder is best. Get lead-free solder if you can, it's healthier for you. Jameco: 668271; RadioShack: 640-0013

3 Desoldering pump This helps when you mess up while soldering. Jameco: 305226; SparkFun (sparkfun.com): TOL-00082

4 Wire stripper, Diagonal cutter, Needlenose pliers Avoid the 3-in-1 versions of these tools. They'll only make you grumpy. These three tools are essential for working with wire, and you don't need expensive ones to have good ones.
Wire stripper: Jameco: 159291; RadioShack: 640-2129A; SparkFun: TOL-00089
Diagonal cutter: Jameco: 161411; RadioShack: 640-2043; SparkFun: TOL-00070
Needlenose pliers: Jameco: 35473; RadioShack: 640-2033; SparkFun: TOL-00079

5 Mini-screwdriver Get one with both Phillips and slotted heads. You'll use it all the time. Jameco: 127271; RadioShack: 640-1963

6 Helping hands These make soldering much easier. Jameco: 681002

7 9–12V DC power supply You'll use this all the time, and you've probably got a spare from some dead electronic device. Make sure you know the polarity of the plug so you don't reverse polarity on a component and blow it up! Most of the devices shown in this book have a DC power jack that accepts a 2.1mm inner diameter/5.5mm outer diameter plug, so look for an adaptor with the same dimensions. Jameco: 170245 (12V, 1000mA); RadioShack: 273-1667 (3–12V, 800mA); SparkFun: TOL-00298

8 Power connector, 2.1mm inside diameter/5.5mm outside diameter You'll need this to connect your microcontroller module or breadboard to a DC power supply. This size connector is the most common for the power supplies that will work with the circuits you'll be building here. Jameco: 159610; Digi-Key (digikey.com): CP-024A-ND

9 Multimeter You don't need an expensive one. As long as it measures voltage, resistance, amperage, and continuity, it'll do the job. Jameco: 220812; RadioShack: 22-810; SparkFun: TOL-00078

10 USB cables You'll need both USB A-to-B (the most common USB cables) and USB A-to-mini-B (the kind that's common with digital cameras) for the projects in this book. SparkFun: CAB-00512, CAB-00598

11 Serial-to-USB converter This converter lets you speak TTL serial from a USB port. Breadboard serial-to-USB modules like the FT232 modules shown here are cheaper than the consumer models, and easier to use in the projects in this book. SparkFun: BOB-00718 or DEV-08165

12 Alligator clip test leads It's often hard to juggle the five or six things you have to hold when metering a circuit. Clip leads make this much easier. Jameco: 10444; RadioShack: 278-016; SparkFun: CAB-00501

13 Microcontroller module The microcontrollers shown here are the Arduino NG and the Arduino Mini. Available from SparkFun and Make (store.makezine.com) in the U.S., PCB-Europe in Europe (pcb-europe.net/catalog/) and from multiple distributors internationally. See arduino.cc/en/Main/Buy for details in your region.

14 Header pins You'll use these all the time. It's handy to have female ones around as well. Jameco: 103377; Digi-Key: A26509-20-ND; SparkFun: PRT-00116

15 Spare LEDs for tracing signals LEDs are to the hardware developer what print statements are to the software developer. They let you see quickly if there's voltage between two points, or if a signal's going through. Keep spares on hand. Jameco: 3476; RadioShack: 276-0069; Digi-Key: 160-1144-ND, 160-1665-ND

16 Resistors You'll need resistors of various values for your projects. Common values are listed in Table 1-1.

17 Analog sensors (variable resistors) There are countless varieties of variable resistors to measure all kinds of physical properties. They're the simplest of analog sensors, and they're very easy to build into test circuits. Flex sensors and force-sensing resistors are handy for testing a circuit or a program.
Flex sensors: Jameco: 150551; Images SI: FLX-01
Force-sensing resistors: Parallax: 30056; Images SI: FSR-400, 402, 406, 408

18 Capacitors You'll need capacitors of various values for your projects. Common values are listed in Table 1-1.

19 Voltage regulators Voltage regulators take a variable input voltage and output a constant (lower) voltage. The two most common you'll need for these projects are 5V and 3.3V. Be careful when using a regulator that you've never used before. Check the data sheet to make sure you have the pin connections correct.
3.3V: Digkey: 576-1134-ND; Jameco: 242115; SparkFun: COM-00526
5V: Digkey: LM7805CT-ND; Jameco: 51262; SparkFun: COM-00107

20 Pushbuttons There are two types you'll find handy: the PCB-mount type like the ones you find on Wiring and Arduino boards, used here mostly as reset buttons for breadboard projects; and panel-mount types used for interface controls for end users. But you can use just about any type you want.
PCB-mount type: Digi-Key: SW400- ND; Jameco: 119011; SparkFun: COM-00097
Panel-mount type: Digi-Key: GH1344-ND; Jameco: 164559PS

21 Potentiometers You'll need potentiometers to let people adjust settings in your project. Jameco: 29081

22 Solderless breadboard Having a few around can be handy. I like the ones with two long rows on either side, so you can run power and ground on both sides. Jameco: 20723 (2 bus rows per side); RadioShack: 276-174 (1 bus row per side); Digi-Key: 438-1045-ND; SparkFun: PRT-00137

23 Ethernet cables A couple of these will come in handy. Jameco: 522781

24 Black, red, blue, yellow wire 22 AWG solid-core hook-up wire is best for making solderless breadboard connections. Get at least three colors, and always use red for voltage and black for ground. A little organization of your wires can go a long way.
Black: Jameco: 36792
Blue: Jameco: 36767
Green: Jameco: 36821
Red: Jameco: 36856; RadioShack: 278-1215
Yellow: Jameco: 36919
Mixed: RadioShack: 276-173

Figure 1-2
The Processing editor window.

Software Tools

Processing

The multimedia programming environment used in this book is called Processing. It's based on Java, and made for designers, artists, and others who don't need to know all the gory details of programming, but want to get something done. It's a useful tool for explaining programming ideas because it takes relatively little Processing code to make big things happen, such as opening a network connection, connecting to an external device through a serial port, or controlling a camera through FireWire. It's a free, open source tool available from www.processing.org. Because it's based on Java, you can include Java classes and methods in your Processing programs. It runs on Mac OS X, Windows, and Linux, so almost anyone can run Processing on their favorite operating system. If you don't like working in Processing, you should be able to use the code samples here and their comments as pseudocode for whatever multimedia environment you prefer. Once you've downloaded and installed Processing on your computer, open the application. You'll get a screen that looks like Figure 1-2.

» Here's your first Processing program. Type this into the editor window, and press the Run button on the top left-hand side of the toolbar:

```
println("Hello World!\n");
```

It's not too flashy a program, but it's a classic. It should print Hello World! in the message box at the bottom of the editor window. It's that easy.

Programs in Processing are called sketches, and all the data for a sketch is saved in a folder with the sketch's name. The editor is very basic, without a lot of clutter to get in your way. The toolbar has buttons to run and stop a sketch, create a new file, open an existing sketch, save the current sketch, or export to a Java applet. You can also export your sketch as a standalone application from the File menu. Files are normally stored in a subdirectory of your **Documents** folder called **Processing**, but you can save them wherever you prefer if you don't like them there.

Here's a second program that's a bit more exciting. It illustrates some of the main programming structures in Processing:

```
/*
    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);
  }
}
```

Every Processing program has two main routines, setup() and draw(). setup() happens once at the beginning of the program. It's where you set all your initial conditions, like the size of the applet window, initial states for variables, and so forth. draw() is the main loop of the program. It repeats continuously until you close the applet window.

In order to use variables in Processing, you have to declare the variable's data type. In the preceding program, the variables redValue, greenValue, and blueValue are all float types, meaning that they're floating decimal-point numbers. Other common variable types you'll use are ints (integers), booleans (true or false values), Strings of text, and bytes.

Like C, Java and many other languages, Processing uses C-style syntax. All functions have a data type, just like variables (and many of them are the void type, meaning that they don't return any values). All lines end with a semicolon, and all blocks of code are wrapped in curly brackets. Conditional statements (if-then statements), for-next loops, and comments all use the C syntax as well. The preceding code illustrates all of these except the for-next loop.

» Here's a typical for-next loop. Try this in a sketch of its own (to start a new sketch, select New from Processing's File menu):

```
for (int myCounter = 0; myCounter <=10; myCounter++) {
  println(myCounter);
}
```

Processing is a fun language to play with, because you can make interactive graphics very quickly. It's also a simple introduction to Java for beginning programmers. If you're a Java programmer already, you can include Java directly in your Processing programs. Processing is expandable through code libraries. You'll be using two of the Processing code libraries frequently in this book: the serial library and the networking library.

For more on the syntax of Processing, see the language reference guide at www.processing.org. To learn more about programming in Processing, check out *Processing: A Programming Handbook for Visual Designers and Artists*, by Casey Reas and Ben Fry (MIT Press, 2007), the creators of Processing.

BASIC users: If you've never used a C-style for-next loop, it can seem a bit forbidding. What this bit of code does is establish a variable called myCounter**. As long as number is less than or equal to ten, it executes the instructions in the curly brackets.** myCounter++ **tells the program to add one to** myCounter **each time through the loop. The equivalent BASIC code is:**

```
for myCounter = 0 to 10
  Print myCounter
next
```

Mac OS X Users: Once you've downloaded and installed Processing, there's an extra step you'll need to take that will make the projects in this book that use Processing possible. Go to the Processing application directory, then to the libraries/serial/ subdirectory. There's a file there called macosx_setup.command. Double-click this. It will run a script that enables Processing to use serial communication to USB, Bluetooth, and other devices. A terminal window will open and run a script that will ask you a few questions. It will also ask for your administrator password, so don't run it unless you have administrator access to your machine. Say "yes" to anything it asks, and provide your password when needed. When it's done, you'll be able to use the serial ports of your computer through Processing. You'll be making heavy use of this capability later on in this book.

Remote Access Applications

One of the most effective debugging tools you'll use in making the projects in this book is a command-line remote access program, which allows you access to the command-line interface of a remote computer. If you've never used a command-line interface before, you'll find it a bit awkward at first, but you get used to it pretty quickly. This tool is especially important when you need to log into a web server, as you'll need the command line to create PHP scripts that will be used in this book.

Most web hosting providers are based on Linux, BSD, Solaris or some other Unix-like operating system. So, when you need to do some work on your web server, you may need to make a command-line connection to your web server.

If you already know how to create PHP and HTML documents and upload them to your web server, you can skip ahead to the "PHP" section.

In a command-line interface, everything is done by typing commands at the cursor. The programs you'll be running and the files you'll be writing and reading aren't on your machine. When you're using the PHP programming language described shortly, for example, you'll be using programs and reading files directly on the web host's computer.

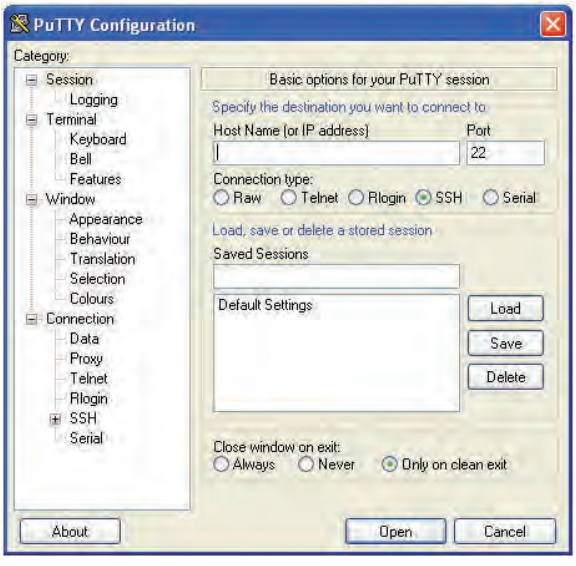
Although this is the most direct way to work with PHP, some people prefer to work more indirectly, by writing text files on their local computers and uploading them to the remote computer. Depending on how restrictive your web hosting service is, this may be your only option (however, there are many inexpensive hosting companies that offer full command-line access). Even if you prefer to work this way, there are times in this book when the command line is your only option, so it's worth getting to know a little bit about it now.

On Windows computers, there are a few remote access programs available, but the one that you'll use here is called PuTTY. You can download it from www.puttyssh.org. Download the Windows-style installer and run it. On Mac OS X and Linux, you can use OpenSSH, which is included with both operating systems, and can be run in the Terminal program with the command ssh.

Before you can run OpenSSH, you'll need to launch a terminal emulation program, which gives you access to your Linux or Mac OS X command line. On Mac OS X, the program is called Terminal, and you can find it in the **Utilities** subdirectory of the **Applications** directory. On Linux, look for a program called xterm, rxvt, Terminal, or Konsole.

NOTE: ssh **is a more modern cousin of a longtime Unix remote access program called** telnet**.** ssh **is more secure, in that it scrambles all data sent from one computer to another before sending it, so it can't be snooped on en route.** telnet **sends all data from one computer to another with no encryption. You should use** ssh **to connect from one machine to another whenever you can. Where** telnet **is used in this book, it's because it's the only tool that will do what's needed for the examples in question. Think of** telnet **as an old friend: maybe not the coolest guy on the block, maybe he's a bit of a gossip, but he's stood by you forever, and you know you can trust him to do the job when everyone else lets you down.**

X



Making the SSH Connection

Mac OS X and Linux

Open your terminal program. These Terminal applications give you a plain text window with a greeting like this:

```
Last login: Wed Feb 22 07:20:34 on ttyp1
ComputerName:~ username$
```

Type ssh username@myhost.com at the command line to connect to your web host. Replace username and myhost.com with your username and host address.

Windows

On Windows, you'll need to start up PuTTY (see Figure 1-3). To get started, type myhost.com (your web host's name) in the Host Name field, choose the SSH protocol, and then click Open.

The computer will try to connect to the remote host, and asks for your password when it connects. Type it (you won't see what you type), followed by the Enter key.

Figure 1-3
The main PuTTY window.

Using the Command Line

Once you've connected to the remote web server, you should see something like this:

```
Last login: Wed Feb 22 08:50:04 2006 from 216.157.45.215
[userid@myhost ~]$
```

Now you're at the command prompt of your web host's computer, and any command you give will be executed on that computer. Start off by learning what directory you're in. To do this, type the following:

```
pwd
```

which stands for "print working directory." It asks the computer to list the name and pathname of the directory in which you're currently working. You'll see that many Unix commands are very terse, so you have to type less. The downside of this is that it makes them harder to remember. The server will respond with a directory path, such as:

```
/home/igoe
```

This is the home directory for your account. On many web servers, this directory contains a subdirectory called **public_html** or **www**, which is where your web files belong. Files that you place in your home directory (that is, outside or **www** or **public_html**) can't be seen by web visitors.

NOTE: You should check with your web host to learn how the files and directories in your home directory are set up.

To find out what files are in a given directory, use the list (ls) command, like so:

```
ls -l .
```

NOTE: The dot is shorthand for "the current working directory." Similarly, a double dot is shorthand for the directory (the parent directory) that contains the current directory.

The –l means "list long." You'll get a response like this:

```
total 44
drwxr-xr-x  13 igoe users 4096 Apr 14 11:42 public_html
drwxr-xr-x   3 igoe users 4096 Nov 25  2005 share
```

This is a list of all the files and subdirectories of the current working directories, and their attributes. The first column lists who's got permissions to do what (read, modify, or execute/run a file). The second lists how many links there are to that file elsewhere on the system; it's not something you'll have much need for, most of the time. The third column tells you who owns it, and the fourth tells you the group (a collection of users) the file belongs to. The fifth lists its size, and the sixth lists the date it was last modified. The final column lists the filename.

In a Unix environment, all files whose names begin with a dot are invisible. Some files, like access-control files that you'll see later in the book, need to be invisible. You can get a list of all the files, including the invisible ones, using the –a modifier for ls, this way:

```
ls -la
```

To move around from one directory to another, there's a "change directory" command, cd. To get into the **public_html** directory, for example, type:

```
cd public_html
```

To go back up one level in the directory structure, type:

```
cd ..
```

To return to your home directory, use the ~ symbol, which is shorthand for your home directory:

```
cd ~
```

If you type cd on a line by itself, it also takes you to your home directory.

If you wanted to go into a subdirectory of a directory, for example the **cgi-bin** directory inside the **public_html** directory, you'd type cd public_html/cgi-bin. You can type the absolute path from the main directory of the server (called the root) by placing a **/** at the beginning of the file's pathname. Any other file pathname is called a relative path.

To make a new directory, type:

```
mkdir directoryname
```

This command will make a new directory in the current working directory. If you then use ls -l to see a list of files in the working directory, you'll see a new line with the new directory. If you then type cd directoryname to switch to the new directory and ls –la to see all of its contents, you'll see only two listings:

```
drwxr-xr-x  2 tqi6023 users 4096 Feb 17 10:19 .
drwxr-xr-x  4 tqi6023 users 4096 Feb 17 10:19 ..
```

The first file, **.** , is a reference to this directory itself. The second, **..** , is a reference to the directory that contains it. Those two references will exist as long as the directory exists. You can't change them.

To remove a directory, type:

```
rmdir directoryname
```

You can remove only empty directories, so make sure that you've deleted all the files in a directory before you remove it. rmdir won't ask you if you're sure before it deletes your directory, though, so be careful. Don't remove any directories or files that you didn't make yourself until you know your way around.

Controlling Access to Files

Type `ls -l .` to get a list of files in your current directory and take a closer look at the permissions on the files. For example, a file marked drwx------ means that it's a directory, and that it's readable, writable, and executable by the system user that created the directory (also known as the owner of the file). Or take the file marked -rw-rw-rw. The – at the beginning means it's a regular file, not a directory, and that the owner, the group of users that the file belongs to (usually, this is the group that the owner is a member of), and everyone else who accesses the system can read and write to this file. The first rw- refers to the owner, the second refers to the group, and the third refers to the rest of the world. If you're the owner of a file, you can change its permissions using the chmod command:

```
chmod go –w filename
```

The options following chmod refer to which users you want to affect. In the preceding example, you're removing write permission (-w) for the group (g) that the file belongs to, and for all others (o) besides the owner of the file. To restore write permissions for the group and others, and to also give them execute permission, you'd type:

```
chmod go +wx filename
```

A combination of u for user, g for group, and o for others, and a combination of + and – and r for read, w for write, and x for execute gives you the capability to change permissions on your files for anyone on the system. Be careful not to accidentally remove permissions from yourself (the user). Also, get in the habit of not leaving files accessible to the group and others unless you need to: on large hosting providers, it's not unusual for you to be sharing a server with hundreds of other users!

Creating, Viewing, and Deleting Files

Two other command-line programs you'll find useful are nano and less. nano is a text editor. It's very bare-bones, and you may prefer to edit your files using your favorite text editor on your own computer and then upload them to your server. But for quick changes right on the server, nano is great. To make a new file, type:

```
nano filename.txt
```

The nano editor will open up. Figure 1-4 shows what it looked like after I typed in some text.

All the commands to work in nano are keyboard commands you type using the Control key. For example, to exit the program, type Control-X. The editor will then ask you if you want to save, and prompt you for a filename. The most common commands are listed along the bottom of the screen.

While nano is for creating and editing files, less is for reading them. less takes any file and displays it to the screen one screenful at a time. To see the file you just created in nano, for example, type:

```
less filename.txt
```

You'll get a list of the file's contents, with a : prompt at the bottom of the screen. Press the spacebar for the next screenful. When you've read enough, type q to quit. There's not much to less, but it's a handy way to read long files. You can even send other commands through less (or almost any command-line program) using the pipe (|) operator. For example, try this:

```
ls –la . | less
```

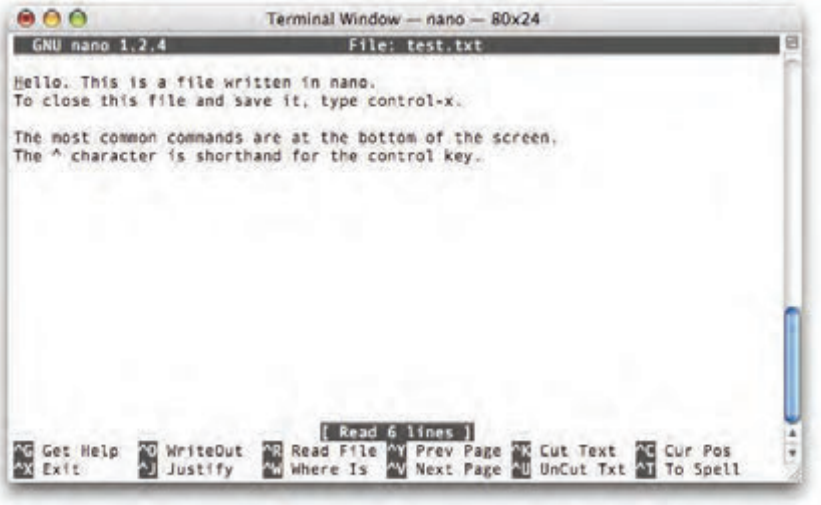


Figure 1-4
The nano text editor.

Once you've created a file, you can delete it using the rm command, like this:

```
rm filename
```

Like rmdir, rm won't ask you if you're sure before it deletes your file, so use it carefully.

There are many other commands available in the Unix command shell, but these will suffice to get you started for now. For more information, type help at the command prompt to get a list of commonly used commands. For any command, you can get its user manual by typing man commandname. For more on getting around Unix and Linux systems using the command line, see *Learning the Unix Operating System* by Jerry Peek, John Strang, and Grace Todino-Gonguet. When you're ready to close the connection to your server, type: logout

PHP

The server programs in this book are mostly in PHP. PHP is one of the most common scripting languages for applications that run on the web server (server-side scripts). Server-side scripts are programs that allow you to do more with a web server than just serve fixed pages of text or HTML. They allow you to access databases through a browser, save data from a web session to a text file, send mail from a browser, and more. You'll need a web hosting account with an Internet service provider for most of the projects in this book, and it's likely that your host already provides access to PHP. If not, talk to your system administrator to see whether it can be installed.

To get started with PHP, you'll need to make a remote connection to your web hosting account using ssh as you did in the last section. Some of the more basic web hosts don't allow ssh connections, so check with yours to see whether they do (and if yours doesn't, look around for an inexpensive hosting company that does; it will be well worth it for the flexibility of working from the command line). Once you're connected, type: php -v

You should get a reply like this:

```
PHP 4.3.9 (cgi) (built: Nov  4 2005 11:49:43)
Copyright (c) 1997-2004 The PHP Group
Zend Engine v1.3.0, Copyright (c) 1998-2004 Zend
Technologies
```

This tells what version of PHP is installed on your server. The code in this book was written using PHP4, so as long as you're running that version or later, you'll be fine. PHP makes it easy to write web pages that can display results from databases, send messages to other servers, send email, and more.

Most of the time, you won't be executing your PHP scripts directly from the command line. Instead, you'll be calling the web server application on your server, most likely a program called Apache, and asking it for a file (this is all accomplished simply by opening a web browser, typing in the address of a document on your web server, and pressing Enter — just like visiting any other web page). If the file you ask for is a PHP script, the web server application will look for your file and execute it. It'll then send a message back to you with the results.

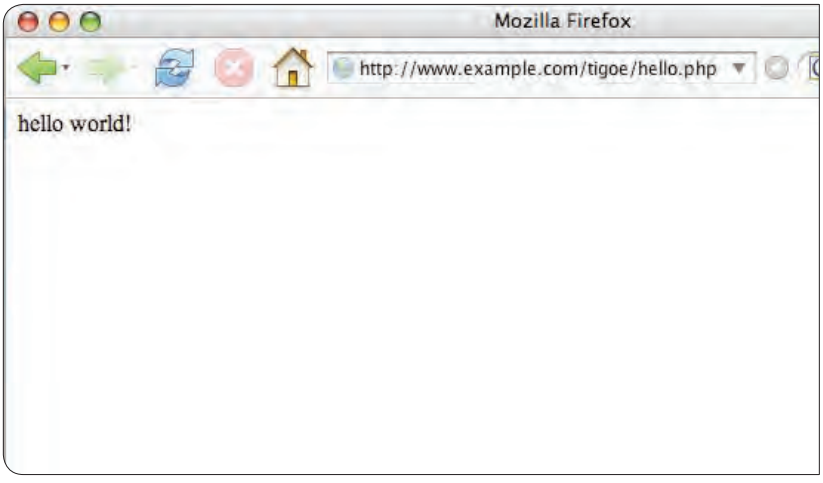


Figure 1-5
The results of your first PHP script, in a browser.

For more on this, see Chapter 3. For now, let's get a simple PHP program or two working. Here's your first PHP program. Open your favorite text editor, type this in, and save it on the server with the name **hello.php** in your **public_html** directory. (Your web pages may be stored in a different directory, such as **www** or **web/public**.)

```
<?php
echo "<html><head></head><body>\n";
echo "hello world!\n";
echo "</body></html>\n";
?>
```

Now, back at the command line, type the following to see the results:

```
php hello.php
```

You should get the following response:

```
<html><head></head><body>
hello world!
</body></html>
```

Now try opening this file in a browser. To see this program in action, open a web browser and navigate to the address of this file on your website. Because you saved it in **public_html,** the address is http://www.example.com/hello.php (replace www.example.com with your web site and any additional path info needed to access your home files, such as http://tigoe.net/~tigoe/hello.php). You should get a web page like the one in Figure 1-5.

If you see the PHP source code instead of what's shown in Figure 1-5, you may have opened up the PHP script as a local file (make sure your web browser's location bar says http:// **instead of** file://**).**

If it still doesn't work, your web server may not be configured or PHP. Another possibility is that your web server uses a different extension for php scripts, such as .php4. Consult with your web hosting provider for more information.

You may have noticed that the program is actually printing out HTML text. PHP was made to be combined with HTML. In fact, you can even embed PHP in HTML pages, by using the <? and ?> tags that start and end every PHP script. If you get an error when you try to open your PHP script in a browser, ask your system administrator if there are any requirements as to which directories PHP scripts need to be in on your server, or on the file permissions for your PHP scripts.

Here's a slightly more complex PHP script. Save it to your server in the **public_html** directory as **time.php**:

```
<?php
/*
   Date printer
   Language: PHP

   Prints the date and time in an HTML page.
*/
//    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";
?>
```

To see it in action, type http://www.example.com/time.php into your browser. You should get the date and time. You can see this program uses a variable, $date, and calls a built-in PHP function, date(), to fill the variable. You don't have to declare the types of your variables in PHP. Any simple, or scalar, variable begins with a $ and can contain an integer, a floating point number, or a string. PHP uses the same C-style syntax as Processing, so you'll see that if-then statements, repeat loops, and comments all look familiar.

For more on PHP, check out www.php.net, the main source for PHP, where you'll find some good tutorials on how to use it. You can also check out *Learning PHP 5* by David Sklar (O'Reilly Media, Inc., 2004) for a more in-depth treatment.

Serial Communication Tools

The remote access programs in the earlier section were terminal emulation programs that gave you access to remote computers through the Internet, but that's not all a terminal emulation program can do. Before TCP/IP was ubiquitous as a way for computers to connect to networks, connectivity was handled through modems attached to the serial ports of computers. Back then, many users connected to bulletin boards (BBSes) and used menu-based systems to post messages on discussion boards, download files, and send mail to other users of the same BBS.

Nowadays, serial ports are used mainly to connect to some of peripheral devices of your computer. In microcontroller programming, they're used to exchange data between the computer and the microcontroller. For the projects in this book, you'll find that using a terminal program to connect to your serial ports is indispensable. There are several freeware and shareware terminal programs available, but to keep it simple, stick with the classics: PuTTY (version 0.59 or later) for Windows users, and the GNU screen program running in a terminal window for Mac OS X and Linux users.

Windows Serial Communication

To get started, you'll need to know the serial port name. Click Start→Run (use the Search box on Vista), type devmgmt.msc, and press Enter to launch Device Manager. If you've got a serial device such as a Wiring or Arduino board attached, you'll see a listing for Ports (COM & LPT). Under that listing, you'll see all the available serial ports. Each new Wiring or Arduino board you connect will get a new name, such as COM5, COM6, COM7, and so forth.

Once you know the name of your serial port, open PuTTY. In the Session category, set the Connection Type to Serial, and enter the name of your port in the Serial Line box, as shown in Figure 1-6. Then click the Serial category at the end of the category list, and make sure that the serial line matches your port name. Configure the serial line for 9600 baud, 8 databits, 1 stop bit, no parity, and no flow control. Then click the Open button, and a serial window will open. Anything you type in this window will be sent out the serial port, and any data that comes in the serial port will be displayed here as ASCII text.

NOTE: Unless your Arduino is running a program that communicates over the serial port (and you'll learn all about that shortly), you won't get any response yet.

Mac OS X and Linux Serial Communication

To get started with serial communication in Mac OS X or Linux, open a terminal window and type:

```
ls /dev/tty.*    # Mac OS X
ls /dev/tty*     # Linux
```

This command will give you a list of available serial ports. The names of the serial ports in Mac OS X and Linux are more unique, but more cryptic than the COM1, COM2, and so on that Windows uses. Pick your serial port and type:

```
screen portname datarate.
```

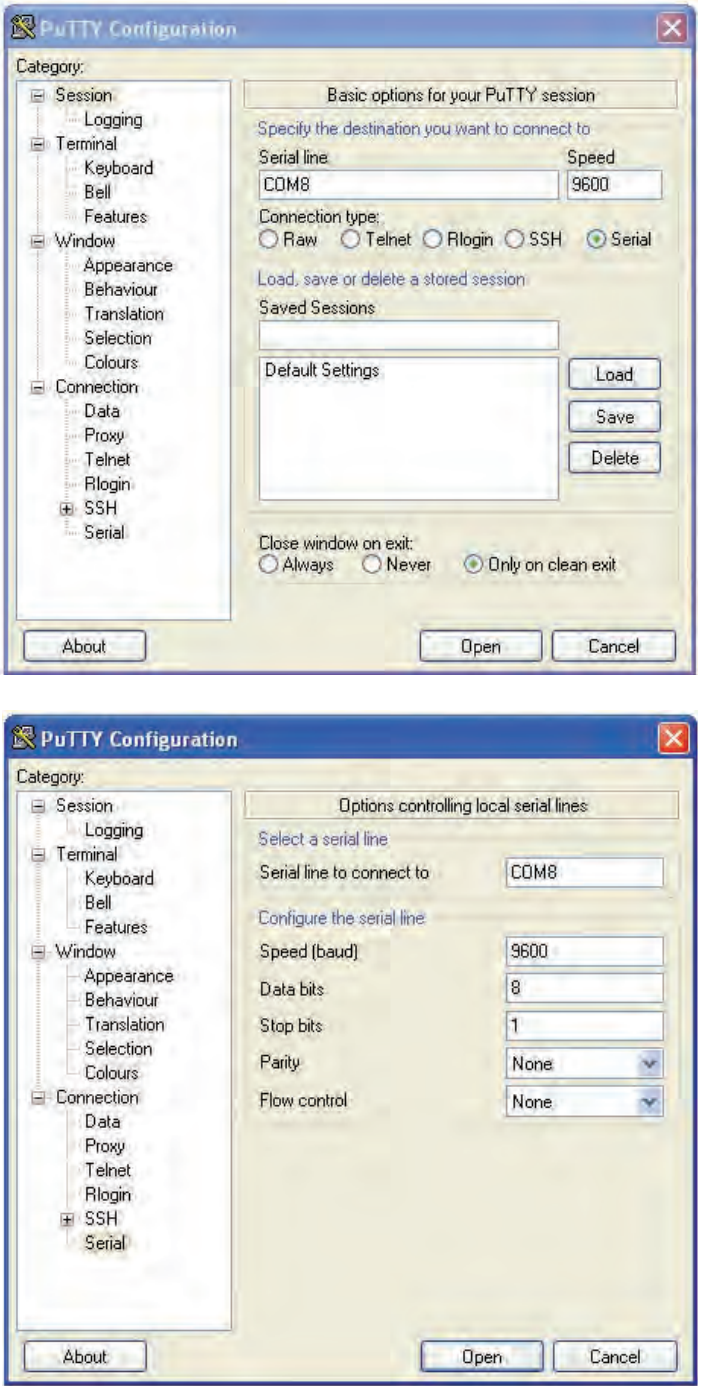


Figure 1-6
Configuring a serial connection in PuTTY.

For example, to open the serial port on an Arduino board (discussed shortly) at 9600 bits per second, you might type screen /dev/tty.usbserial-1B1 9600 on Mac OS X. On Linux, the command might be screen /dev/ttyUSB0 9600. The screen will be cleared, and any characters you type will be sent out the serial port you opened. They won't show up on the screen, however. Any bytes received in the serial port will be displayed in the window as characters. To close the serial port, type Control-A followed by Control-\.

In the next section, you'll use a serial communications program to communicate with a microcontroller.

Hardware

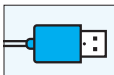
Arduino and Wiring

The main microcontroller used in this book is the Arduino module. Arduino is based on a similar module called Wiring. You should be able to use Arduino or Wiring interchangeably for the examples in this book. Both modules are the children of the Processing programming environment and the Atmel AVR family of microcontrollers. In fact, you'll find that the editors for Processing, Wiring, and Arduino look almost identical. Both programming environments are free and open source, available through hardware.processing.org. You can buy the actual modules from the original developers or from SparkFun at www.sparkfun.com or from Make at store.makezine.com. If you're a hardcore hardware geek and like to make your own printed circuit boards, you can download the plans and make your own. I recommend the former, as it's much quicker (and more reliable, for most people). Figures 1-7 and 1-8 show Wiring and several variants of Arduino.

One of the best things about Wiring and Arduino is that they are cross-platform. This is a rarity in microcontroller development environments. They work well on Mac OS X, Windows, and (with some effort) Linux.

Another good thing about these environments is that, like Processing, they can be extended. Just as you can include Java classes and methods in your Processing programs, you can include C/C++ code, written in AVR-C, in your Wiring and Arduino programs. For more on how to do this, see the Wiring and Arduino websites.

X



Who's Got the Port?

Serial ports aren't easily shared between applications. In fact, only one application can have control of a serial port at a time. If PuTTY or the screen program has the serial port open to an Arduino module, for example, the Arduino programming application can't download new code to the module. When an application tries to open a serial port, it requests exclusive control of it either by writing to a special file called a lock file or by asking the operating system to lock the file on its behalf. When it closes the serial port, it releases the lock on the serial port. Sometimes when an application crashes while it's got a serial port open, it can forget to close the serial port, with the result that no other application can open the port. When this happens, the only thing you can do to fix it is to restart the operating system, which clears all the locks (alternatively, you could wait for the operating system to figure out that the lock should be released). To avoid this problem, make sure that you close the serial port whenever you switch from one application to another. Linux and Mac OS X users should get in the habit of closing down screen with Ctrl-A Ctrl-\ every time, and Windows users should disconnect the connection in PuTTY. Otherwise, you may find yourself restarting your machine a lot.

» opposite page top

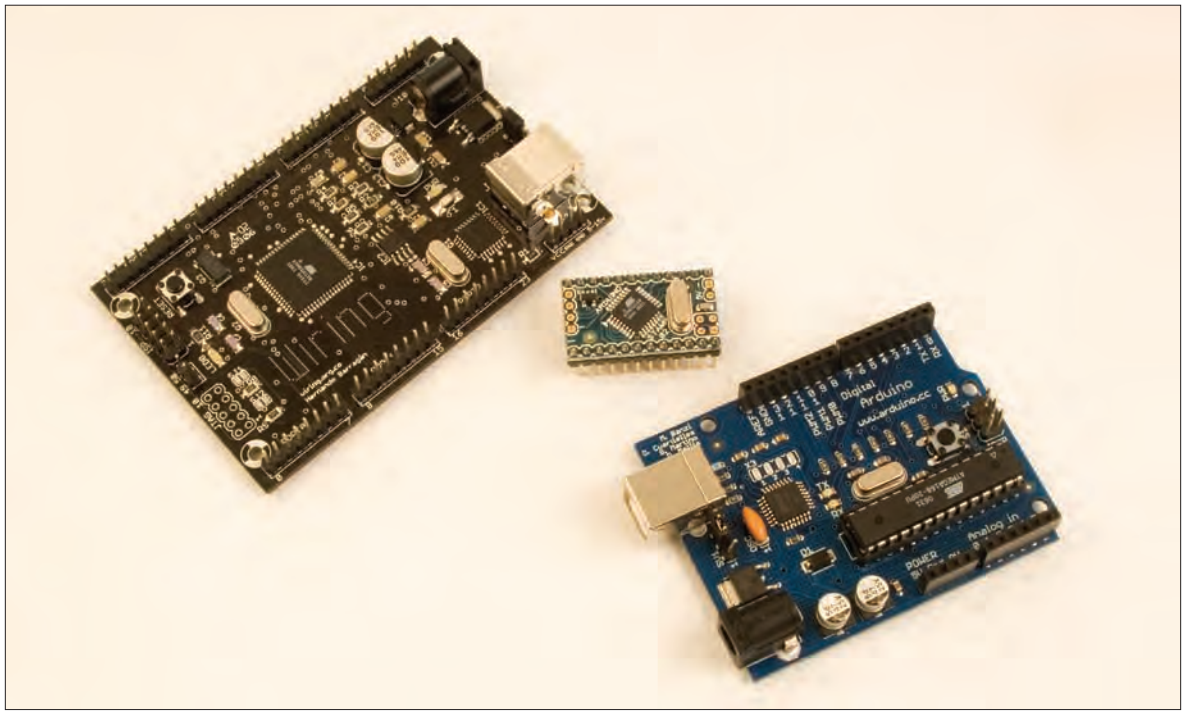
Figure 1-7

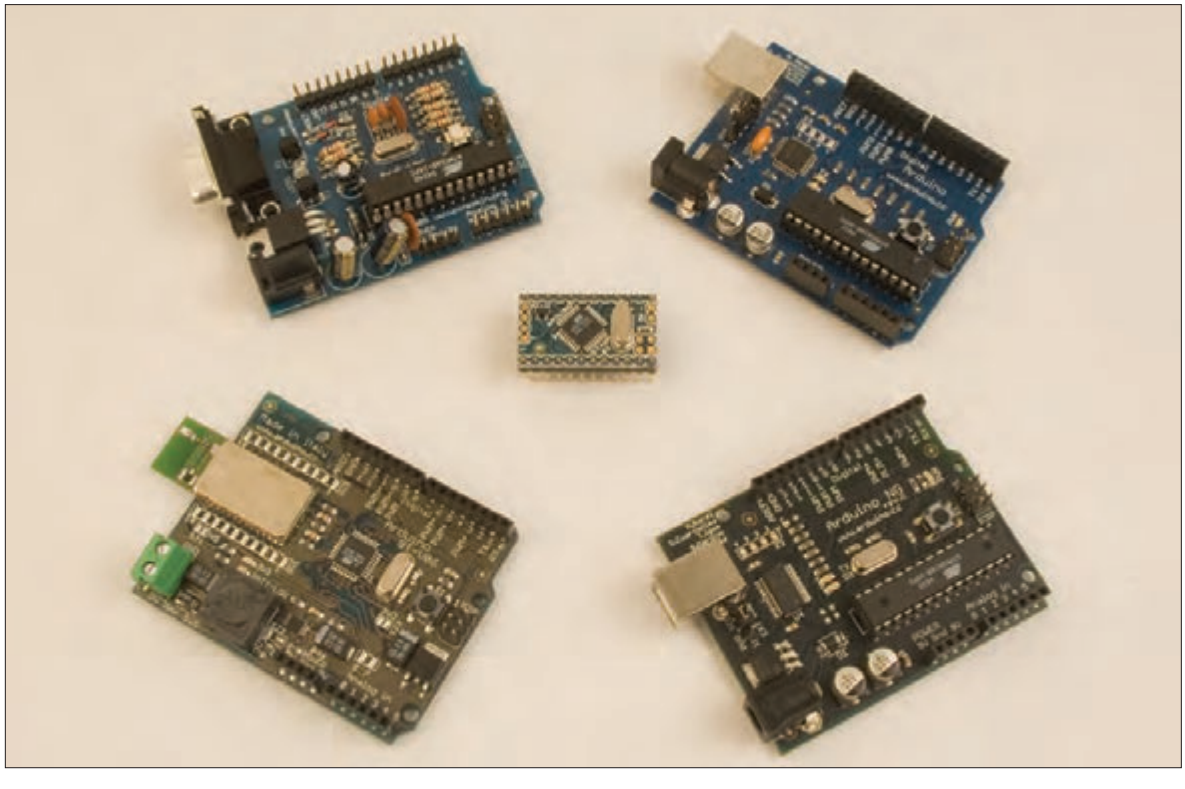
Wiring Board, Arduino NG board, Arduino Mini.

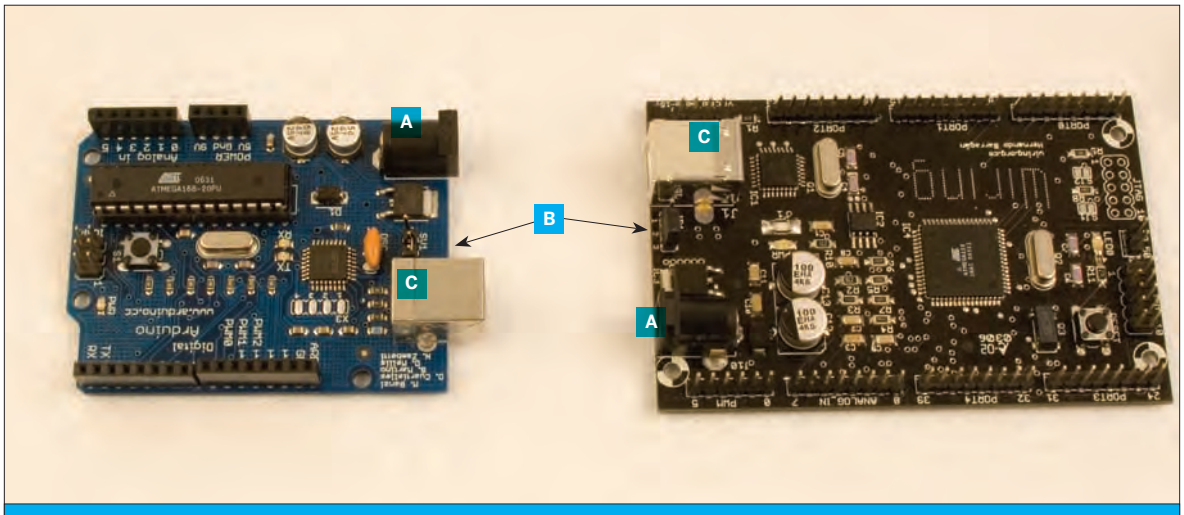
» opposite page bottom

Figure 1-8

The Arduino microcontroller modules.
CLOCKWISE FROM TOP LEFT: the original Arduino serial module; the ArduinoUSB; the Arduino NG; the Arduino Bluetooth; and finally, the Arduino Mini, center.


wiring
Digital
Arduino
www.arduino.cc
POWER
Analog In





A. Power B. Power Jumper C. Serial (via USB)

Wiring and Arduino Compared

Given the similarities between Wiring and Arduino, you're probably wondering which to choose. The programming language is the same for both, and the programming environments are virtually identical, so the major factors to consider are price, size, and number of inputs and outputs.

Wiring is the larger of the two modules, and the more expensive. It has more input and output connections and some useful features such as hardware interrupt pins and two hardware serial ports. Two serial ports can be handy when you're working on projects in this book, because you can use one serial port to talk to your communications device, and another to talk to the computer on which you're programming the microcontroller. There is a software serial library for both Wiring and Arduino that allows you to use any two I/O pins as a serial port. It's more limited than a hardware serial port, in that it can't send and receive data as quickly as a hardware serial port.

Wiring boards can be ordered online from www.sparkfun.com or directly from www.wiring.org.co.

Arduino is the less expensive of the two modules, and the smaller. It has fewer inputs and outputs than Wiring, and only one hardware serial port. The Arduino developers have made a few different Arduino boards. The original board has an RS-232 serial interface, and all the components are large enough that you can solder them by hand. It was designed for people who want to make their own board from scratch. The Arduino USB board is the default board. It's not as easy to assemble by hand, but most people buy them pre-assembled. It has a USB interface. The Arduino Bluetooth board is a variant on the USB board that has a wireless interface for programming and serial communication. It's the most expensive of the Arduino models to date, but handy if you know you're going to connect to it all the time through Bluetooth. The Arduino Mini is a tiny version of the Arduino, suitable for use on a breadboard. For people familiar with the Parallax BASIC Stamp 2 or the NetMedia BX-24, the Mini is a comfortable alternative. You can also build an Arduino module on a solderless breadboard.

Arduino also features add-on modules called shields, which allow you to add pre-assembled circuits to the main module. At this writing, there are four shields on the market. PCB Europe (pcb-europe.net/catalog) sells a board for controlling DC motors, and a prototyping shield for making your own circuits. SparkFun (www.sparkfun.com) sells a breadboard prototyping shield along with the various Arduino boards. Libelium (www.libelium.com) sells a ZigBee radio shield.

The projects in this book can be built with other microcontrollers as well. Like all microcontrollers, the Arduino and Wiring modules are just small computers. Like every computer, they have inputs, outputs, a power supply, and a communications port to connect to other devices. You can power these modules either through a separate power supply or through the USB connection to your computer. The jumper shown in Figure 1-9 switches power from the external supply to the USB supply. For this introduction, you'll power the module from the USB connection. For many projects, you'll want to disconnect them from the computer once you're finished programming them. To do this, you'll need to switch the power jumper to power the board from the external power supply.

Both Wiring and Arduino have four power pins. On the Wiring board, they're labeled 5V, Gnd, GND and 9-15V. On the Arduino, they're labeled 5V, Gnd, Gnd, and 9V. In both cases, the 5V connection outputs 5V relative to the two ground pins. The 9V or 9-15V pin is connected directly to the voltage input on the external power jack, so the output voltage of that pin is equal to whatever your input voltage is. You can also use this connection to connect these modules directly to 9-15V battery power, if you set the power jumper to external power.

Figure 1-10 shows the inputs and outputs for the Arduino, the Arduino Mini, and the Wiring module. Each module has the same standard features that most microcontrollers have: analog inputs, digital inputs and outputs, and power and ground connections. Some of the I/O pins can also be used for serial communication. The Wiring and Arduino boards also have a USB connector, a programming header to allow you to reprogram the firmware (you'll never do that in this book), and a reset button. The Arduino Mini does not have these features, but they can be added using its companion USB-to-serial board. Figure 1-11 shows a typical breadboard setup for the Mini. You'll see these diagrams repeated frequently, as they are the basis for all of the microcontroller projects in the book.

Getting Started

Because the installation process for Wiring and Arduino is almost identical, I'll detail only the Arduino process here.

« opposite page

Figure 1-9

Arduino and Wiring modules. Note the jumper to switch power from the USB connection to an external power supply.

Wiring users will find things similar enough to follow along and do the same steps, substituting "Wiring" for "Arduino" in the instructions that follow.

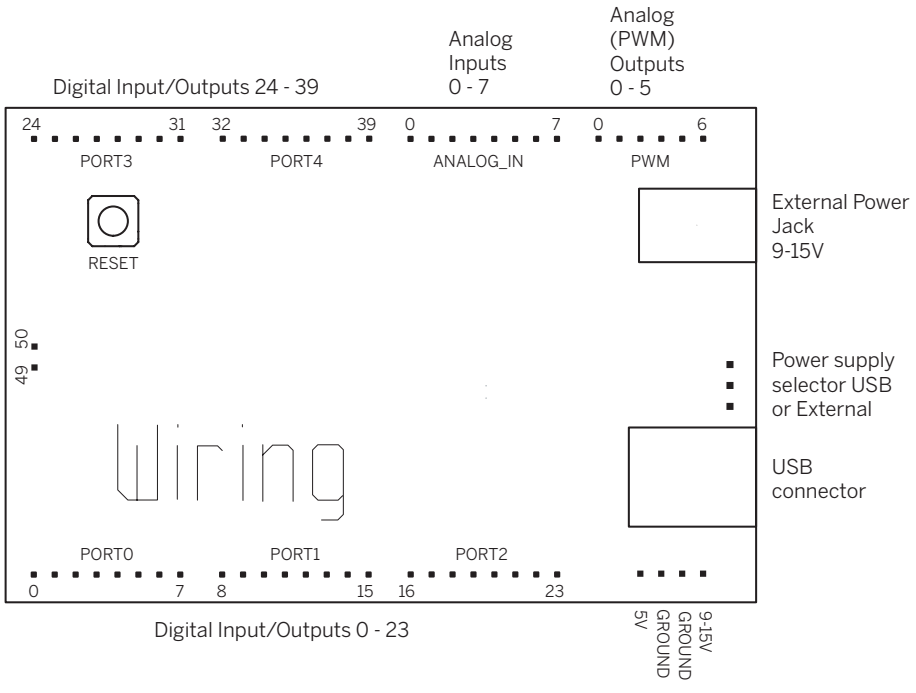
Once you've downloaded the Arduino software, you'll need to do a bit of configuring to get things ready for use. Expand the downloaded file and you'll get a directory called **arduino-0009** (if there is a newer version of the software available, the number will be different). Move this somewhere convenient: on a Mac, you might put it in your **Applications** directory; on Windows, maybe in **C:\Program Files**; on Linux, you might want to keep it in your home directory or drop it into **/usr/local**. Now navigate to the directory **arduino-009/drivers** subdirectory. In that directory, you'll find an installer for the FTDI USB serial driver (not needed under Linux). This is the USB device on the module that allows your computer to communicate with the module via USB. Install it. Macintosh users will also find a file in the **arduino-0009** directory called **macosx_setup.command**. This is the same as the **macosx_setup.command** for Processing that was described earlier, so if you already ran it to configure Processing, you won't need to do it again. If you haven't, double-click the file and follow the instructions that come up.

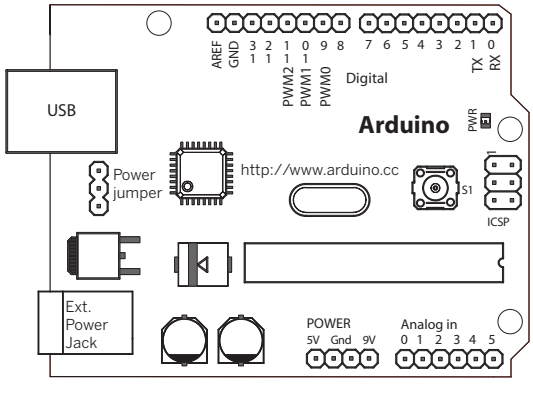
Arduino and Wiring are new to the market, and updates to their software occur frequently. The notes in this book refer to Arduino version 0009 and Wiring version 0012. By the time you read this, the specifics may be slightly different, so check the Arduino and Wiring websites for the latest details.

Now you're ready to launch Arduino. Connect the module to your USB port, and double-click the Arduino icon to launch the software. The editor looks like Figure 1-12.

The environment is based on Processing, and has the same New, Open, Save, and Export buttons on the main toolbar. In Arduino and Wiring, the Run function is called Verify. It compiles your program to check for any errors, and the Export function is called Upload to Module instead. It uploads your code to the microcontroller module. There's an additional button, the Serial Monitor, that you can use to receive serial data from the module while you're debugging.

X





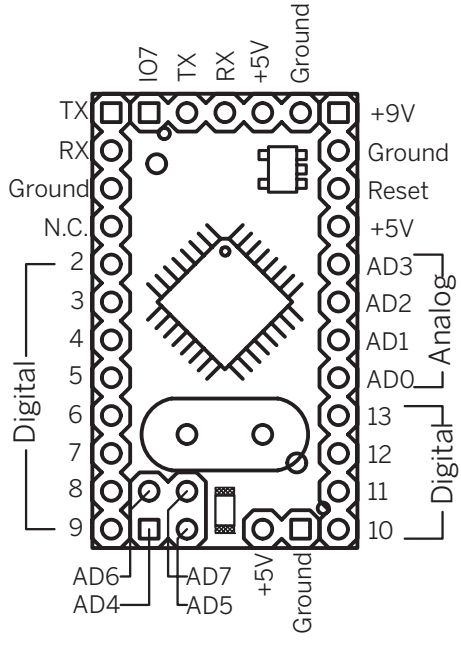
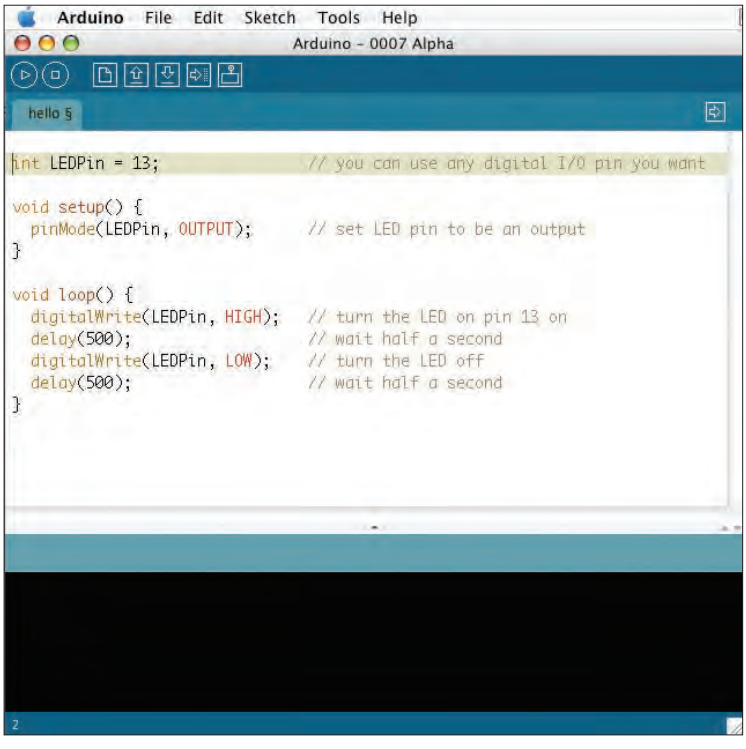


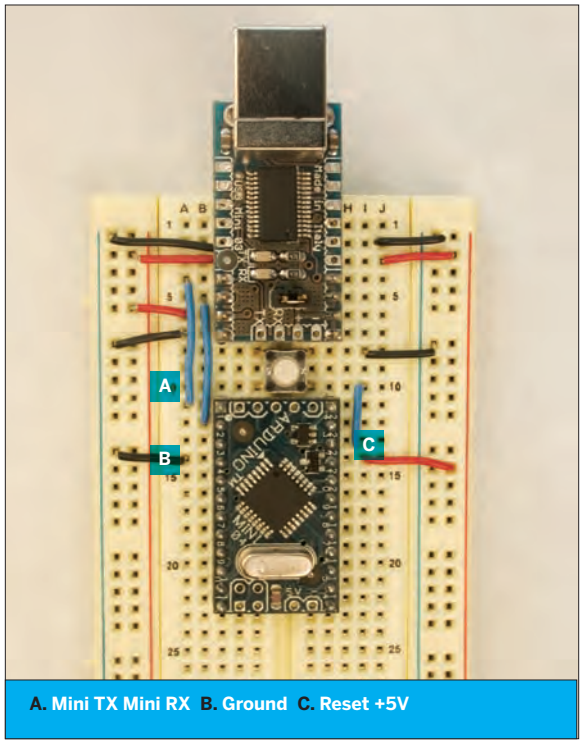
Figure 1-10
Wiring, Arduino NG, and Arduino Mini pin diagrams.

Figure 1-12
The Arduino programming environment. The Wiring environment looks identical. to this, except for the color.



» bottom left

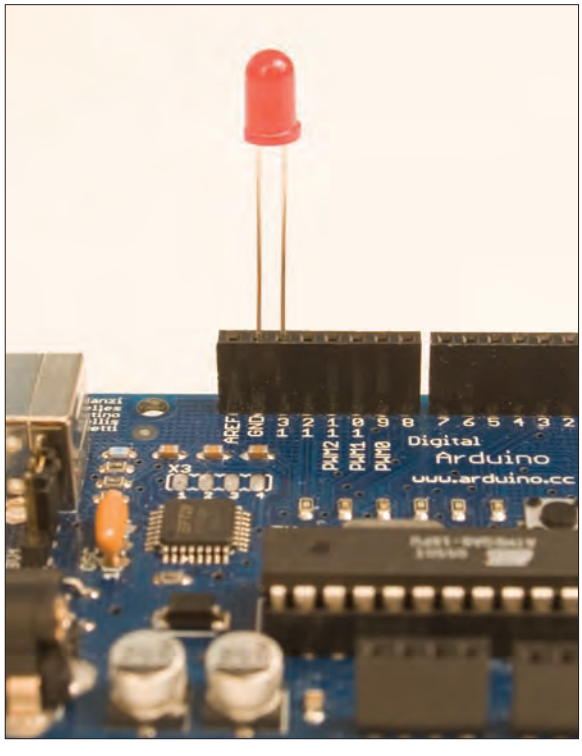
Figure 1-11
Typical wiring for an Arduino Mini.



A. Mini TX Mini RX B. Ground C. Reset +5V

» bottom right

Figure 1-13
LED connected to pin 13 of an Arduino board .



Try It

Here's your first program:

```
/*  Blink
    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
}
```

In order to see this run, you'll need to connect an LED from pin 13 of the board to ground (GND) as shown in Figure 1-13. The positive (long) end of the LED should go to 13, and the short end to ground.

Then type the code into the editor. Click on Tools→Serial Port to choose the serial port of the Arduino module. On the Mac or Linux, the serial port will have a name like /dev/tty.usbserial-1B1 (the letters and numbers after the dash will be slightly different each time you connect it). On Windows, it should be COMx, where x is some number (for example, COM5).

Next, select the model of AVR microcontroller on your Arduino or Wiring module (you'll have to inspect the board to determine this). It will be either ATmega8 or ATmega168. Make the appropriate choice from the Tools→Microcontroller (MCU) menu.

NOTE: On Windows, COM1–COM4 are generally reserved for built-in serial ports, whether or not your computer has them.

Once you've selected the port and model, click Verify to compile your code. When it's compiled, you'll get a message at the bottom of the window saying Done compiling. Then press the reset button on the module to reset it and prepare it to accept a new program. Then click Upload. This will take several seconds. Once it's done, you'll get a message saying Done uploading, and a confirmation message in the serial monitor window that says:

```
Atmel AVR ATmega168 is found.
Uploading: flash
```

NOTE: If your Arduino uses an ATmega8, it will report that instead. You must make sure that you have configured the Arduino to use the model of ATmega microcontroller on your board.

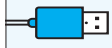
Press the reset button on the module again, and after about five seconds, the LED you wired to the output pin will begin to blink. That's the microcontroller equivalent of "Hello World!" (If you're using an Arduino Diecimila or later model, you won't have to press the reset button when you upload.)

NOTE: If it doesn't work, you might want to seek out some external help. The Arduino (www.arduino.cc/cgi-bin/yabb2/YaBB.pl) and Wiring (wiring.org.co/cgi-bin/yabb/YaBB.pl) forums are full of helpful people who love to hack these sort of things.

Serial Communication

One of the most frequent tasks you'll use a microcontroller for in this book is to communicate serially with another device, either to send sensor readings over a network or to receive commands to control motors, lights, or other outputs from the microcontroller. Regardless of what device you're communicating with, the commands you'll use in your microcontroller program will be the same. First you'll configure the serial connection for the right data rate. Then you'll read bytes in, write bytes out, or both, depending on what device you're talking to, and how the conversation is structured.

NOTE: If you've got experience with the Basic Stamp or PicBasic Pro, you will find Arduino serial communications a bit different than what you are used to. In PBasic and PicBasic Pro, the serial pins and the data rate are defined each time you send a message. In Wiring and Arduino, the serial pins are unchangeable, and the data rate is set at the beginning of the program. This way is a bit less flexible than the PBasic way, but there are some advantages, as you'll see shortly.



Where's My Serial Port?

The USB serial port that's associated with the Arduino or Wiring module is actually a software driver that loads every time you plug in the module. When you unplug, the serial driver deactivates and the serial port will disappear from the list of available ports. You might also notice that the port name changes when you unplug and plug in the module. On Windows machines, you may get a new COM number. On Macs, you'll get a different alphanumeric code at the end of the port name.

Never unplug a USB serial device when you've got its serial port open; you must exit the Wiring or Arduino software environment before you unplug anything. Otherwise, you're sure to crash the application, and possibly the whole operating system, depending on how well-behaved the software driver is.

Try It

This next Arduino/Wiring program listens for incoming serial data. It adds one to whatever serial value it receives, and sends the result back out. It also blinks an LED on pin regularly, on the same pin as the last example, to let you know that it's still working:

```
/*
    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:
```

Continued from previous page.

```
    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
  }
}
```

To send bytes from the computer to the microcontroller module, first compile and upload this program. Then click the Serial Monitor icon (the rightmost icon on the toolbar). The screen will change to look like Figure 1-14. Set the serial rate to 9600 baud.

Type any letter in the text entry box and press Enter or click Send. The module will respond with the next letter in sequence. For every character you type, the module adds one to that character's ASCII value, and sends back the result. Terminal applications represent all bytes they receive as ASCII.

Wiring Components to the Module

The Arduino and Wiring modules don't have many sockets for connections other than the I/O pins, so you'll need to keep a solderless breadboard handy to build subcircuits for your sensors and actuators (output devices). Figure 1-15 shows a standard setup for connections between the two.

Specialty Devices

You'll encounter some specialty devices as well, such as the Lantronix Xport, WiPort, and Cobox Micro. The Lantronix modules are serial-to-Ethernet modules. Their main function is to connect devices with a serial communications interface (such as all microcontrollers) to Ethernet networks. It's possible to program your own serial-to-Ethernet module directly on a microcontroller with a few spare parts, but it's a lot of work. The Lantronix modules cost more, but they're much more convenient. You'll also encounter serial-to-Bluetooth modules, serial-to-ZigBee modules, RFID modules, and other microcontrollers whose main job is to connect other devices. The details on connecting these will be explained one by one as you encounter them in the projects that follow.

Basic Circuits

There are two basic circuits that you'll use a lot in this book: digital input and analog input. If you're familiar with microcontroller development, you're already familiar with them. Any time you need to read a sensor value, you can start with one of these two. Even if you're using a custom sensor in your final object, you can use these circuits as placeholders, just to see any changing sensor values.

Digital input

A digital input to a microcontroller is nothing more than a switch. The switch is connected to voltage and to a digital input pin of the microcontroller. A high-value resistor (10 kilohms is good) connects the input pin to ground. This is called a pull-down resistor. Other electronics tutorials may connect the switch to ground and the resistor to voltage. In that case, you'd call the resistor a pull-up resistor. Pull-up and pull-down resistors provide a reference to power (pull-up) and ground (pull-down) for digital input pins. When a switch is wired as shown in Figure 1-16, closing the switch sets the input pin high. Wired the other way: closing the switch sets the input pin low.

The circuit in Figure 1-17 is called a voltage divider. The variable resistor and the fixed resistor divide the voltage between them. The ratio of the resistors' values deter-

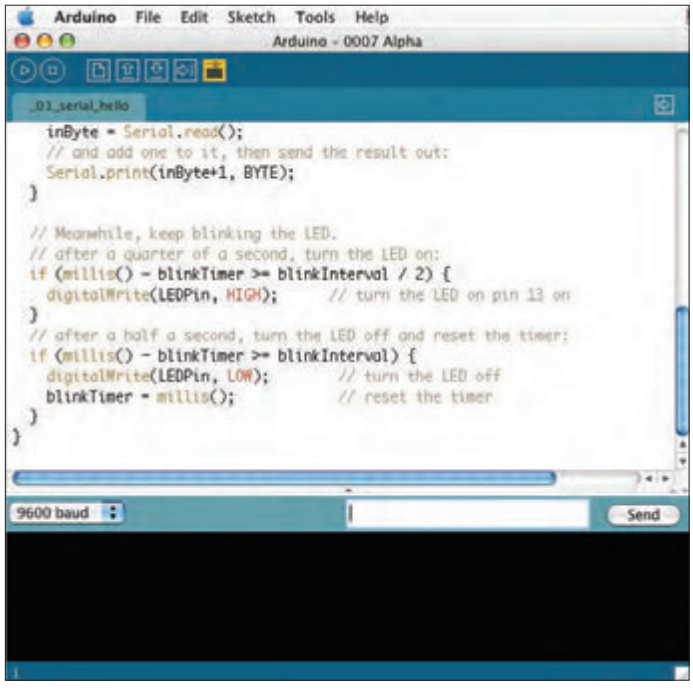


Figure 1-14
The Serial monitor in Arduino.

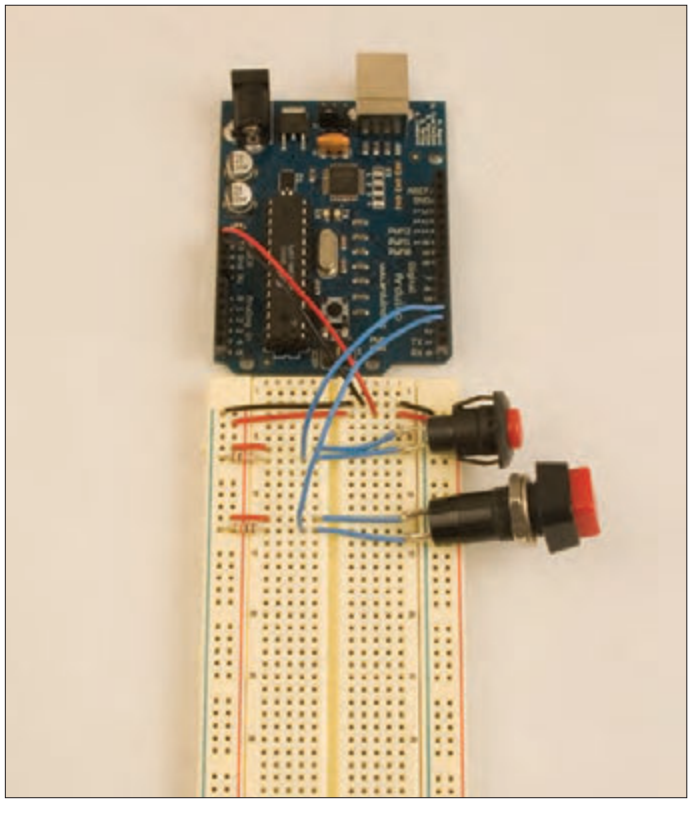


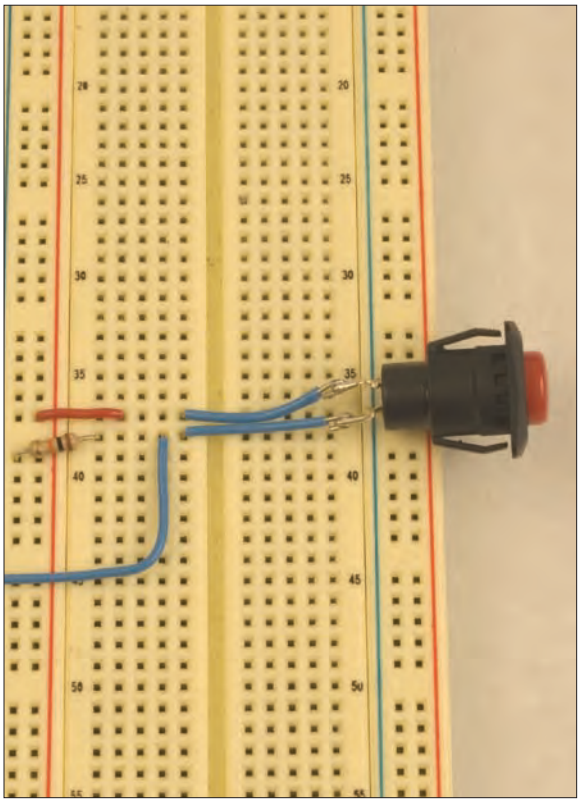
Figure 1-15
Arduino connected to a breadboard. +5V and ground run from the module to the long rows of the board. This way, all sensors and actuators can share the +5V and ground connections of the board. Control or signal connections from each sensor or actuator run to the appropriate I/O pins. In this example, two pushbuttons are attached to digital pins 2 and 3 as digital inputs.

mines the voltage at the connection between them. If you connect the analog-to-digital converter of a microcontroller to this point, you'll see a changing voltage as the variable resistor changes. You can use any kind of variable resistor: photocells, thermistors, force-sensing resistors, flex-sensing resistors, and more.

The potentiometer, shown in Figure 1-18, is a special type of variable resistor. It's a fixed resistor with a wiper that slides along the conductive surface of the resistor. The resistance changes between the wiper and both ends of the resistor as you move the wiper. Basically, a potentiometer (pot for short) is two variable resistors in one package. If you connect the ends to voltage and ground, you can read a changing voltage at the wiper.

Most of the circuits in this book will be shown on a breadboard. By default, the two side rows on each side of the board will be used for power and ground lines, typically +5V for power. On most of the boards, you'll notice wires connecting each of the side rows to two of the top rows. For some projects, the board will be powered from a Wiring or Arduino module or USB power, so there will be no need for a voltage regulator. For others, you will need one. I use separate wires rather than connecting from one side to the other directly, so that when I need a voltage regulator, it can be added easily. Figure 1-19 shows a board with and without a regulator.

There are many other circuits you'll learn in the projects that follow, but these are the staples of all the projects.
X



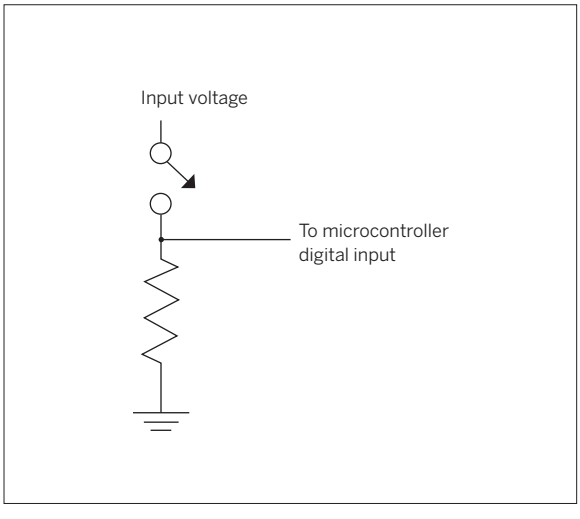
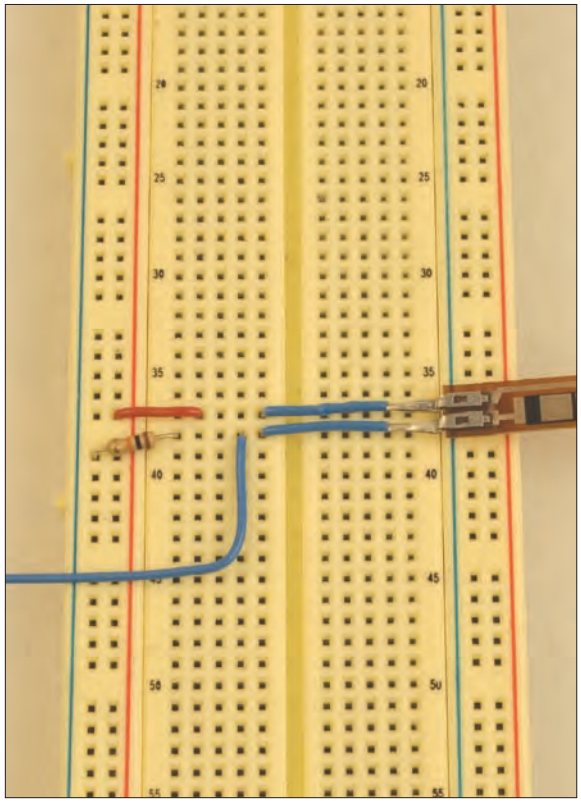


Figure 1-16
Digital input to a microcontroller.



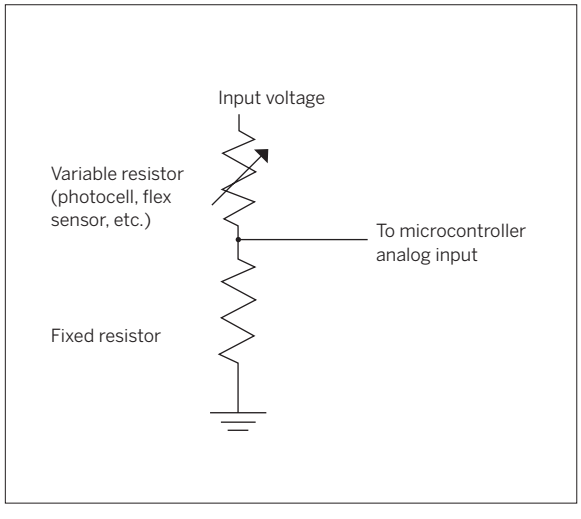
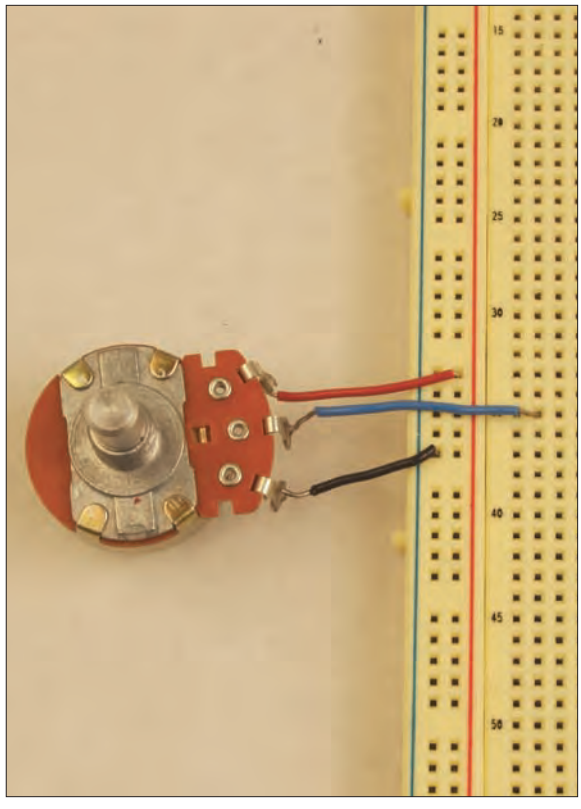


Figure 1-17
Voltage divider used as analog input to a microcontroller.



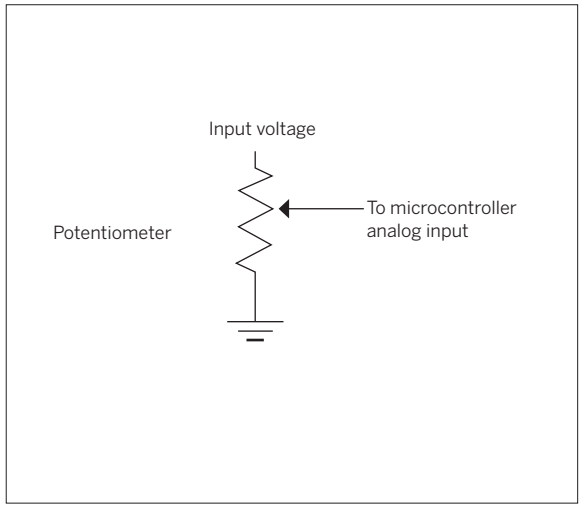


Figure 1-18
Potentiometer used as analog input to a microcontroller.

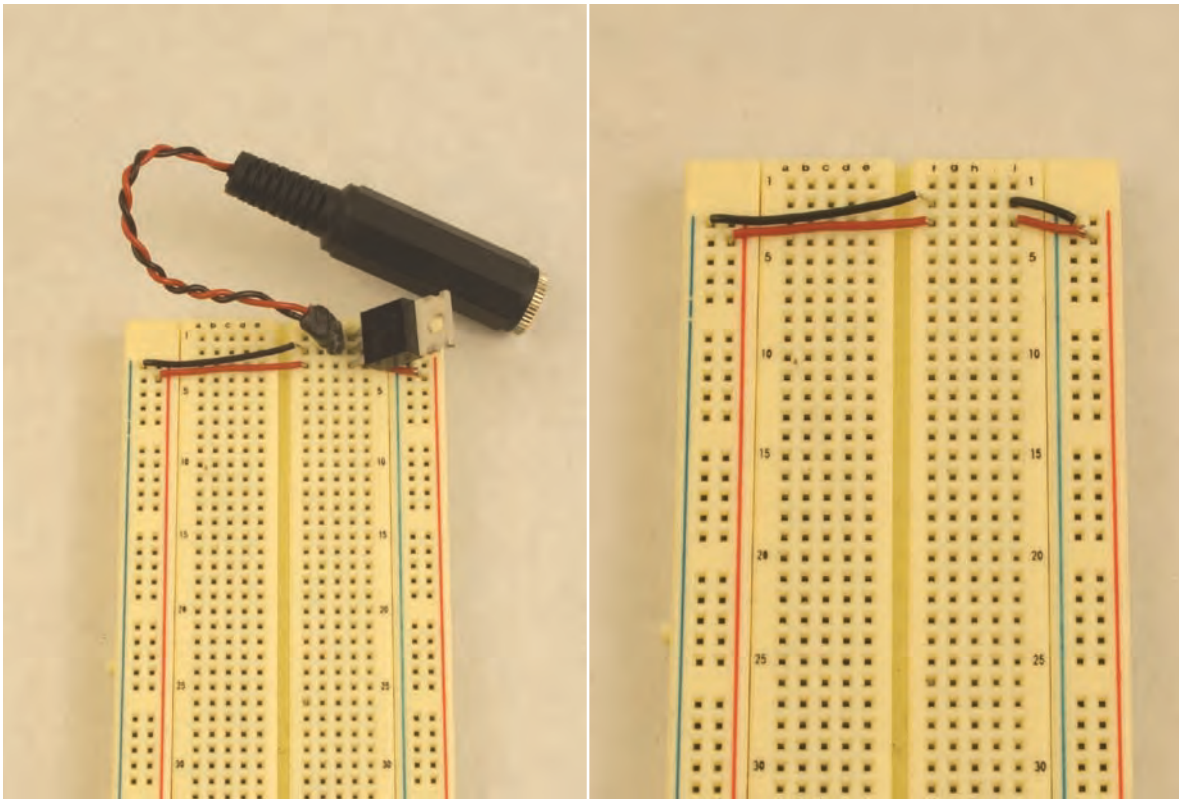


Figure 1-19
Breadboard with a regulator, and without one.

You will run across different variations on many of the modules and components used in this book. For example, the Arduino module has at least five variations, shown in Figure 1-8. The FTDI USB-to-serial module used in later chapters has at least three variations. Even the voltage regulators used in this book have different variations. Be sure to check the data sheet on whatever component or module you're using, as your version may vary from what is shown here.

It Ends with the Stuff You Touch

Though most of this book is about the fascinating world of making things talk to each other, it's important to remember that you're most likely building your project for the enjoyment of someone who doesn't care about the technical details under the hood.

Even if you're building it only for yourself, you don't want to have to fix it all the time. All that matters to the person using your system are the parts that she can see, hear, and touch. All the inner details are irrelevant if the physical interface doesn't work. So don't spend all of your time focusing on the communication between devices and leave out the communication with people. In fact, it's best to think about the specifics of what the person does and sees first.

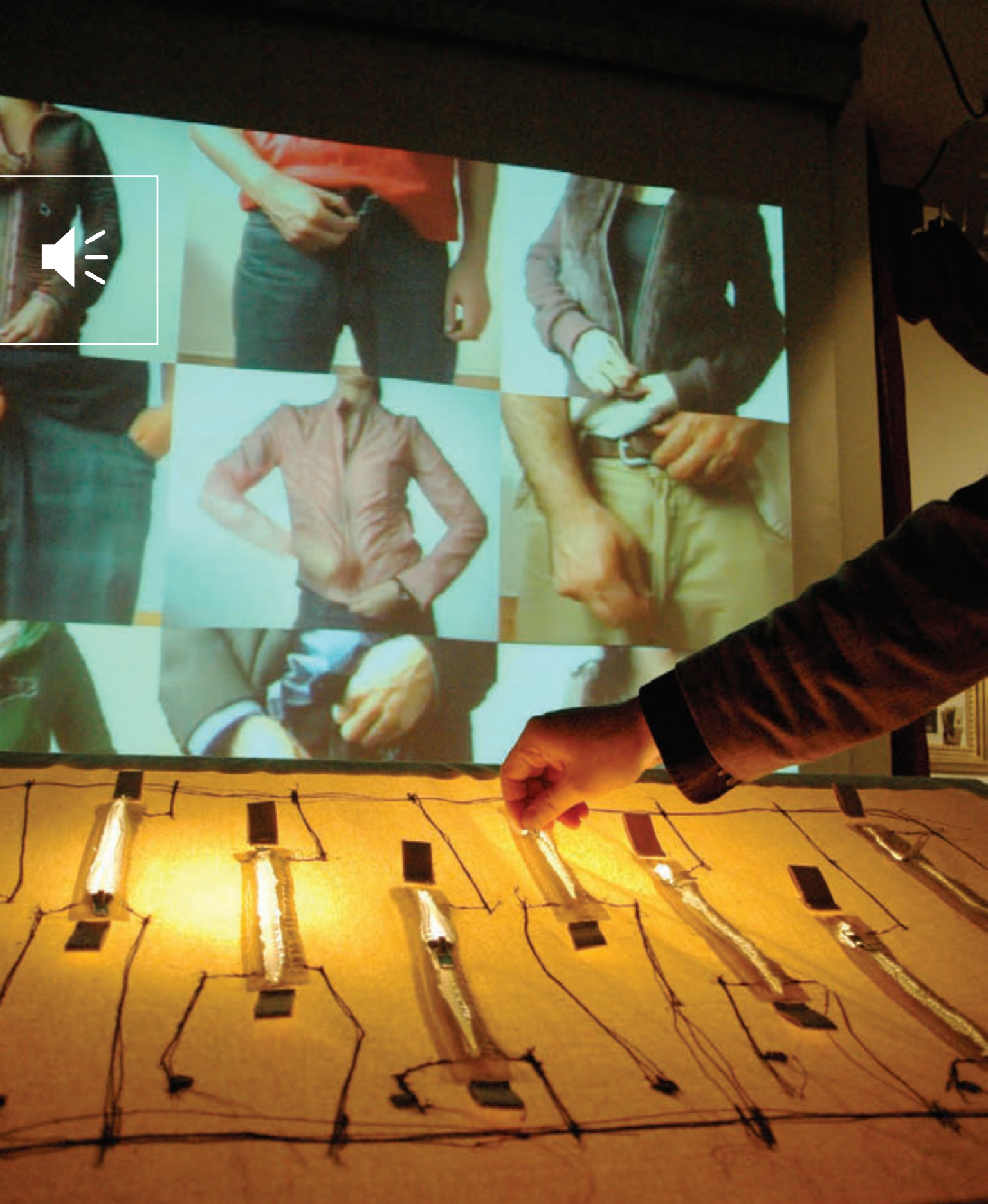
There are a number of details that are easy to overlook, but are very important to humans. For example, many network communications can take several seconds or more. In a screen-based operating system, progress bars acknowledge a person's input and keep her informed as to the progress of the task. Physical objects don't have progress bars, but they should incorporate some indicator as to what they're doing — perhaps as simple as an LED that gently pulses while the network transfer's happening, or a tune that plays.

Find your own solution, but make sure you give some physical indication as to the invisible activities of your objects.

Don't forget the basic elements, either. Build in a power switch or a reset button. Don't forget a power indicator. Design the shape of the object so that it's clear which end is up. Make your physical controls clearly visible and easy to operate. Plan the sequence of actions you expect a person to take, and lay out the physical affordances for those actions in a sensible sequence. You can't tell people what to think about your object — you can only show them how to interact with it through its physical form. There may be times when you violate convention in the way you design your controls, perhaps in order to create a challenging game, or to make the object seem more "magical," but make sure you're doing it intentionally. Always think about the participant's expectations first.

By including the person's behavior in your system planning, you solve some problems that are computationally difficult, but easy for human intelligence to solve. Ultimately, the best reason to make things talk to each other is to give people more reasons to talk to each other.

X



2

MAKE: PROJECTS

The Simplest Network

The most basic network is a one-to-one connection between two objects. This chapter covers the details of two-way communication, beginning with the characteristics that have to be agreed upon in advance. You'll learn about some of the logistical elements of network communications: data protocols, flow control, and addressing. You'll practice all of this by building a simple example: one-to-one serial communication between a microcontroller and a personal computer. Once you've got that working, you'll replace the cable connecting the two with Bluetooth radios and learn about modem communications.

◂◂ **Joo Youn Paek's Zipper Orchestra (2006)** is a musical installation that lets you control video and music using zippers. The zippers are wired to a microcontroller using conductive thread, and the microcontroller communicates serially with a multimedia computer that drives the playback of the zipper movies and sounds as you zip.
Photo courtesy of Joo Youn Paek.

Layers of Agreement

Before you can get things to talk to each other, you have to lay some ground rules for the communication between them. These agreements can be broken down into five layers, each of which builds on the previous ones:

- **Physical**
How are the physical inputs and outputs of each device connected to the other? How many connections between the two devices do you need to get messages across?

- **Electrical**
What voltage levels will you send to represent the bits of your data?

- **Logical**
Does an increase in voltage level represent a zero or a one? This is one of the most common sources of problems in the projects that follow.

- **Data**
What's the timing of the bits? Are the bits read in groups of 8, 9, or 10 bits? More? Are there bits at the beginning or end of each group to punctuate the groups?

- **Application**
How are the groups of bits arranged into messages? What is the order in which messages have to be exchanged in order to get something done?

This is a simplified version of a common model for thinking about networking called the Open Systems Interconnect (OSI) model. Networking issues are never really this neatly separated, but if you keep these elements distinct in your mind, troubleshooting any connection will be much easier. Thinking in layers like this gives you somewhere to start looking for the problem, and a way to eliminate parts of the system that are *not* the problem.

No matter how complex the network gets, never forget that the communication between electronic devices is all about pulses of energy. Serial communication involves changing the voltage of an electrical connection between the sender and receiver at a specific rate. Each interval of time represents one bit of information. The sender changes the voltage to send a value of 0 or 1 for the bit in question, and the receiver reads whether the voltage is high or low. There are two methods (see Figure 1-1) that sender and receiver can use to agree on the rate at which bits are sent. In asynchronous serial communication, the rate is agreed upon mutually and clocked independently by sender and receiver. In synchronous serial communication, it's controlled by the sender, who pulses a separate connection high and low at a steady rate. Synchronous serial communication is used mostly for communication between integrated circuits (such as the communication between a computer processor and its memory chips). The rest of this chapter concentrates only on asynchronous serial communication, because that's the form of serial communication underlying the networks in the rest of the book.

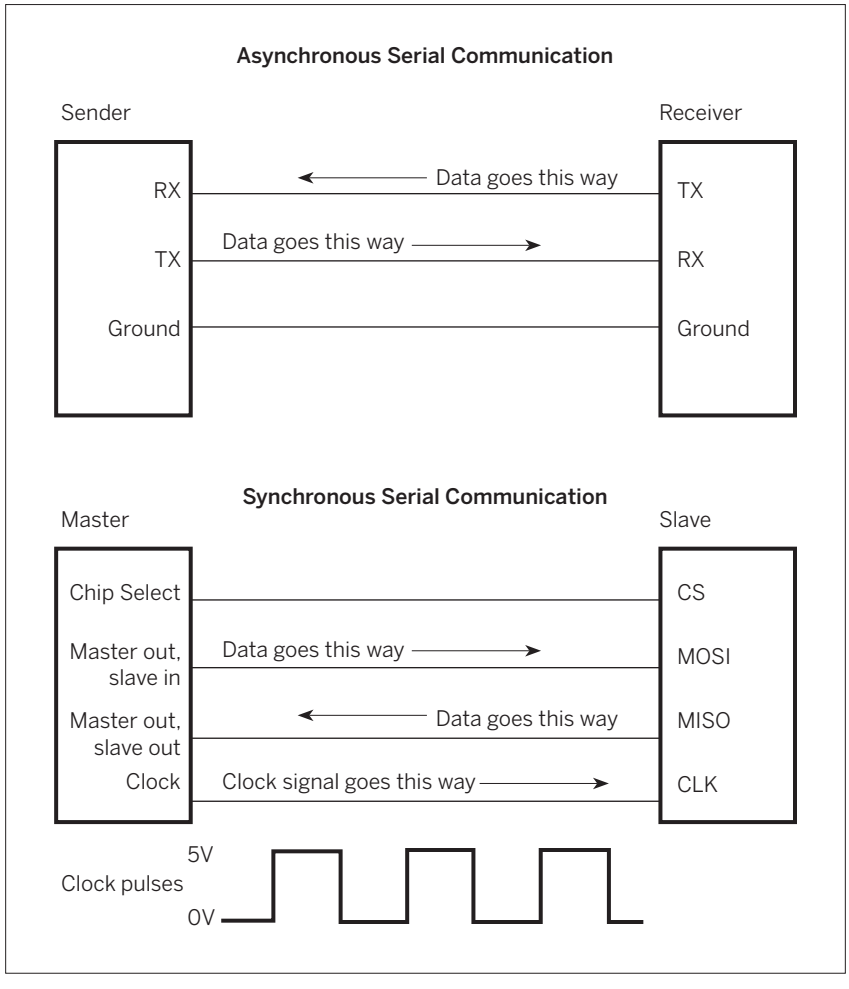
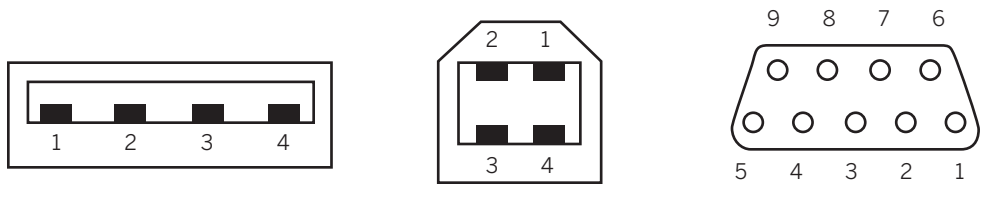


Figure 2-1
Types of serial communication.

Figure 2-2
Physical connections: USB, RS-232 serial.



USB Type A

USB Type B

RS-232

1 – +5V
2 – Data -
3 – Data +
4– Ground

2 – PC Receive
3 – PC Ttransmit
5 – PC Ground

Making the Connection: The Lower Layers

You're already familiar with one example of serial communication, between a microcontroller and a personal computer. In Chapter 1, you connected an Arduino module to a personal computer through the computer's USB port. If you're working with a different microcontroller such as Parallax' Basic Stamp, you probably made the connection using a serial-to-USB converter, or used an older PC that still had a 9-pin serial port. That simple connection involved two serial protocols.

First, there's the protocol that the microcontroller speaks, called TTL serial:

- **Physical layer**
The Arduino module receives data on digital I/O pin 0, and sends it out on pin 1.

- **Electrical layer**
It uses pulses of 5 volts or 0 volts to represent bits.

- **Logical layer**
A 5-volt signal represents the value 1, and a 0-volt signal represents the value 0.

- **Data layer**
Data is sent at 9600 bits per second. Each byte contains 8 bits, preceded by a start bit and followed by a stop bit (which you never have to bother with).

- **Application layer**
At the application layer, you sent one byte from the PC to the Arduino and processed it, and the Arduino sent back one byte to the PC.

But wait, that's not all that's involved. The 5-volt and 0-volt pulses didn't go directly to the PC. First they went to a serial-to-USB chip on the board that communicates using TTL serial on one side, and USB on the other.

Second, there's USB, the Universal Serial Bus protocol. It differs from TTL serial in many ways:

- **Physical layer**
USB sends data on two wires, called Data+ and Data–. Every USB connector also has a 5-volt power supply line and a ground line.

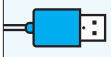
- **Electrical layer**
The signal on Data– is always the polar opposite of what's on Data+, so that the sum of their voltages is always zero. Because of this, a receiver can check for electrical errors by adding the two data voltages together. If the sum isn't zero, the receiver can disregard the signal at that point.

- **Logical layer**
A +5-volt signal (on Data+) or –5-volt signal (on Data–) represents the value 1, and a 0-volt signal represents the value 0.

- **Data Layer**
The data layer of USB is more complex than TTL serial. Data can be sent at up to 480 megabits per second. Each byte contains 8 bits, preceded by a start bit and followed by a stop bit. Many USB devices can share the same pair of wires, sending signals at times dictated by the controlling PC. This arrangement is called a bus (the B in USB). As there can be many devices on the same bus, the operating system gives each one its own unique address, and sees to it that the bytes from each device on the bus go to the applications that need them.

- **Application layer**
At the application layer, the USB-to-serial converter on the Wiring and Arduino boards sends a few bytes to the operating system to identify itself. The operating system then associates the hardware with a library of driver software that other programs can use to access data from the device.

All that control is transparent to you, because the computer's USB controller only passes you the bytes you need. The USB chip on your Arduino board presents itself to the operating system as a serial port, and sends data through the USB connection at the rate you choose (9600 bits per



USB: An Endless Source of Serial Ports

One of the great things about microcontrollers is that because they're cheap, you can use many of them.

For example, in a project with many sensors, you can either write a complex program on the microcontroller to read them all, or you can give each sensor its own microcontroller. If you're trying to get all the information from those sensors into a personal computer, you might think it's easier to use one microcontroller, because you've got a limited number of serial ports. Thanks to USB, however, that's not the case. If your microcontroller speaks USB, or if you've got a USB-to-serial adaptor for it, you can just plug it in and it will show up in the operating system as another serial port.

For example, if you plug three Arduino modules into the same computer through a USB hub, you'll get three new serial ports, named something like this on Mac OS X:

```
/dev/tty.usbserial-5B21
/dev/tty.usbserial-5B22
/dev/tty.usbserial-5B24
```

In Windows, you'd see something like COM8, COM9, COM10.

If you're using a microcontroller that doesn't have its own USB-to-serial converter, you can buy one for about $15 to $40 — Keyspan (www.keyspan.com) and IOGear (www.iogear.com) sell decent models. You can get a USB-to-TTL-serial cable from FTDI for about $20 (part number TTL-232R, also available from Mouser.com), and SparkFun sells a breadboard USB-to-serial module for about $15 (part number BOB-00718). The SparkFun module is shown in a circuit in Figure 2-4. The other converters are self-explanatory.

Like the MAX3323 circuit, this circuit is a handy testing circuit for some of the radio and Ethernet modules you'll see in the chapters that follow that have TTL serial interfaces. In fact, it's the default circuit for interfacing these devices to a computer in this book. If your computer doesn't have USB, you can use the MAX3323 circuit instead. You can also use the MAX3323 circuit in conjunction with the commercially available USB-to-serial adaptors mentioned above.

second, in the example in Chapter 1).

One more protocol: if you use a BASIC Stamp or another microcontroller with a non-USB serial interface, you probably have a 9-pin serial connector connecting your microcontroller to your PC, or to a USB-to-serial adaptor. This connector, called a DB-9 or D-sub-9 connector, is a standard connector for another serial protocol, RS-232. RS-232 was the main serial protocol for computer serial connections before USB, and it's still quite common on many computer peripheral devices:

- **Physical layer**
A computer with an RS-232 serial port receives data on pin 2, and sends it out on pin 3. Pin 5 is the ground pin.

- **Electrical layer**
RS-232 sends data at two levels: 5 to 12 volts, and –5 to –12 volts.

- **Logical layer**
A 5 to 12 volt signal represents the value 0, and a –5 to –12 volt signal represents the value 1.

NOTE: Note that this logic is the reverse of TTL serial. It's referred to as inverted logic. Most microcontrollers have the capacity to send serial data using inverted or true logic.

- **Data layer**
This is the same as TTL's, 8 bits per byte with a start and stop bit.

So why is it possible to connect some microcontrollers, like the BASIC Stamp or the BX-24, directly to RS-232 serial ports? It is because the voltage levels of TTL serial, 0 to 5 volts, are just barely enough to register in the higher RS-232 levels, and because you can invert the bits when sending or receiving from the microcontroller. RS-232 doesn't carry any of the addressing overhead of USB, so it's an easier protocol to deal with. Unfortunately, it's becoming obsolete, so USB-to-serial converters are increasingly common tools for microcontroller programmers. Because Wiring and Arduino both have an integrated USB-to-serial converter, you can just plug them into a USB port.

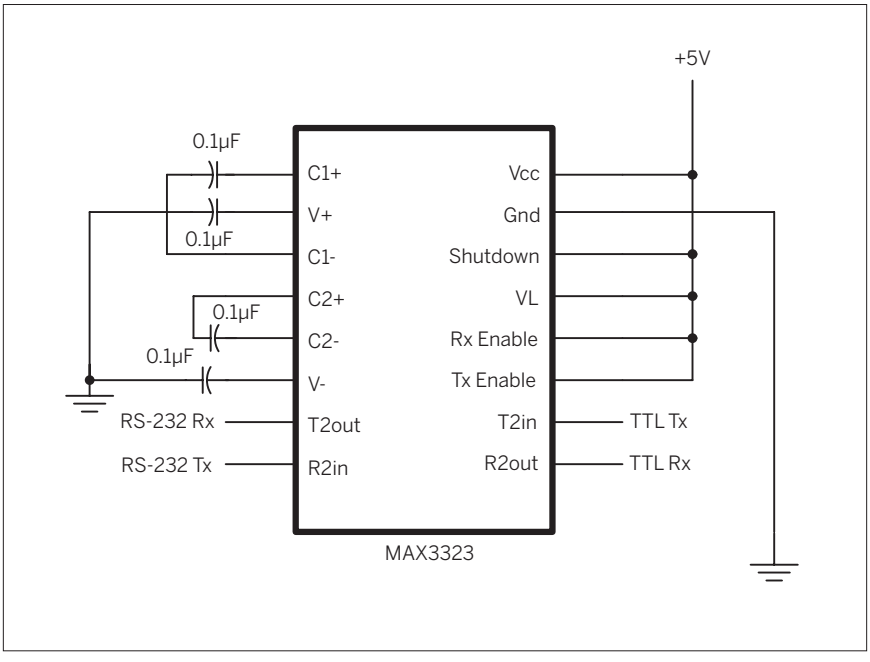
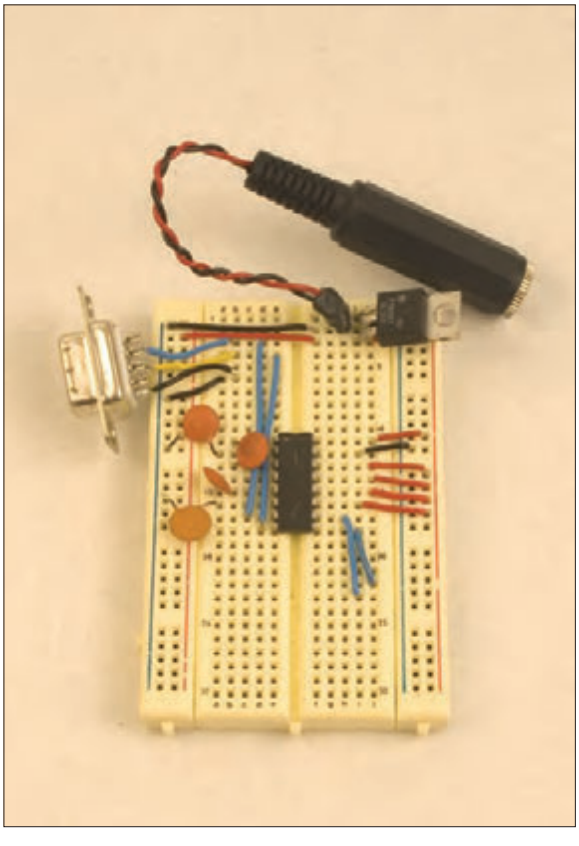


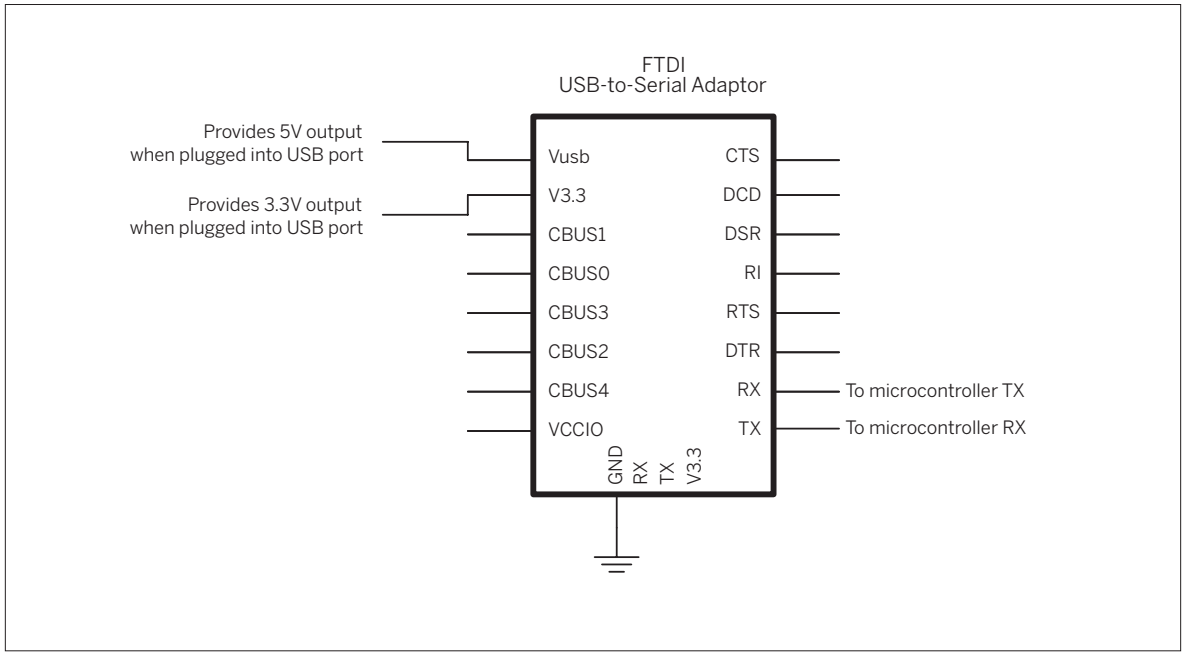
Figure 2-3
The MAX3323 chip. This circuit is really handy when you need to get any 3.3 to 5-volt TTL device to talk to a personal computer with an RS-232 serial port. This will also work for the MAX232.



When you're lucky, you never have to think about this kind of protocol mixing, and you can just use converters to do the job for you. You're not always lucky, though, so it's worth knowing a little about what's happening behind the scenes. For example, one of the most common problems in getting a serial device to communicate with a personal computer is converting the device's serial signals to USB or RS-232. A handy chip that does the TTL-to-RS-232 conversion for you is the MAX3323, available from Maxim Technologies (www.maxim-ic.com). It takes in RS-232 serial, and spits out 3.3V to 5-volt TTL serial, and vice versa. If you power it from a 3.3V source, you get 3.3V TTL serial output, and if you power it from 5V, you get 5V TTL serial output. Figure 2-3 shows the typical schematic for a MAX232 and a MAX3323.

If you've done a lot of serial projects, you may know the MAX232, which preceded the MAX3323. In fact, the MAX232 was so common that the name came to be synonymous for all TTL-to-RS-232 converters, whether Maxim made them or not. The MAX232 worked only at 5 volts, but the MAX3323 works at 3.3 to 5 volts. Because 3.3 volts is beginning to replace 5 volts as a standard supply voltage for electronic parts, it's handy to use a chip that can do both.

X



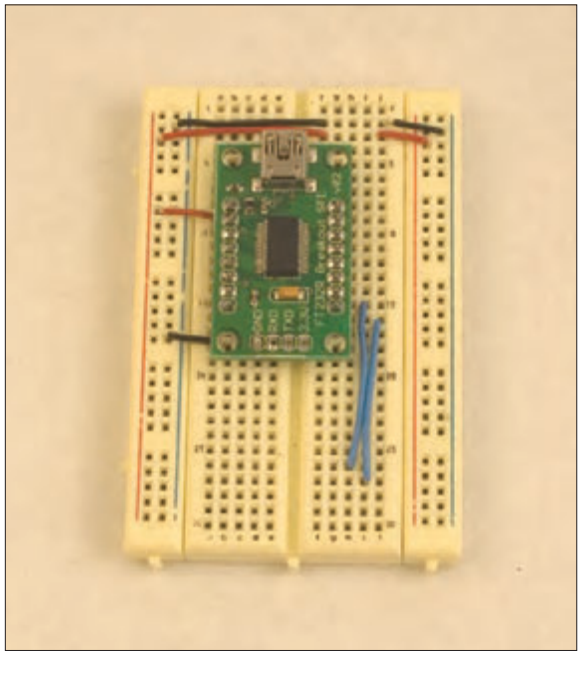


Figure 2-4
SparkFun's FTDI-to-serial module on a breadboard.

Saying Something: The Application Layer

Now that you've got a sense of how to make the connections between devices, let's build a couple of projects to understand how to organize the data sent in order to get things done.

Project 1

Monski pong

In this example, you'll make a replacement for a mouse. If you think about the mouse as a data object, it looks like Figure 2-5.

MATERIALS

- **2 flex sensor resistors** Images SI, Inc, (www.imagesco.com) part number FLX-01, or Jameco (www.jameco.com) part number 150551
- **2 momentary switches** Available from any electronics retailer. Pick the one that makes you the happiest. Jameco part number 174414 is shown here.
- **4 10-kilohm resistors** Available at many retailers, for example Digi-Key (www.digikey.com) part number 10K-QBK-ND, and many others.
- **1 solderless breadboard** For instance, Digi-Key part number 438-1045-ND, Jameco part number 20601
- **1 Arduino microcontroller module**
- **1 personal computer**
- **All necessary converters to communicate serially from microcontroller to computer** For the Arduino and Wiring modules, all you'll need is a USB cable. For other microcontrollers, you'll probably need a USB-to-serial converter and a connector to connect to your breadboard. Whatever you've used in the past for serial communication will work for this project.
- **1 small pink monkey** aka Monski. You may want a second one for a two-player game.

What the computer does with the mouse's data depends on the application. For this application, you'll make a small pink monkey play pong by waving his arms. He'll also have the capability to reset the game by pressing a button, and to serve the ball by pressing a second button.

Connect long wires to the flex sensors, so that you can sew the sensors into the arms of the monkey without having the microcontroller in his lap. A couple of feet should be fine for testing.

Connect long wires to the buttons as well, and mount them in a piece of scrap foam-core or cardboard until you've decided on a final housing for the electronics. Label the buttons "Reset" and "Serve." Wire the sensors to the microcontroller as shown in Figure 2-6.

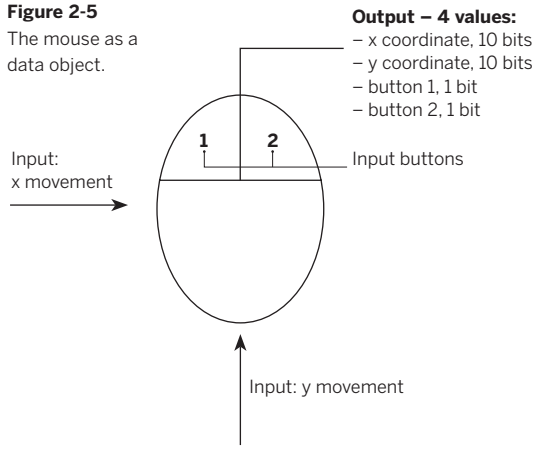


Figure 2-5
The mouse as a data object.

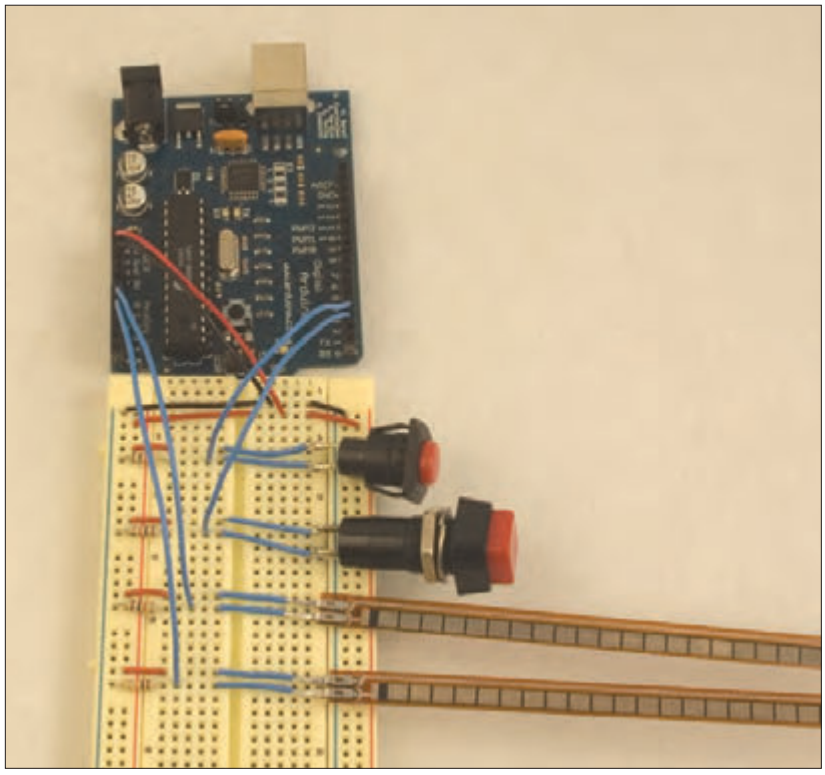
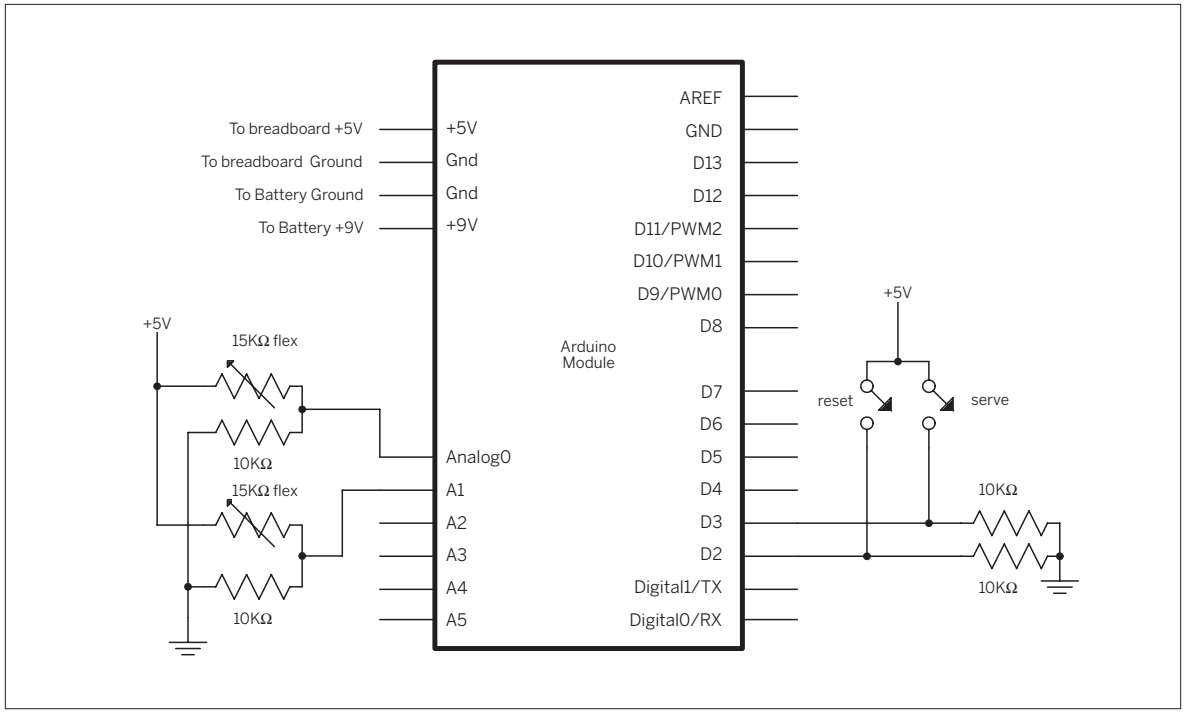


Figure 2-6
The Monski pong circuit. Sensors are shown here with short wires so that the image is clear. You should attach longer wires to your sensors, though.



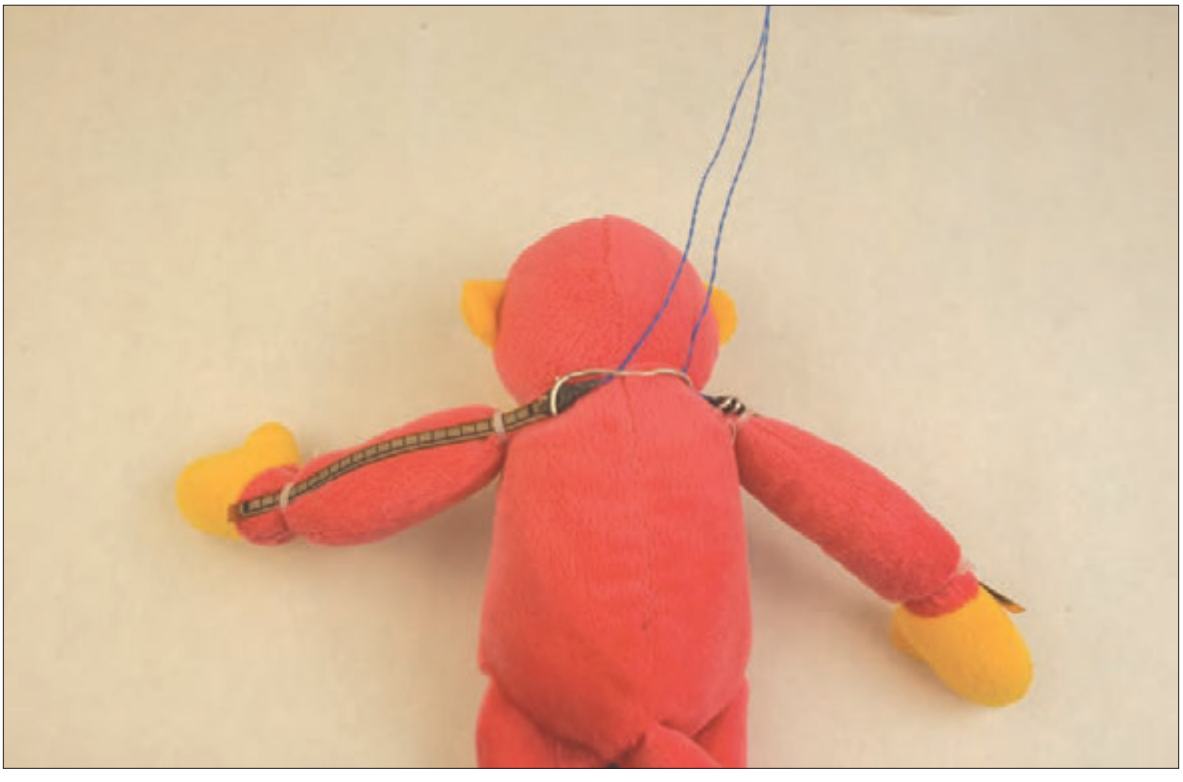


Figure 2-7
A stable support for the sensors is essential if you want good readings from them. Once you know your support works, move it inside the monkey and test it.

Cut a small slit in each of the monkey's armpits to insert the sensors. If you don't want to damage the monkey, you can use tie-wraps or tape to secure the sensors to the outsides of his arms. The sensors need to be positioned so that their movement is consistent, so you should add some sort of support that keeps them in position relative to each other. A piece of flexible modeling wire will do the job nicely. Make sure that both the sensors are facing the same direction, because flex sensors give different readings when flexed one direction than they do when flexed the other direction. Insulate the connections well, because the foam inside the monkey might generate considerable static electricity when he's moving. Hot glue will do the job nicely.

Make sure that the sensors and electrical connections are stable and secure before you start to work on code. Debugging is much harder if the electrical connections aren't consistent.

X

Test It Now use the following code on the Arduino module to confirm that the sensors are working:

If you open the Serial Monitor in Arduino, or your preferred serial terminal application at 9600 bits per second as you did in Chapter 1, you'll see a stream of results like this:

```
284,284,1,1,
285,283,1,1,
286,284,1,1,
289,283,1,1,
```

Before you go to the next section, where you'll be writing some Processing code to interpret the output of this program, be sure to undo this change.

```
/*
   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);
}
```

» Just as you programmed it, each value is separated by a comma, and each set of readings is on a line by itself. Try replacing the part of your code that prints the results with this:

```
// print the results:
Serial.print(leftValue, BYTE);
Serial.print(44, BYTE);
Serial.print(rightValue, BYTE);
Serial.print(44, BYTE);
Serial.print(reset, BYTE);
Serial.print(44, BYTE);
Serial.print(serve, BYTE);
Serial.print(13, BYTE);
Serial.print(10, BYTE);
```

When you view the results in the serial monitor or terminal, you'll get something that looks like garbage, like this:

```
.,P,,,
(,F,,,
(,A,,,
),I,,,
```

"What's going on? The original example displays the values of the sensors as their ASCII code numbers, while this modification sends out the raw binary values. The Serial Monitor and the terminal programs assume that every byte they receive is an ASCII character, so they display the ASCII characters corresponding to the raw binary values in the second example. For example, the values 13 and 10 correspond to the ASCII return and newline characters, respectively, and remove the need for the println in the original example. The value 44 corresponds to the ASCII comma character. Those are the bytes you're sending in between the sensor readings in the second example. The sensor variables (leftValue, rightValue, reset, and serve) are the source of the mystery characters. In the third line of the output, when the second sensor's value is 65, you see the character A, because the ASCII character A has the value 65. For a complete list of the ASCII values corresponding to each character, see www.asciitable.com.

Which way should you format your sensor values? As raw binary, or as ASCII? It depends on the capabilities of the system that's receiving the data, and of those that are passing it through. When you're writing software on a personal computer, it's often easier for your software to interpret raw values. However, many of the network protocols you'll use in this book are ASCII-based. In addition, ASCII is readable by humans, so you may find it

What's ASCII?

ASCII is the American Symbolic Code for Information Interchange. It's a scheme that was created in 1967 by the American Standards Association (now ANSI) as a means for all computers, regardless of their operating systems, to be able to exchange text-based messages. In ASCII, each letter, numeral, or punctuation mark in the Roman alphabet is assigned a number. Anything an end user types is then converted to a string of numbers, transmitted, then reconverted on the other end. In addition to letters, numbers, and punctuation marks, certain page-formatting characters, like the linefeed and carriage return (ASCII 10 and 13, respectively) have ASCII values. That way, not only the text of a message, but also the display format of the message, could be transmitted. These are referred to as control characters. They take up first 32 values in the ASCII set (ASCII 0 – 31). All of the numbers, letters, punctuation, and control characters are covered by 128 possible values. ASCII is too limited to display non-English characters, however, and its few control characters don't offer enough control in the age of graphic user interfaces. Unicode, a more comprehensive code that's a superset of ASCII, has replaced ASCII as the standard for text interchange, and markup languages like PostScript and HTML have replaced ASCII's page formatting, but the original code still lingers on.

easier to send the data as ASCII. For Monski pong, use the number-formatted version (the first example), and you'll see why it's the right choice later in the chapter.

So, undo the changes you made to the Sensor Reader program shown earlier, and make sure that it's working as it did originally. Once you've got the microcontroller sending the sensor values consistently to the terminal, it's time to send them to a program where you can use them to display a pong game. This program needs to run on a host computer that's connected to your Wiring or Arduino board. Processing will do this well.

Try It

Open the Processing application and enter the following code:

```
/*
 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 sketch
// 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) {
```

Continued from previous page.

```
    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();
  }
}
```

Data Packets, Headers, Payloads, and Tails

Now that you've got data going from one object (the microcontroller attached to the monkey) to another (the computer running Processing), take a closer look at the sequence of bytes you're sending to exchange the data. Generally, it's formatted like this:

Left arm sensor (0–1023)	Right arm sensor (0–1023)	Reset button (0 or 1)	Server button (0 or 1)	Return character, linefeed character
1–4 bytes	1–4 bytes	1 byte	1 byte	2 bytes

Each section of the sequence is separated by a single byte whose value is ASCII 44 (a comma). You've just made your first data protocol. The bytes representing your sensor values and the commas that separate them are the payload, and the return and newline characters are the tail. The commas are the delimiters. This data protocol doesn't have a header, but many do.

A header is a sequence of bytes identifying what's to follow. It might also contain a description of the sequence to follow. On a network, where many possible devices could receive the same message, the header might contain the address of the sender or receiver, or both. That way any device can just read the header to decide whether it needs to read the rest of the message. Sometimes a header is as simple as a single byte of a constant value, identifying the beginning of the message. In this example, the tail performs a similar function, separating one message from the next.

On a network, many messages like this are sent out all the time. Each discrete group of bytes is called a packet, and includes a header, a payload, and usually a tail. Any given network has a maximum packet length. In this example, the packet length is determined by the size of the serial buffer on the personal computer. Processing can handle a buffer of a few thousand bytes, so this 16-byte packet is easy for it to handle. If you had a much longer message, you'd have to divide the message up into several packets, and reassemble them once they all arrived. In that case, the header might contain the packet number, so the receiver knows the order in which the packets should be re-assembled.

Make sure that you've shut down the Wiring or Arduino application so that it releases the serial port, then run this Processing application. You should see a list of the sensor values in the message window like this:

```
Sensor 0: 482    Sensor 1: 488    Sensor 2: 1    Sensor 3: 0
Sensor 0: 482    Sensor 1: 488    Sensor 2: 1    Sensor 3: 0
```

Next, it's time to use the data to play pong. First, add a few variables at the beginning of the Processing sketch before the setup() method, and change the setup() to set the window size and initialize some of the variables (the new lines are shown in blue):

```
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

void setup() {
  // set the window size:
  size(640, 480);

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

  // Open whatever port 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 horizontal paddle positions:
  leftPaddleX = 50;
  rightPaddleX = width - 50;

  // set no borders on drawn shapes:
  noStroke();
}
```

Now replace the serialEvent() method with this version, which puts the serial values into the sensor variables:

```
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
```

Continued from previous page.

```
    //and convert the sections into integers:
    int sensors[] = int(split(myString, ','));

    // if you received all the sensor strings, use them:
    if (sensors.length == 4) {
      // assign the sensor strings' values to the appropriate variables:
      leftPaddle = sensors[0];
      rightPaddle = sensors[1];
      resetButton = sensors[2];
      serveButton = sensors[3];

      // print out the variables:
      print("left: "+ leftPaddle + "\tright: " + rightPaddle);
      println("\treset: "+ resetButton + "\tserve: " + serveButton);
    }
  }
}
```

Finally, put some code into the draw() method to draw the paddles:

```
void draw() {
  background(0);
  // draw the left paddle:
  rect(leftPaddleX, leftPaddle, paddleWidth, paddleHeight);

  // draw the right paddle:
  rect(rightPaddleX, rightPaddle, paddleWidth, paddleHeight);
}
```

You may not see the paddles when you first run this code, or until you flex the sensors. You'll need to write a scaling function to scale the range of the sensors to the range of the paddles' vertical motion. For this part, it's important that you have the sensors embedded in the monkey's arms, as you'll be fine-tuning the system, and you want the sensors in the locations where they'll actually get used. Once you're set on the sensors' positions in the monkey, run the Processing program again and watch the left and right sensor numbers as you flex the monkey's arms. Write down of the maximum and minimum values on each arm. To scale the sensors' values to the paddles' movements, use a formula like this:

```
paddlePosition = paddleRange * (sensorValue – sensorMinumum) / sensorRange
```

Add the maximum and minimum values for your sensors as variables before the setup() method. Change these values to match the actual ones you get when you flex the sensors:

```
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
```

Then change the serialEvent() method to include the scaling function for the flex sensor variables. You need to modify the if() statement that puts the sensor readings in the paddle variables. Modify the body of the if() statement that appears after the line int sensors[] = int(split(myString, ','));
as shown:

Now the paddles should move from the top of the screen to the bottom as the you wave the monkey's arms.

NOTE: The variables relating to the paddle range in this example are floating-point numbers (floats), because when you divide integers, you get only integer results. 480/400, for example gives 1, not 1.2, when both are integers. Likewise, 400/480 returns 0, not 0.8333. Using integers when you're dividing two numbers that are in the same order of magnitude produces useless results. Beware of this when writing scaling functions.

```
  // 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);
  }
```

Finally, it's time to add the ball. The ball will move from left to right diagonally. When it hits the top or bottom of the screen, it will bounce off and change vertical direction. When it reaches the left or right, it will reset to the center. If it touches either of the paddles, it will bounce off and change horizontal direction. To make all that happen, you'll need five new variables at the top of the program, just before the setup() method:

```
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
```

In the setup() method, after you set the size of the window (the call to size(640, 480)), you need to give the ball an initial position in the middle of the window:

```
  // initialize the ball in the center of the screen:
  xPos = width /2;
  yPos = height/2;
```

Now, add two methods at the end of the program, one called animateBall() and another called resetBall(). You'll call these from the draw() method shortly:

```
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) {
     resetBall();
  }
  // if the ball goes off the screen right:
  if (xPos > width) {
    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);
}
```

Continued from opposite page.

```
void resetBall() {
  // put the ball back in the center
  xPos = width /2;
  yPos = height/2;
}
```

» You're almost ready to set the ball in motion. But first, it's time to do something with the reset and serve buttons. Add another variable at the beginning of the code (just before the setup() method with all the other variable declarations) to keep track of whether the ball is in motion, and two more to keep score:

```
boolean ballInMotion = false;  // whether the ball should be moving
int leftScore = 0;
int rightScore = 0;
```

» Now you're ready to animate the ball. It should move only if it's been served. The following code goes at the end of the draw() method. The first if() statement starts the ball in motion when the serve button is pressed. The second moves it if it's in service, and the third resets the ball to the center and resets the score when the reset button is pressed:

```
  // 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;
  }
```

» Modify the animateBall() method so that when the ball goes off the screen left or right, the appropriate score is incremented (added lines are shown in blue):

```
  // 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();
  }
```

» Last but not least, add the scoring display. To do this, add two new global variables before the setup() method:

```
PFont myFont;
int fontSize = 36;
```

» Then add two lines before the end of the setup() method to initialize the font:

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

» Finally, add two lines before the end of the draw() method to display the scores:

```
  // print the scores:
  text(leftScore, fontSize, fontSize);
  text(rightScore, width-fontSize, fontSize);
```

Now you can play Monski pong! For added excitement, get a second pink monkey and put one sensor in each monkey so you can play with a friend.

NOTE: You can find a complete listing for this program in Appendix C.

Flow Control

You may notice that the paddles don't move as smoothly onscreen as Monski's arms move. Sometimes the paddles seem not to move at all for a fraction of a second, and sometimes they seem to lag behind the actions you're taking. This is because the communication between the two devices is asynchronous.

Although they agree on the rate at which data is exchanged, it doesn't mean that the receiving computer's program has to use the bits at the same time as they're sent. Monitoring the incoming bits is actually handled by a dedicated hardware circuit, and the incoming bits are stored in a memory buffer called the serial buffer until the current program is ready to use them. Most personal computers have a buffer that can hold a couple thousand bytes. The program using the bits (Processing, in the previous example) is handling a number of other tasks, like redrawing the screen, handling the math that goes with it, and sharing processor time with other programs through the operating system. It may get bytes from the buffer less than a hundred times a second, even though the bytes are coming in much faster.

There's another way to handle the communication between the two devices that can alleviate this problem. If Processing asks for data only when it needs it, and if the microcontroller only sends one packet of data when it gets a request for data, the two will be in tighter sync.

To make this happen, first wrap the whole of the loop() method in the **Arduino** program (the Sensor Reader program shown back in the beginning of the "Project #1: Monski pong" section) in an if() statement like this:

Next, add some code to the Monski pong **Processing** program.

```
void loop() {
    // check to see whether there is a byte available
    // to read in the serial buffer:
    if (Serial.available() > 0)   {
        // read the serial buffer;
        // you don't care about the value of
        // the incoming byte, just that one was
        // sent:
        int inByte = Serial.read();
        // the rest of the existing main loop goes here
        // ...
    }
}
```

First, add a new global variable before the setup() method. This variable will keep track of whether you've received any data from the microcontroller:

```
boolean madeContact = false;        // whether you've made initial contact
                                    // with the microcontroller
```

At the beginning of the draw() method, add this:

```
  // If you haven't gotten any data from the microcontroller yet, send out
  // the serial port to ask for data. What value you send doesnt matter,
  // since the microcontroller code above isn't doing anything with the
  // byte you send. So send a carriage return for debugging purposes:
  if (madeContact == false) {
    myPort.write('\r');
  }
```

Finally, change the serialEvent() method. New lines are shown in blue:

```
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;
```

Continued from previous page.

```
      // 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 sensnor values:
      print("left: "+ leftPaddle + "\tright: " + rightPaddle);
      println("\treset: "+ resetButton + "\tserve: " + serveButton);

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

Now the paddles should move much more smoothly. What's happening now is this: the microcontroller is programmed to check to see whether it's received any bytes serially. If it has, it reads the byte just to clear the buffer, then sends out its data. Whenever it gets no bytes, it sends no bytes. Processing, meanwhile, starts its program by sending a byte out. This triggers the microcontroller to send an initial set of data. Processing reads this data in the serial event() method; then, when it's got all the data, it sends another byte to request more data. The microcontroller, seeing a new byte coming in, sends out another packet of data, and the whole cycle repeats itself. Neither program has more data from the other than it can deal with at any given moment. The slight delay it introduces is not noticeable in the display. In fact, it's less of a delay than the previous program had. Notice that the value of the byte sent is irrelevant. It's used only as a signal from the Processing code to let the microcontroller know when it's ready for new data. This method of handling data flow control is sometimes referred to as a handshake method, or call-and-response. Whenever you're sending packets of data, call-and-response flow control can be a useful way to ensure consistent exchange.

X

Project 2

Wireless Monski pong

Monski pong is fun, but it'd be more fun if Monski didn't have to be tethered to the computer through a USB cable. This project breaks the wired connection between the microcontroller and the personal computer, and introduces a few new networking concepts: the modem and the address.

MATERIALS

- **1 completed Monski pong project** from earlier
- **1 9V battery and battery snap connector** SparkFun (www.sparkfun.com) part number PRT-00091, or Digi-Key (www.digikey.com) part number 2238K-ND
- **1 BlueSMiRF Bluetooth modem module** from SparkFun (part WRL-00582)
- **1 project box** to house the microcontroller, battery, and radio board

NOTE: If your personal computer doesn't have a built-in Bluetooth radio, you'll also need a Bluetooth adaptor for it. The Bluetooth v.1.2 USB module that SparkFun carries will work fine, and most computer stores carry USB Bluetooth adaptors.

Bluetooth: A Multilayer Network Protocol

The new piece of hardware in this project is the Bluetooth module. This module has two interfaces: two of its pins, marked RX and TX, are an asynchronous serial port that can communicate with a microcontroller. It also has a radio that communicates using the Bluetooth communications protocol. It acts as a modem, translating between the Bluetooth and regular asynchronous serial protocols.

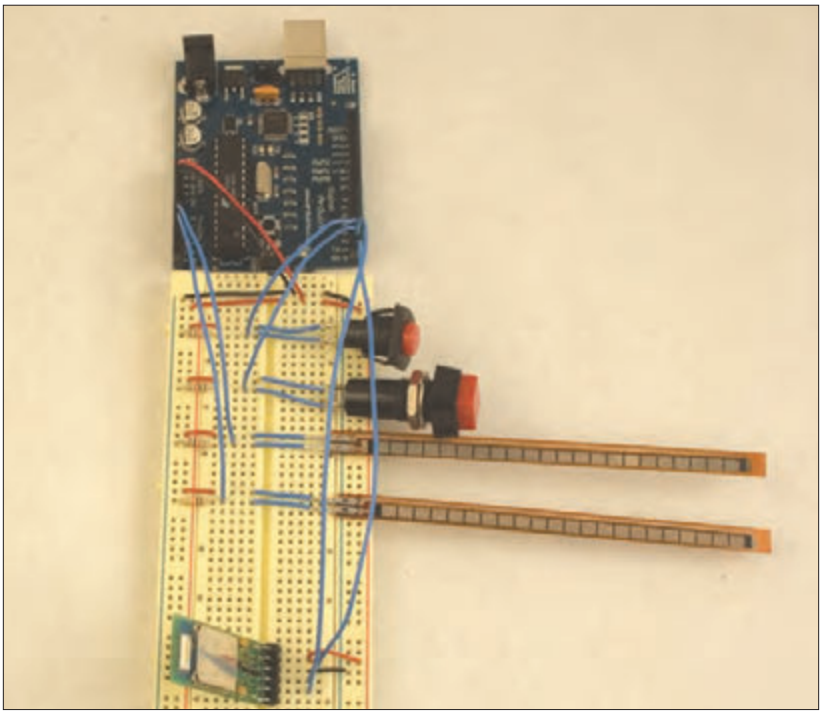
NOTE: The first digital modems were devices that took data signals and converted them to audio signals to send them across a voice telephone connection. They modulated the data on the audio connection, and demodulated the audio signal back into a data signal. These simple serial-to-audio modems are becoming increasingly rare, but their descendants are everywhere, from the set-top boxes that modulate and demodulate your Internet connection from the cable television data signal to the sonar modems that convert data into ultrasonic pings to be sent from ships to submarine exploration robots used in marine research.

Bluetooth is a multilayered communications protocol, designed to replace wired connections for a number of applications. As such, it's divided into a group of possible applications protocols called profiles. The simplest Bluetooth devices are serial devices like the module used in this project. These implement the Bluetooth Serial Port Profile (SPP). Other Bluetooth devices implement other protocols. Wireless headsets implement the audio Headset Profile. Wireless mice and keyboards implement the Human Interface Device (HID) Profile. Because there are a number of possible profiles a Bluetooth device might support, there is also a Service Discovery Protocol, via which radios exchange information about what they can do. Because the protocol is standardized, you get to skip over most of the details of making and maintaining the connection so you can concentrate on exchanging data. It's a bit like how RS-232 and USB made it possible for you to ignore most of the electrical details necessary to connect your microcontroller to your personal computer, so you could focus on sending bytes in the last project.

Add the Bluetooth module to the Monski pong breadboard as shown in Figure 2-8. Connect the module's ground to the ground on the Arduino module, and its input pin to the 5-volt output from the module. Move the power jumper on the Arduino module so that it's closest to the DC power jack. Then connect the battery's black wire to the module's ground, and its red wire to the +9V power pin. The module will start up, and the Bluetooth radio's green LED will blink.

Pairing Your Computer with the Bluetooth Module

To make a wireless connection from your computer to the module, you have to pair the two of them. To do this, open your computer's Bluetooth control panel to browse for new devices. In Mac OS X, choose the Apple menu→System Preferences, then click Bluetooth. In the Settings tab, make sure Bluetooth is turned on, and check Discoverable and Show Bluetooth Status in Menu Bar. In the Devices tab, click Set Up New Device to launch the Bluetooth Setup Assistant. When you have to choose a device type,



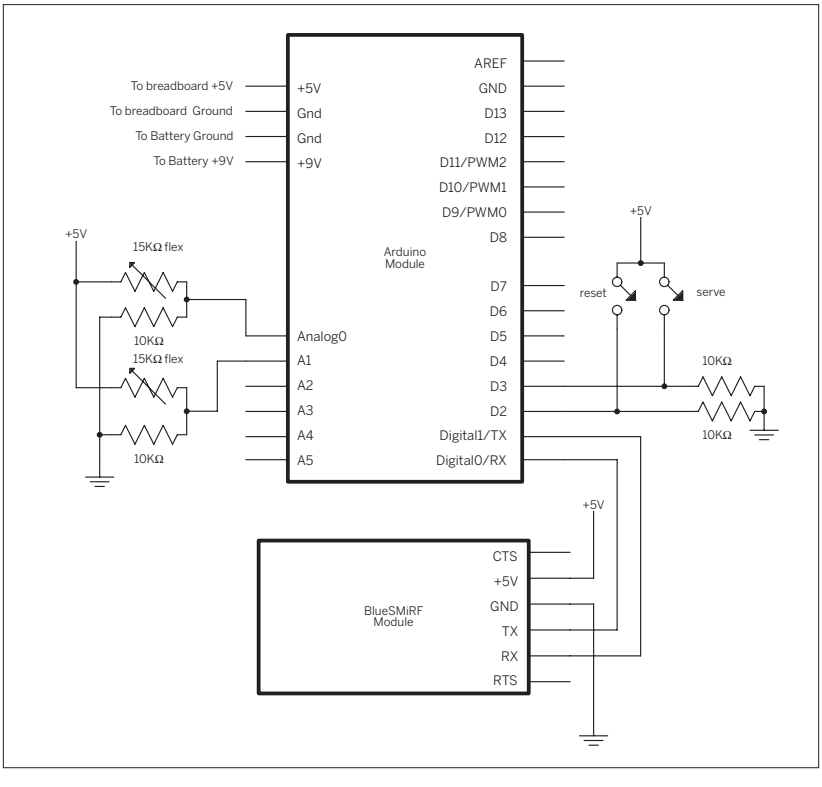
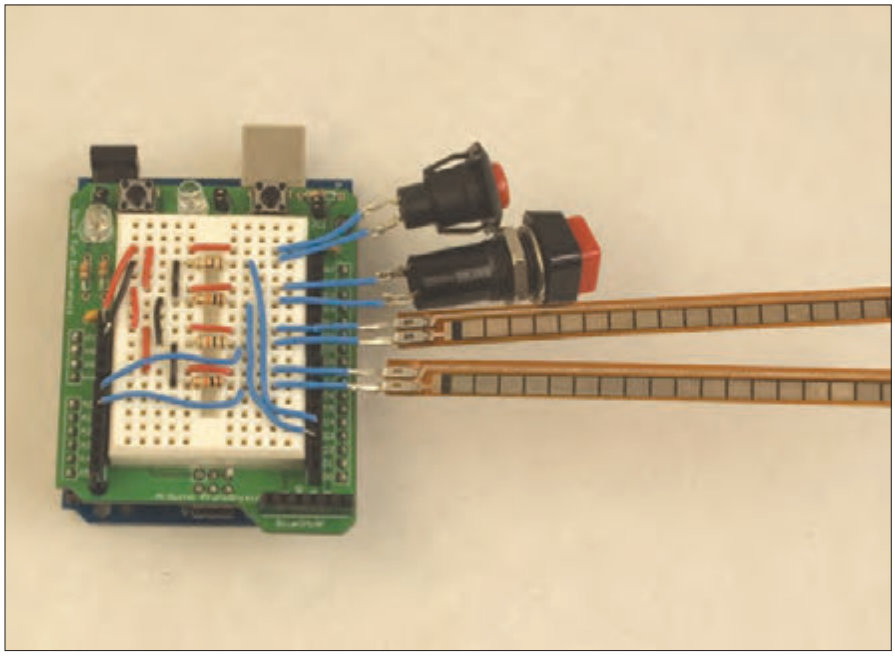
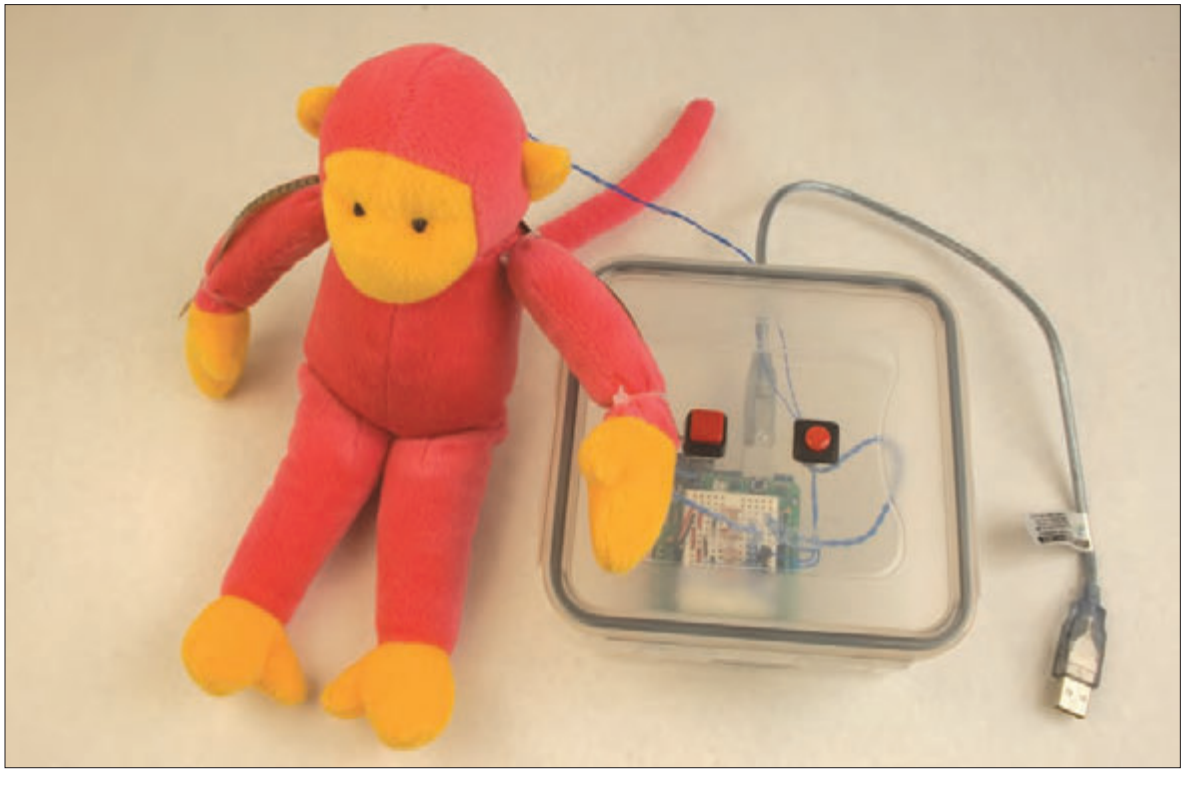


Figure 2-8
Monski pong board, with Bluetooth module added. Be sure to switch the power jumper to the DC power jack side. Once you've built the circuit for this, drill holes in the project box for the buttons and the wires leading to the flex sensors. Mount the breadboard, Arduino module, and battery in the project box.

Finishing Touches: Tidy it up, box it up



Make it small
You might want to shrink Monski pong down so it's more compact. This figure shows the Monski pong circuit on a breadboard shield. This is the same circuit as the one in Figure 2-6, just on a different breadboard so it can fit in a project box.



All boxed up
Kitchen storage containers make excellent project boxes. Here's the Monski pong controller with Monski attached.

choose Any Device. The computer will search for devices, and will find one called BlueRadios-COM0-1. If you have no other Bluetooth devices nearby, it will be the only one. Choose this device, and when asked for a passkey, enter default. A connection will be established, and you'll be told that there are no selected services on this device. That's okay. Continue until you can quit the Assistant.

Mac OS X users: If you're using a version of Mac OS X before version 10.4, you can only enter numeric passkeys. You'll need to change the module's passkey first. To do this, connect the TX and RX pins to a serial port on the computer. See the next project in this chapter for the details on connecting the Bluetooth module to the serial port. Once you're connected, send the following string to the module to change the password:

```
ATSP,0000,default,\r
```

Replace the 0000 with your own numeric password, and press Enter in place of \r. (In this book, \r denotes a carriage return, or ASCII value 13, and \n denotes a newline character, or ASCII value 10). Then reset the module by unplugging it and plugging it back in. Now you can follow the previous instructions to make a pairing between your computer and the Bluetooth module. Once you've done that, reconnect the Bluetooth module to the Monski pong project as shown above.

In Windows, the process differs, depending on the Bluetooth radio you have installed. There are several different Bluetooth radios available for Windows machines, and each one has a slightly different user interface. XP Service Pack 2 introduced a unified configuration interface for Bluetooth, but some older radios still use the vendor-specific configuration tools. The user manual for your specific radio will cover the details you need, but the process will be something like this:

Right-click the Bluetooth Icon in the lower righthand corner of the taskbar (this area is called the system tray or notification area) to access the Bluetooth settings. Check the device or security properties to make sure that your computer's Bluetooth device is discoverable, connectable, and pairable. Check the service properties to make sure that Bluetooth COM port service is enabled. Then add a new Bluetooth device. When you get the option to search for new devices, do so, and you should see the BlueSMiRF (with a name such as BlueRadios COM) device show up in the list of available devices. If you have no other Bluetooth devices nearby, it will be the only one. When prompted for a password, enter default. This step will add a new serial port to your list of serial ports. Make note of the port name (mine is COM40), so you can use it later.

Adjusting the Monski pong Program

Once your computer has made contact with the Bluetooth module, you can connect to it like a serial port. Run the Monski pong Processing sketch and check the list of serial ports. You should see the new port listed along with the others. Take note of which number it is, and change this line in the setup() method:

```
myPort = new Serial(this, Serial.list()[0], 9600);
```

For example, if the Bluetooth port is the ninth port in your list, change the line to open Serial.list[8]. With no other changes in code, you should now be able to connect wirelessly. Monski is free to roam around the room as you play pong When the Processing program makes a connection to the Bluetooth module, the green LED on the module will turn off and the red one will come on.

If you plug or unplug any serial devices after you do this, including the Arduino, you'll need to quit and restart the Processing program, as the count of serial ports will have changed.

If you haven't modified your Arduino and Processing code to match the call-and-response version of the Monski pong program shown in the "Flow Control" section earlier, you might have a problem making a connection to the radio. If so, the green LED will stay on and not flash. What's happening is that the microcontroller module is sending serial data constantly, and the Bluetooth module's serial buffer is filling up. When a wireless connection is made, the Bluetooth module sends a string out on the TX pin with the address of the device that made the connection, like so:

```
CONNECT,000D93039D96
```

Because this line always ends with a carriage return, you can listen for that in your microcontroller code by using the call-and-response method described earlier, so make those modifications before going any further. Now the program will do nothing until it sees an initial carriage return; then it will send data continually.

Project 3

Negotiating in Bluetooth

The steps you went through with the Bluetooth Assistant or Bluetooth Wizard negotiated a series of exchanges between your computer and the BlueSMiRF module that included discovering other radios, learning the services offered by those radios, and pairing to open a connection. It's very convenient to be able to do this from the graphical user interface, but it'd be even better if the devices could negotiate this exchange themselves. In the section that follows, you'll negotiate some parts of that exchange directly, in order to understand how to program devices that can do that negotiation.

MATERIALS

- **1 BlueSMiRF basic module** from SparkFun (part WRL-00582)
- **1 USB-to-serial converter** The SparkFun BOB-00718 is shown next, but you can also use the MAX3323 and USB-to-RS-232 converter version as shown earlier.
- **1 solderless breadboard**

Wire the Bluetooth module to the USB-to-serial converter as shown in Figure 2-9. The USB-to-serial converter converts the TTL serial signals from the BlueSMiRF module to USB. Connect the converter to a USB port on your computer and open your serial terminal program. open a connection to the USB-to-serial converter's serial port at 9600 bits per second.

Modems are designed to open a connection to another modem, negotiate the terms of data exchange, carry on an exchange, then disconnect. To do this, they have to have two operating modes, usually referred to as command mode, in which you talk *to* the modem, and data mode, in which you talk *through* the modem. Bluetooth modems are no different in this respect. Most Bluetooth modems use a set of commands based on the original commands designed for telephone modems, known as the Hayes AT command protocol. All commands in the Hayes command protocol (and therefore in Bluetooth command protocols as well) are sent using ASCII characters. There's a common structure to all the AT commands. Each command sent from the controlling device (like a microcontroller or personal computer) to the modem begins with the ASCII string AT followed by a short string of letters and numbers representing the command, followed by any parameters of the command, separated by commas. The command ends with an ASCII carriage return. For example, here's the string to ask the BlueRadios module inside of the BlueSMiRF module for its firmware version:

```
ATVER,ver1\r
```

The \r is a carriage return. Hit the Return key whenever you see it here. Type it into the serial terminal program now. The BlueRadios module should respond with something like this:

```
Ver 3.4.1.2.0
```

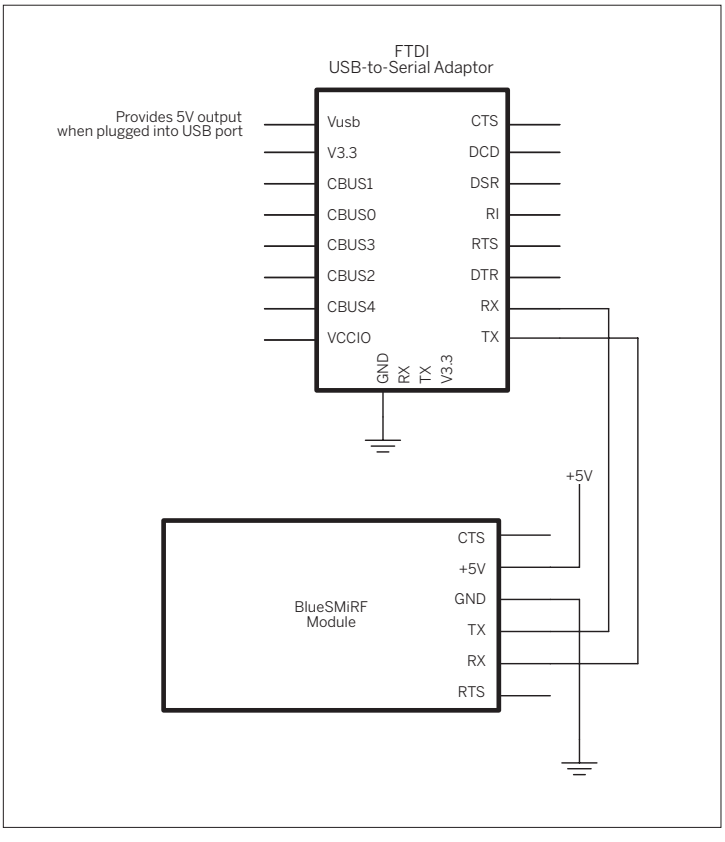
Any time you want to just check that the module is working, type `AT\r`. It will respond with OK. There's a list of all the commands available for this module available from www.sparkfun.com or from www.blueradios.com. A few of them are covered here. Each Bluetooth modem manufacturer has its own set of AT commands, and unfortunately they're all different. But they are all based on the AT command protocol, so they'll all have the same basic format as the one you see here.

Currently, the module is in command mode. One of the first things you'd like it to do is to give you a list of other Bluetooth-enabled devices in the area. If you've got more devices than just your laptop around, take a rough guess of how many, and type this sequence of commands:

```
ATUCL\r
```

This clears any current commands, and puts the module in idle mode. The module will return OK. Then type:

```
ATDI,3,00000000\r
```



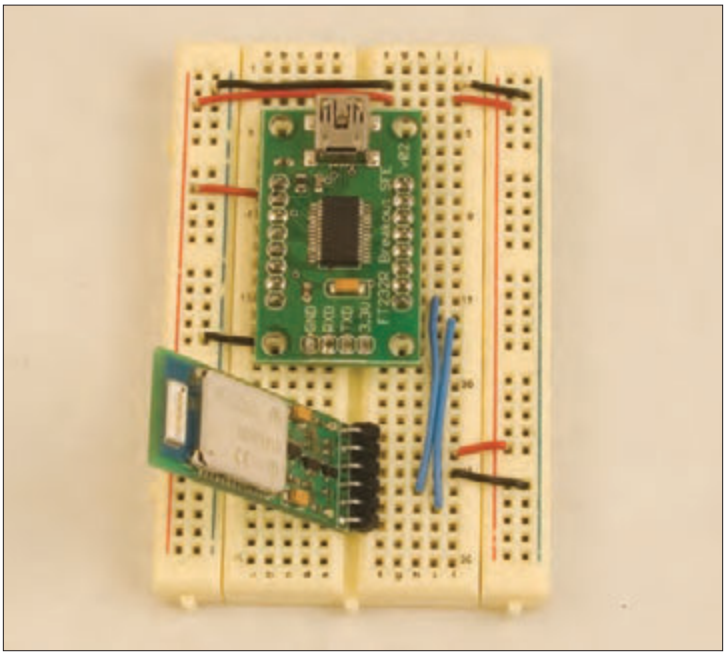


Figure 2-9
BlueSMiRF module connected to a FT232RL USB-to-serial converter.

ATDI tells it to look for other radios. 3 tells it to look until it finds three others. 00000000 tells it to look for any type of Bluetooth radio (phone, headset, serial device, etc). After several seconds, it will come back with a list like this:

```
00038968505F,00200404
000D93039D96,0010210C
00119FC2AD3C,0050020C
DONE
```

This is a list of all the other Bluetooth devices it found. The first part of every string is the device's unique address. That's the part you need in order to make a connection to it. Manufacturers of Bluetooth devices agree on a standard addressing scheme so no two devices get the same address. The second part is the device class (what type of device it is), and the third, when there is one, the device's name. Names don't have to be unique, but addresses do, which is why you always use the address to connect.

Now that you've got a list of connections, try to connect to the one that represents your computer, like so:

```
ATDM,address,1101\r
```

ATDM tells the module to attempt to connect. The address tells it what device to connect to, and 1101 tells it what profile to use; in this case, the serial port profile. The BlueSMiRF should respond:

```
NO ANSWER
```

Next you need to open the serial port on your computer that's connected to its Bluetooth radio. In **Mac OS X**, it's the Bluetooth PDA-Sync port. Open a second window in your terminal program, and connect to that serial port using GNU Screen, like so:

```
screen /dev/tty.Bluetooth-PDA-Sync 9600
```

For **Windows** users, the COM port varies depending on your Bluetooth device, but if you check the Device Manager's list of serial ports, you'll see several associated with the Bluetooth radio. Use the lowest numbered one that will open in PuTTY (most of them will refuse to open). That should be the port that your radio can connect to.

If you're using the standard Windows Bluetooth interface, right-click the Bluetooth icon in the system tray and choose Device Properties and Security. Make sure your radio is discoverable, connectable, and pairable.

Go back to the window with the serial connection to the BlueSMiRF, and send the following command:

```
ATPAIR, address\r
```

where address is the Bluetooth address of your computer that you discovered earlier using ATDI.

The computer will ask you for a passkey. Enter default. The BlueSMiRF should reply:

```
PAIRED,address
```

Once you're paired, you need to connect, using the following AT command:

```
ATDM,address,1101\r
```

When you get a good connection to the BlueSMiRF, the LED on it will turn red, and you'll get a message like this:

```
CONNECT,000D93039D96
```

You're now out of command mode and into data mode. You should be able to type directly from one window to the other.

NOTE: You can't initiate a connection from the BlueSMiRF to the computer unless you've already paired with the BlueSMiRF from the computer previously. This is so because BlueSMiRF radios can't initiate a serial connection unless they've already made a pairing with the other device. You'll see more on these radios in Chapter 6.

To get out of data mode (to check the modem's status, for example), type:

```
+++\r
```

This will give you an OK prompt again. You can type any of the AT commands you want now, and get replies. To return to data mode, type:

```
ATMD\r
```

Finally, when you're in command mode, you can type ATDH\r to disconnect, and you'll get this reply:

```
NO CARRIER
```

That means you're disconnected. If you want to connect to another device, start by putting the module back in idle

mode with `ATUCL\r` and following the same steps as previously.

Because the AT commands are just text strings, you can easily use them in microcontroller programs to control the module, make and break connections, and exchange data.

Because all the commands are in ASCII, it's a good idea to exchange data in ASCII mode, too. So the data string that you set up earlier to send Monski's sensor readings in ASCII would work well over this modem.
X

Conclusion

The projects in this chapter have covered a number of ideas that are central to all networked data communication. First, keep in mind that data communication is based on a layered series of agreements, starting with the physical layer, then the electrical, the logical, the data, and finally the application layer. Keep these layers in mind as you design and troubleshoot your projects and you'll find it easier to isolate problems

Second, remember that serial data can be sent either as ASCII or as raw binary values, and which you choose to use depends both on the capabilities and limitations of the devices you're connecting, and on all the devices in the middle that connect them. It might not be wise to send raw binary data, for example, if the modems or the software environments you program in are optimized for ASCII data transfer.

Third, when you think about your project, think about the messages that need to be exchanged, and come up with a data protocol that adequately describes all the information you need to send. This is your data packet. You might want to add header bytes, separators, or tail bytes to make reading the sequence easier.

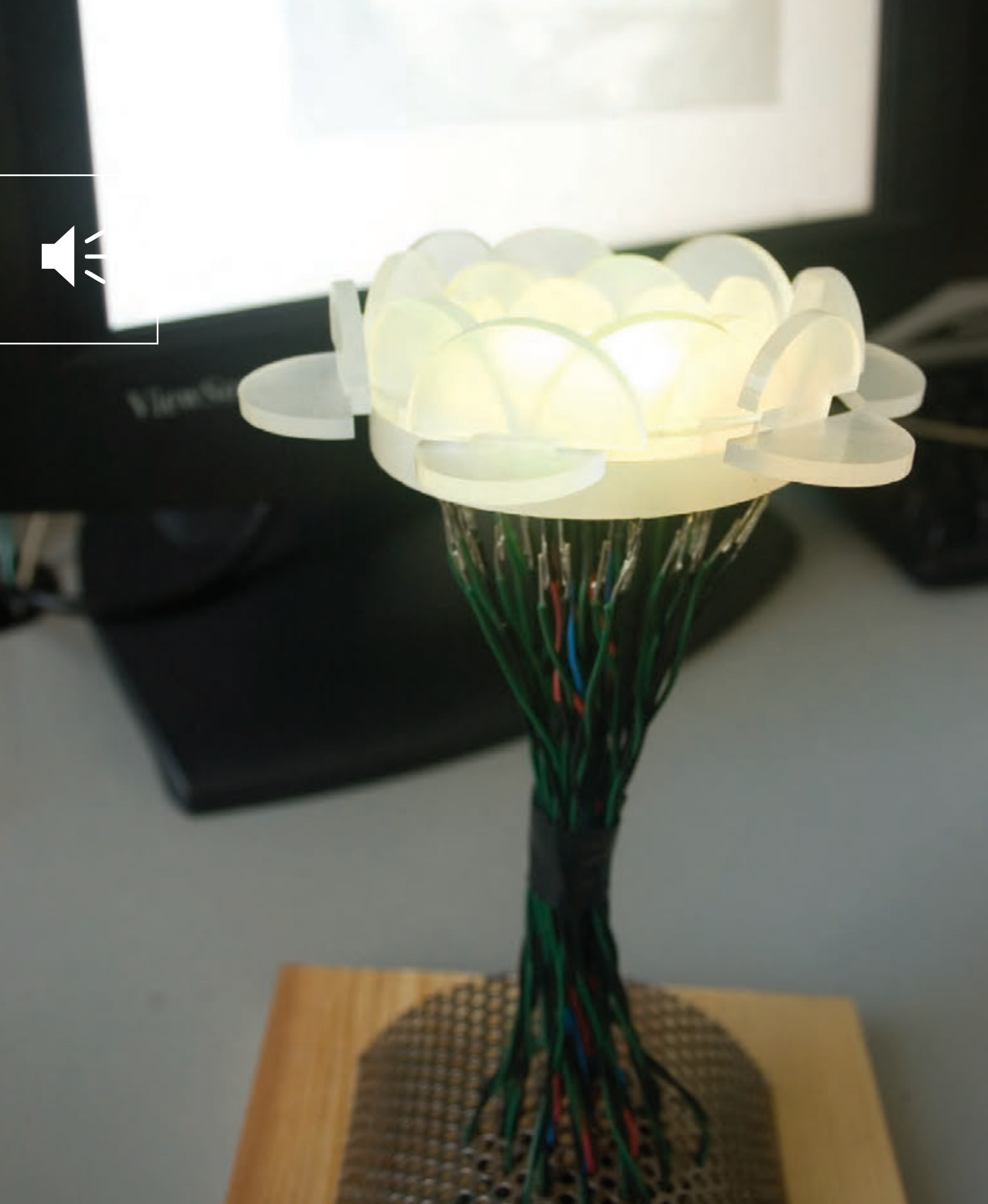
Fourth, think about the flow of data, and look for any ways you can ensure a smooth flow with as little overflowing of buffers or waiting for data as possible. A simple call-and-response approach can make data flow much smoother.

Finally, get to know the modems and other devices that link the objects at the end of your connection. Make sure you understand their addressing schemes and any command protocols they use so that you can factor their strengths and limitations into your planning, and eliminate those parts that make your life more difficult. Whether you're connecting two objects or two hundred, these same principles will apply.
X

The JitterBox by Gabriel Barcia-Colombo

The JitterBox is an interactive video Jukebox created from a vintage 1940's radio restored to working condition. It features a tiny video-projected dancer who shakes and shimmies in time with the music. The viewer can tune the radio and the dancer will move in time with the tunes. The JitterBox uses serial communication from an embedded potentiometer tuner which is connected to an Arduino microcontroller in order to select from a range of vintage 1940's songs. These songs are linked to video clips and played back out of a digital projector. The dancer trapped in the JitterBox is Ryan Myers.


54
70
90
120
150
60
80
100
130
160



3

MAKE: PROJECTS

A More Complex Network

Now that you've got the basics of network communications, it's time to tackle something more complex. The best place to start is with the most familiar data network: the Internet. It's not actually a single network, but a collection of networks all owned by different network service providers and linked using some common protocols. This chapter describes the structure of the Internet, the devices that hold it together, and the shared protocols that make it possible. You'll get hands-on experience with what's going on behind the scenes when your web browser or email client is doing its job, and you'll use the same messages those tools use to connect your own objects to the Net.

Networked Flowers by Doria Fan, Mauricio Melo, and Jason Kaufman.
Networked Flowers is a personal communication device for sending someone digital blooms. Each bloom has a different lighting animation. The flower sculpture has a network connection. The flower is controlled from a website that sends commands to the flower when the web visitor chooses a lighting animation.

Network Maps and Addresses

In the last chapter, it was easy to keep track of where messages went, because there were only two points in the network you built: the sender and the receiver. In any network with more than two objects, from three to three billion, you need a map to keep track of which objects are connected to which, and an addressing scheme to know how a message gets to its destination.

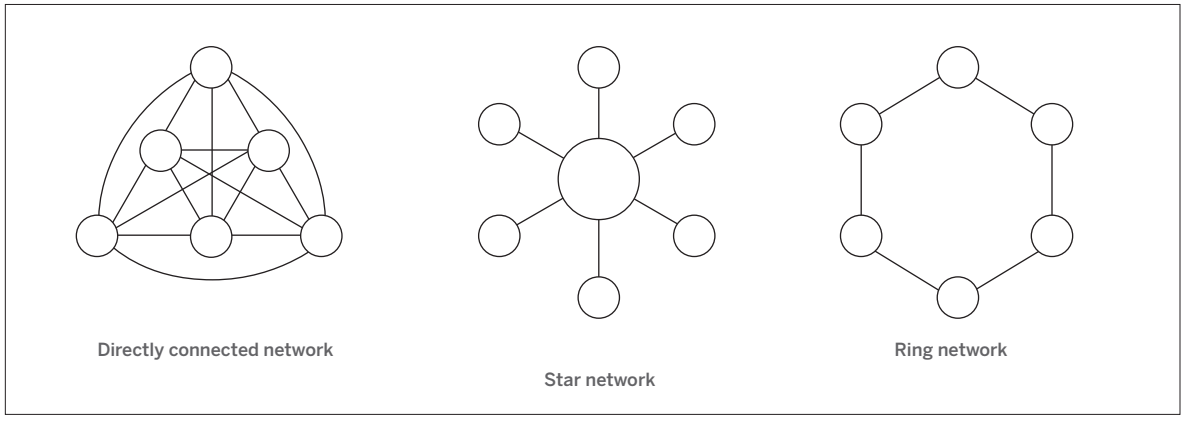
Network Maps: How Things Are Connected

The arrangement of physical connections on a network depends on how you want to route the messages on that network. The simplest way is to make a physical connection from each object in the network to every other object. That way, messages can get sent directly from one point to another. The problem with this approach, as you can see from the directly connected network in Figure 3-1, is that the number of connections gets large very fast, and the connections get tangled. A simpler alternative to this is to put a central controller in the middle, and pass all messages through this hub, as seen in the star network. This way works great as long as the hub continues to function, but the more objects you add, the faster the hub has to be to process all the messages. A third alternative is to daisy-chain the objects, connecting them together in a ring. This design makes for a small number of connections, and it means that any message has two possible paths, but it can take a long time for messages to get halfway around the ring to the most distant object.

In practice (such as on the Internet), a multitiered star model like the one shown in Figure 3-2 works best. Each connector (symbolized by a light-colored circle) has a few objects connected to it, and each connector is linked to a more central connector. At the more central tier (the dark-colored circles in Figure 3-2), each connector may be linked to more than one other connector, so that messages can pass from one endpoint to another via several different paths. This system takes advantage of the redundancy of multiple links between central connectors, but avoids the tangle caused by connecting every object to every other object.

If one of the central connectors isn't working, messages are routed around it. The connectors at the edges are the weakest points. If they aren't working, the objects that depend on them have no connection to the network.

Figure 3-1
Three types of network: direct connections between all elements, a star network, and a ring network.



As long as the number of objects connected to each of these is small, though, the effect on the whole network is minimal. It may not seem minimal when you're using the object whose connector fails, but at least the rest of the network remains stable, so it's easy to reconnect when your connector is working again.

If you're using the Internet as your network, you can take this model for granted. If you're building your own network, however, it's worth comparing all of these models to see which is best for you. In some simpler systems, one of the three networks shown in Figure 3-1 might do the job just fine, and save you some complications. As you get further in the book, you'll see some examples of these, but for the rest of this chapter, you'll work with the multitiered model by relying on the Internet as your infrastructure.
X

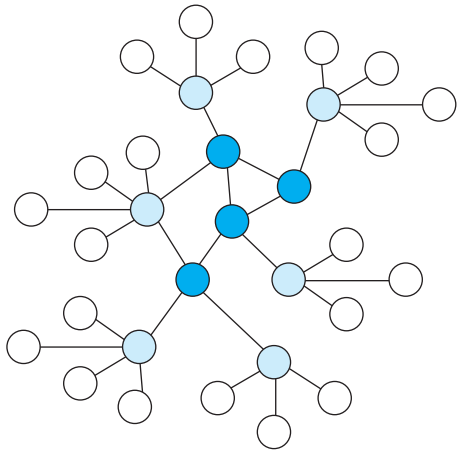


Figure 3-2
A complex, multitiered network.

Modems, Hubs, Switches, and Routers

The connectors in Figure 3-2 represent several different types of devices on the Internet. The most common among these are modems, hubs, switches, and routers. Depending on how your network is set up, you may be familiar with one or more of these. There's no need to go into detail as to the differences, but some basic definitions are in order:

A modem is a device that converts one type of signal into another, and connects one object to one other object. Your home cable or DSL modem is an example. It takes the digital data from your home computer or network, converts it to a signal that can be carried across the phone line or cable line, and connects to another modem on the other end of the line. That modem is connected to your Internet service provider's network. By this definition, the Bluetooth radios from Chapter 2 could be considered modems, as they convert electrical signals into radio signals and back.

A hub is a device that multiplexes data signals from several devices and passes them upstream to the rest of the net. It doesn't care about the recipients of the messages it's carrying — it just passes them through in both directions. All the devices attached to a hub receive all the messages that pass through the hub, and each one is responsible for filtering out any messages that aren't addressed to it. Hubs are cheap and handy, but they don't really manage traffic at all.

A switch is like a hub, but more sophisticated. It keeps track of the addresses of the objects attached to it, and passes on only messages addressed to those objects. Objects attached to a hub don't get to see messages that aren't addressed to them.

Modems, hubs, and switches generally don't actually have their own addresses on the network (though some cable and DSL modems do). A router, on the other hand, is visible to other objects on the network. It has an address of its own, and can mask the objects attached to it from the rest of the net. It can give them private addresses, meaningful only to the other objects attached to the router, and pass on their messages as if they come from the router itself. It can also assign IP addresses to objects that don't have one when they're first connected to the router.

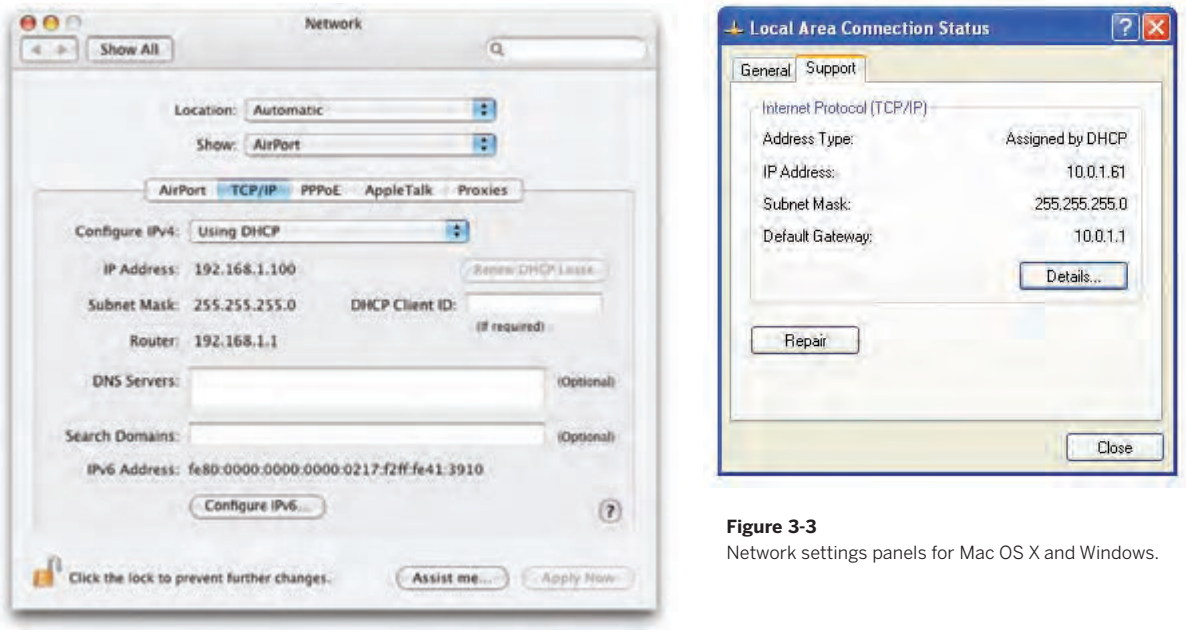


Figure 3-3
Network settings panels for Mac OS X and Windows.

Hardware Addresses and Network Addresses

Whether you're using a simple network model where all the objects are directly connected, a multitiered model, or anything in between, you need an addressing scheme to get messages from one point to another on the network. When you're making your own network from scratch, you have to make up your own addressing scheme. For the projects you're making in this book, however, you're relying on existing network technologies, so you get to use the addressing schemes that come with them. For example, when you used the Bluetooth radios in Chapter 2, you used the Bluetooth protocol addressing scheme. When you connect Internet devices, you use the Internet Protocol (IP) addressing scheme. Because most of the devices you connect to the Internet also rely on a protocol called Ethernet, you also use the Ethernet address protocol. A device's IP address can change when it's moved from one network to another, but its hardware address, or Media Access Control (MAC) address, is burned into the device's memory and doesn't change. It's a unique ID number assigned by the manufacturer that differentiates that device from all the other Ethernet devices on the planet. Wi-fi adapters also have hardware addresses.

You're probably already familiar with your computer's IP address and maybe even its hardware address. In Mac OS X, Click Apple Menu→Location→Network Preferences to open the Network control panel, and you'll get a list of the possible ways your computer can connect to the net. Click on the popup menu labeled Show and you get a list of the network interfaces. It's likely that you have at least a built-in Ethernet interface and an Airport interface. The built-in Ethernet and Airport interfaces both have hardware addresses, and if you choose either interface, you can find out that interface's hardware address. For Airport, it's listed as the Airport ID under the Airport tab, and for the built-in Ethernet, it's called the Ethernet ID under the Ethernet tab. In either interface, click on the TCP/IP tab and you can see the machine's IP address if you're connected to a network.

In Windows, click the Start Menu→Control Panel, then double-click Network Connections. Each network interface has its own icon in this control panel. Click Local Area Connection for your built-in Ethernet connection, or Wireless Network Connection for yourWi-FI connection. Under the Support tab, click Details to see the IP settings and hardware address.

Figure 3-3 shows the network connection settings for Mac OS X and Windows. No matter what platform you're on, the hardware address and the Internet address will take these forms:

The hardware address is made up of six numbers, written in hexadecimal notation, like this: 00:11:24:9b:f3:70

The IP address is made up of four numbers, written in decimal notation, like this: 192.168.1.20

You'll need to know the IP address to send and receive messages, and you'll need to know the hardware address in order to get an IP address on some networks, so make note of both of them for every device you're using whenever you start to work on a new project.

Street, City, State, Country: How IP Addresses Are Structured

Geographic addresses can be broken down into layers of detail, starting with the most specific (the street address) and moving to the most general (the country). Internet addresses are also multilayered. The most specific part is the final number, which tells you the address of the computer itself. The numbers that precede this tell you the subnet that the computer is on. Your router shares the same subnet as your computer, and its number is usually identical except for the last number. The numbers of an IP address are called octets, and each octet is like a section of a geographic address. For example, imagine a machine with this number: 192.168.0.20

The router that this machine is attached to most likely has this address: 192.168.0.1

Each octet can range from 0–255, and some numbers are reserved by convention for special purposes. For example, the router is often the address xxx.xxx.xxx.1. The subnet can be expressed as an address range, 217.123.152.xxx. Sometimes a router manages a larger subnet, or even a group of subnets, each with their own local router. The router that this router is connected to might have the address 192.168.0.1

Each router controls access for a defined number of machines below it. The number of machines it controls is encoded in its subnet mask. You've probably encountered a subnet mask in configuring your personal computer. A typical subnet mask looks like this: 255.255.255.0

You can read the number of machines in the subnet by reading the value of the last octets of the subnet mask. It's easiest if you think of the subnet in terms of bits. Four bytes is 32 bits. Each bit you subtract from the subnet increases the number of machines it can support. Basically, you "subtract" the subnet mask from its maximum value of 255.255.255.255 to get the number of machines. For example, if the subnet were 255.255.255.255, then there could be only one machine in the subnet, the router itself. If the last octet is 0, as it is above, then there can be up to 255 machines in the subnet in addition to the router. A subnet of 255.255.255.192 would support 63 machines and the router (255 – 192 = 64), and so forth. Table 3-1 shows a few other representative values to give you an idea.

Table 3-1. The relationship between subnet mask and maximum number of machines on a network.

Subnet mask	Maximum number of machines on the subnet, including the router
255.255.255.255	1 (just the router)
255.255.255.192	64
255.255.255.0	256
255.255.252.0	1024
255.255.0.0	65,536

Knowing the way IP addresses are constructed helps you to manage the flow of messages you send and receive. Normally, all of this is handled for you by the software you use: browsers, email clients, and so forth. But when you're building your own networked objects, it's necessary to know at least this much about the IP addressing scheme so you can find your router and what's beyond it.

Numbers into Names

By now you're probably thinking that this is ridiculous, because you only know internet addresses by their names, like www.makezine.com, or www.archive.net. You never deal with numerical addresses, nor do you want to. There's a separate protocol, the Domain Name System (DNS), for assigning names to the numbers. Machines on the network called nameservers keep track of which names are assigned to which numbers. In your computer's network configuration, you'll notice a slot where you can enter the DNS address. Most computers are configured to obtain this address from a router using the DHCP protocol (which also provides their IP address), so you don't have

Private and Public IP Addresses

Not every object on the Internet can be addressed by every other object. Sometimes, in order to be able to support more objects, a router hides the addresses of the objects attached to it, and sends all their outgoing messages to the rest of the net as if they came from the router itself. There are special ranges of addresses set aside in the IP addressing scheme for use as private addresses. For example, all addresses in the range 192.168.xxx.xxx are to be used for private addressing only. This address range is in common use in home routers, and if you have one, all the devices on your home network probably show up with addresses in this range. When they send messages to the outside world, though, those messages show up as if they came from your router's public IP address. Here's how it works:

My computer, with the address 192.168.1.45 on my home network, makes a request for a web page on a remote server. That request goes first to my home router. On my home network, the router's address is 192.168.1.1, but to the rest of the Internet, my router presents a public address, 66.187.145.75. The router passes my message on, sending it from its public address, and requesting that any replies come back to its public address. When it gets a reply, it sends the reply to my computer. Thanks to private addressing and subnet masks, multiple devices can share a single public IP address. This ability expands the total number of things that can be attached to the Internet.

to worry about configuring DNS. In some of this chapter's projects, you won't be going out to the Internet at large, so your devices won't have names, just numbers. When that happens, you'll need to know their numerical addresses.

NOTE: And when you go out to the Internet at large from a microcontroller, you first need to use a DNS utility on your computer to look up the numeric address for the hosts you want to talk to, then embed the numeric address in your microcontroller program. This is because there's just enough room in the Lantronix network modules used here to support basic networking functionality, and unfortunately, DNS support is usually not included.

Packet Switching: How Messages Travel the Net

So how does a message get from one machine to another? Imagine the process as akin to mailing a bicycle. The bike's too big to mail in one box, so first you break it into box-sized pieces. On the network, this is initially done at the Ethernet layer, also called the datalink layer, where each message is broken into chunks of more or less the same size, given a header containing the packet number. Next, you'd put the address (and the return address) on the bike's boxes. This step is handled at the IP layer, where the sending address and the receiving address are attached to the message in another header. Finally, you send it. Your courier might want to break the shipment up among several trucks to make sure each truck is used to its best capacity. On the Internet, this happens at the transport layer. This is the layer of the network responsible for making sure packets get to their destination. There are two protocols used to handle transport of packets on the Internet: Transmission Control Protocol, or TCP, and User Datagram Protocol, or UDP. You'll learn more about these later. The main difference between them is that TCP provides more error checking from origin to destination, but is slower than UDP. UDP trades off error checking for speed.

Each router sends off the packets one at a time to whatever routers it's connected to. If it's attached to more than one other router, it sends the packets to whichever router is least busy. The packets may each take a different route to the receiver, and they may take several hops across several routers to get there. Once they reach their destination, the receiver strips off the headers and reassembles the message. This method of sending messages in chunks across multiple paths is called packet switching. It ensures that every path through the network is used most efficiently, but sometimes packets are dropped or lost. You'll learn more on how that's handled in Chapters 5 and 6. For now, assume that the network is reliable enough that you can forget about dropped packets.

There's a command-line tool that can be useful in determining if your messages are getting through, called *ping*. It sends a message to another object on the net to say "Are you there?" and waits for a reply.

To use it, open up the command-line application on your computer (Terminal on Mac OS X, the Command Prompt on Windows, and xterm or similar on Linux/Unix). On Mac OS X or Linux, type the following:

```
ping -c 10 127.0.0.1
```

On Windows, type this:

```
ping -n 10 127.0.0.1
```

This sends a message to address 127.0.0.1 and waits for a reply. Every time it gets a reply, it tells you how long it took, like this:

```
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.166 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.157 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.182 ms
```

After counting ten packets (that's what the –c 10 on Mac and -n 10 on Windows means), it stops and gives you a summary like this:

```
--- 127.0.0.1 ping statistics ---
10 packets transmitted, 10 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.143/0.164/0.206/0.015 ms
```

It gives you a good picture not only of how many packets got through, but also how long they took. It's a useful way to learn quickly if a given device on the Internet is reachable or not, and how reliable the network is between you and that device. Later on, you'll be using devices that have no physical interface that you can see activity on, so you'll need *ping* to know whether they're working or not.

NOTE: 127.0.0.1 is a special address called the loopback address or localhost address. Whenever you use it, the computer you're sending it from loops back and sends the message to itself. You can also use the name localhost **in its place. You can test many network applications using this address, even when you don't have a network connection, as you'll see in future examples.**
X

Clients, Servers, and Message Protocols

Now you know how the Internet is organized, but how do things get done on the Net? For example, how does an email message get from you to your friend? Or how does a web page get to your computer when you type a URL into your browser or click on a link? It's all handled by sending messages back and forth between objects, using the transport scheme just described. When you know that works, you can take it for granted and concentrate on the messages.

How Web Browsing Works

Figure 3-4 is a map of the routes web pages take to reach your computer. Your browser sends out a request for a page to a web server, and the server sends the page back. Which route the request and the reply take is irrelevant, as long as there is a route. The web server itself is just a program running on a computer somewhere else on the Internet. A server is a program that provides a service to other programs on the Net. The computer that a server runs on, also referred to as a server, is expected to be online and available at all times so that the service is not disrupted. In the case of a web server, the server provides access to a number of HTML files, images, sound files, and other elements of a website to clients from all over the net. Clients are programs that take advantage of services. Your browser, a client, makes a connection to the server to request a page. To facilitate that, the computer that your browser is running on makes a connection to the computer that the server is running on, and the exchange is made.

The server computer shares its IP address with every server program running on it by assigning each program a port number. For example, every connection request

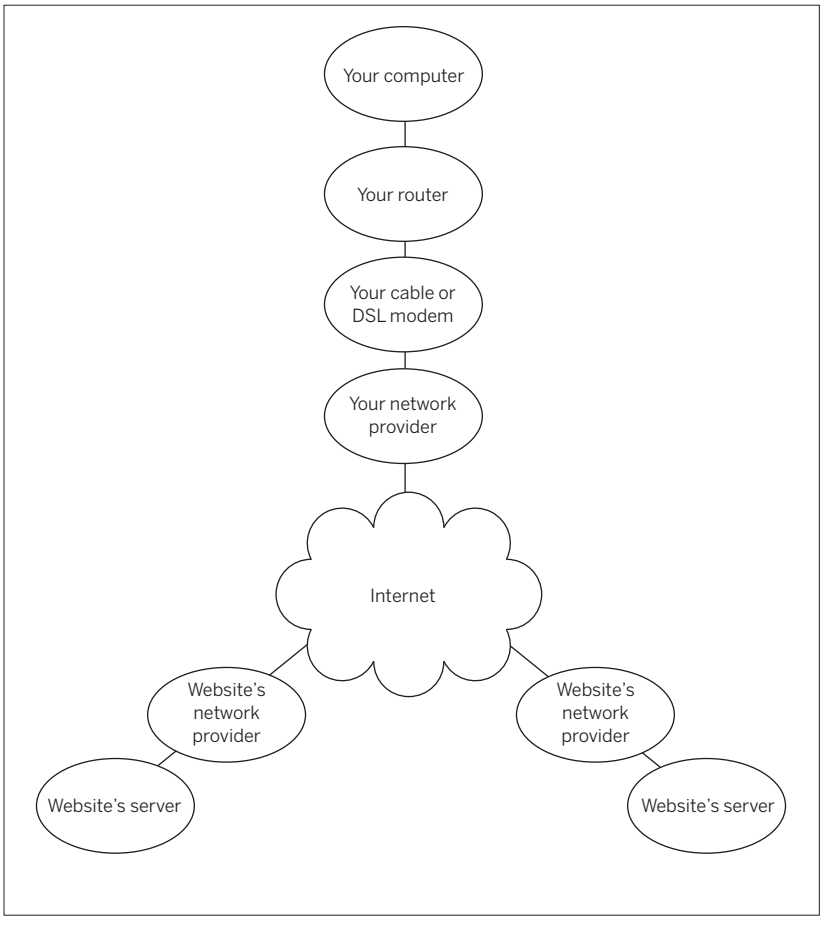


Figure 3-4
The path from a website to your browser. Although the physical computers are in many different locations, that doesn't matter to you, as long as you know the websites' addresses.

for port 80 is passed to the web server program. Every request for port 25 is passed to the email server program. Any program can take control of an unused port, but only one program at a time can control a given port. In this way, network ports work much like serial ports. Many of the lower port numbers are assigned to common applications like mail, file transfer, telnet, and web browsing. Higher port numbers are either disabled or left open for custom applications. (You'll write one of those soon.) A specific request goes like this:

1. You type http://www.makezine.com/index.html into your browser.
2. The browser program contacts www.makezine.com on port 80.
3. The server program accepts the connection.
4. The browser program asks for a specific file name, **index.html**.
5. The server program looks up that file on its local file system, and prints the file out via the connection to the browser. Then it closes the connection.
6. The browser reads the file, looks up any other files it needs (like images, movies, style sheets, and so forth), and repeats the connection request process, getting all the files it needs to display the page. When it has all the files, it strips out any header information and displays the page.

All of the requests from browser to server and all of the responses from server to browser (except the images and movie files) are just strings of text. To see this process in action, you can duplicate the request process in the terminal window. Open up your command program again just as you did for the *ping* example shown earlier. (If you're using Windows Vista, you may need to enable telnet with Control Panel→Programs→Turn Windows features on or off.)

Try It Type:

```
telnet www.google.com 80
```

The server will respond as follows (on Windows, you may see only a blank window):

```
Trying 64.233.161.147...
Connected to www.l.google.com.
Escape character is '^]'.
```

The built-in Windows version of telnet is not especially good. In particular, you won't be able to see what you type without setting the localecho **option, and the informative "Trying . . . Connected" prompts do not appear. You may want to try a replacement such as Dave's Telnet, aka dtelnet (**sourceforge.net/projects/dtelnet**).**

Type the following:

```
GET /index.html HTTP/1.0
HOST: www.google.com
```

You'll be using HTTP/1.0 requests in your code to keep things simple. Programs that make HTTP/1.1 requests are required to accept responses in chunks, which would complicate how you handle those responses. You may still see "HTTP/1.1" in the OK response you get from the server.

Press the Return key twice after this last line. The server will then respond with:

```
HTTP/1.0 200 OK
Cache-Control: private
Content-Type: text/html; charset=ISO-8859-1
Server: GWS/2.1
Date: Thu, 15 Mar 2007 14:58:20 GMT
Connection: Close
```

NOTE: If telnet doesn't close on its own, you may need to press Control-] to get to the telnet prompt, where you can type q followed by Enter to exit.

After the header, the next thing you'll see is a lot of HTML that looks nowhere near as simple as the normal Google web interface. This is the HTML of the index page of Google. This is how browsers and web servers talk to each other, using a text-based protocol called the hypertext transport protocol (HTTP). The http:// at the beginning of every web address tells the browser to communicate using this protocol. The stuff that precedes the HTML is the HTTP header information. Browsers use it to learn the types of files that follow, how the files are encoded, and more. The end user never needs this information, but it's very useful in managing the flow of data between client and server.

Remember the PHP time example from Chapter 1? It should still be sitting on your own web server, at www.example.com/time.php (replace www.example.com with the address of your server, which may be 127.0.0.1 if it's running on your local machine). Try getting this file from the command line.

Try It Modify the PHP program slightly, removing all the lines that print any HTML, like so:

Now telnet into the web server on port 80 and request the file from the command line. Don't forget to specify the HOST in your request, as shown earlier in the request to Google.

You should get a much more abbreviated response.

```
<?php
/* Date page
   Language: PHP
   Prints the date. */

// get the date, and format it:
$date = date("Y-m-d h:i:s\t");
// include the date:
echo "< $date >\n";

?>
```

Even though the results of this approach aren't as pretty in a browser, it's very simple to extract the date from within a Processing program or even a microcontroller program. Just look for the < character in the text received from the server, read everything until you get the > character, and you've got it.

HTTP requests don't just request files. You can add parameters to your request. If the file you're requesting is actually a program (like a CGI script), it can do something with those parameters. To add parameters to a request, add a question mark at the end of the request, and parameters after that. Here's an example:

http://www.example.com/get-parameters.php?name=tom&age=14

In this case, you're sending two parameters, name and age. Their values are "tom" and "14", respectively. You can add as many parameters as you want, separating them with the ampersand (&).

Test It Here's a PHP script that reads all the values sent in via a request and prints them out:

Save this script to your server as **get-parameters.php** and view it in a browser using the URL shown earlier (you may need to modify the path to the file if you've put it in a subdirectory). You should get a page that says:

```
name: tom
age: 14
```

```
<?php
/*
   Parameter reader
   Language: PHP

   Prints any parameters sent in using an HTTP GET command.
*/
// 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";

?>
```

» You could also request it from the command line like you did earlier (be sure to include the ?name=tom&age=14 at the end of the argument to GET, as in GET /get-parameters.php?name=tom&age=14). You'd get something similar, with the HTTP header:

```
HTTP/1.1 200 OK
Date: Thu, 15 Mar 2007 15:10:51 GMT
Server: Apache
X-Powered-By: PHP/5.1.2
Vary: Accept-Encoding
Connection: close
Content-Type: text/html; charset=UTF-8


<html><head></head><body>
name: tom<br>
age: 14<br>
</body></html>
```

» Of course, because PHP is a programming language, you can do more than just print out the results. Try this script:

Try requesting this script with the same parameter string as the last script, ?name=tom&age=14, and see what happens. Then change the age to something over 21.

NOTE: One great thing about PHP is that it automatically converts ASCII strings of numbers like "14" to their numerical values. Because all HTTP requests are ASCII-based, PHP is optimized for ASCII-based exchanges like this.

```
<?php
/*
   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.
*/
// 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";

?>
```

How Email Works

Transferring mail also uses a client-server model. It involves four applications: your email program and your friend's, and your email server (also called the mail host) and your friend's. Your email program adds a header to your message to say that this is a mail message, who the message is to and from, and what the subject is. Next, it contacts your mail server, which then sends the mail on to your friend's mail server. When your friend checks her mail, her mail program connects to her mail server and downloads any waiting messages. The mail server is online all the time, waiting for new messages for all of its users.

The transport protocol for mail is called SMTP, the Simple Mail Transport Protocol. Just like HTTP, it's text-based, and you can use it from a command line. When a mail server delivers a message, it has to figure out which servers are responsible for handling incoming mail (for example mail.example.com or smtp.example.com).

If you'd like to find the name of these servers, open a command window/terminal program, and use the *nslookup* command with the -q=mx option (this looks up the mail server for the domain you specify):

```
C:\ >nslookup -q=mx gmail.com
Server:  Unknown
Address:  192.168.254.1:53

Non-authoritative answer:
gmail.com       MX preference = 50,   mail exchanger = gsmtp163.google.com
gmail.com       MX preference = 50,   mail exchanger = gsmtp183.google.com
gmail.com       MX preference = 5,    mail exchanger = gmail-smtp-in.l.google.com
gmail.com       MX preference = 10,   mail exchanger = alt1.gmail-smtp-in.l.google.com
gmail.com       MX preference = 10,   mail exchanger = alt2.gmail-smtp-in.l.google.com
```

You could use any of the listed mail exchangers to send email to a gmail.com recipient, but don't. If you accidentally say the wrong thing to someone else's mail server (mistyping one of the SMTP commands, for example), your IP address might get reported to one of the organizations that tracks outbreaks of malicious software, and you could find yourself on a list of banned IP addresses.

Also, if you're connecting to the Internet through a cable or DSL modem, it's very likely that the SMTP port (25) is tightly controlled. Most ISPs allow you to connect only to *their* SMTP servers in order to prevent rogue users (and malicious software) from sending spam messages directly to recipients' SMTP servers. So for this kind of testing, it's best to use the SMTP server that your ISP specifies for outbound email rather than one of the mail exchangers you got from nslookup.

To start with, open a telnet connection to your outgoing SMTP server, like this:

```
telnet smtp.example.com 25
```

The server will respond something like this:

```
Trying 69.49.109.11...
Connected to mail.example.com.
Escape character is '^]'.
220 mail.example.com ESMTP Sendmail 8.13.1/8.13.1; Thu, 16 Mar
2006 16:04:22 -0500
```

Now you have to say hello, or in SMTP syntax, HELO. You must use the domain name of the email address you are sending mail from, for example: HELO example.com

The server will respond:

```
250 example.com  Hello, nice to meet you.
```

The dialogue goes on like this (when you see \r, press Return or Enter instead):

You send: `MAIL FROM: <you@example.com>\r`

Server responds: `250 Ok`

You: `RCPT TO: <friend@example.com>\r`

NOTE: You might want to send mail to yourself the first time, so you can check whether it works.

Server: `250 Ok`

You: `DATA\r`

Server: 354 enter mail, end with "." on a line by itself

The server now expects you to send several lines of text, starting with the subject, followed by the body of your email. Different servers expect you to end the message in different ways. Some look for a pair of carriage returns, but most look for a period on a line by itself — in other words: `\r.\r`

```
Subject: test message

Hello,
This is a test.
Goodbye.
```

Press Return twice at the end of your message, or Return followed by a period and then Return again, or whatever else your mail server asked for on the line that started with 354.

The server will respond like this: `250 Ok: queued as 12345`

You respond with: `QUIT\r`

And the server says: `221 Bye`

Finally, it closes the connection. Now check your mail to see whether you got a message from yourself. If you did, do the hokey pokey in celebration. If you didn't get a message, check your spam folder. Mail messages sent in this bare-bones fashion may be misconstrued as spam by eager-to-please email servers.

Because the whole mail transaction is text-based, you can make this happen from any program you want, whether it's on your personal computer or a microcontroller, as long as you've got an Internet connection.

X

Mail servers are pretty picky about what you type, and if you mistype something, pressing Backspace or Delete might confuse the mail server, even if everything looks OK on your end. If in doubt, close the telnet connection and try again from the beginning.

Project 4

A Networked Cat

Web browsing and email are all very simple for humans, because we've developed computer interfaces that work well with our bodies. Keyboards work great with our fingers, and mice glide smoothly under our hands. It's not so easy for a cat to send email, though. This project attempts to remedy that, and to show you how to build your first physical interface for the Internet.

MATERIALS

- **Between 2 and 4 force-sensing resistors** You have options: **Interlink 400 series FSRs** (www.interlinkelec.com). You can get these from Images Co (www.imagesco.com) or Trossen Robotics (www.trossenrobotics.com). The Interlink model 400 is shown in this project, but any of the 400 series will work well. **FlexiForce** from Parallax (www.parallax.com), part number 30056, also works well.
- **Between 2 and 4 10-kilohm resistors** available at many retailers, including from Digi-Key (www.digikey.com) as part number 10K-QBK-ND, Newark (www.newark.com) as part number 84N2322, and many others.
- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND or Jameco (www.jameco.com) part number 20601.
- **1 Arduino microcontroller module** (see Chapter 1)
- **1 personal computer**
- **1 web camera** (USB or FireWire)
- **1 webcam program**
- **1 cat mat**
- **1 cat** A dog will do if you have no cat.
- **2 thin pieces of wood or thick cardboard, about the size of the cat mat**
- **1 soft pad, about the size of the cat mat** A thick hand towel or dish towel will do.

If you're a cat lover, you know how cute they can be when they curl up in the sun in their favorite spot for a nap. You might find it useful in stressful times at work to think of your cat, curled up and purring away. Wouldn't it be nice if the cat sent you an email when he lays down for a nap? It'd be even better if you could then check in on the cat's website to see him at his cutest.

The system works like this: The force sensing resistors are mounted under the cat mat and attached to a microcontroller. The microcontroller is attached to a personal computer. There's also a camera attached to the personal computer. When the cat lies down on the mat, its extra weight will cause a change in the sensor readings. The microcontroller then sends a signal to a program on the personal computer, which calls a CGI script on a web server. The CGI script sends an email to the cat owner, notifying him that the cat is being particularly cute. Meanwhile, a separate program is uploading pictures of the cat to the web from a webcam attached to the computer.

Making a Web Page for the CatCam

The new piece of software in this project is a program to take pictures from the webcam and upload them. There are many good shareware and freeware packages available. A quick Google search for "webcam software" and the name of your operating system will turn up several. On Mac OS X, Evocam from Evological (www.evological.com) is a good shareware package. If you use it, please pay your shareware fees. On Windows, Fwink (lundie.ca/fwink) is a good basic freeware package.

All of these applications share some common attributes. There is a configuration menu or panel that lets you choose your camera and control its brightness, contrast, and other settings. There is also one for setting the address of a server to upload to, and for specifying how often to do so.

Any USB camera should work well on Windows. On Mac OS X, you can also use FireWire cameras like Apple's iSight or Unibrain's fire-I. Many of the USB cameras won't work too well on Mac OS X without a driver, but there's a good open-source driver called macam, available on SourceForge at webcam-osx.sourceforge.net, that works with a number of the USB cameras. To use it with applications

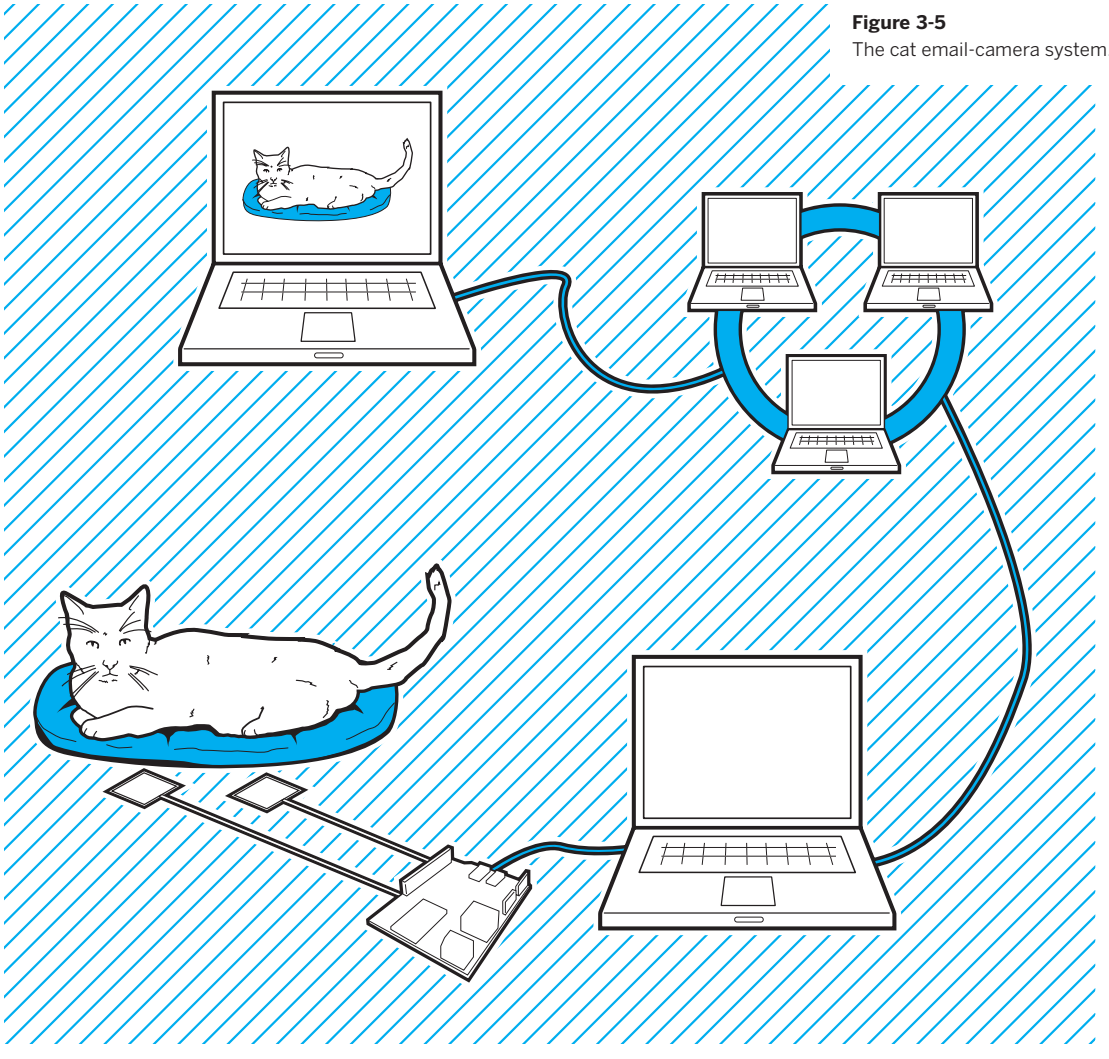
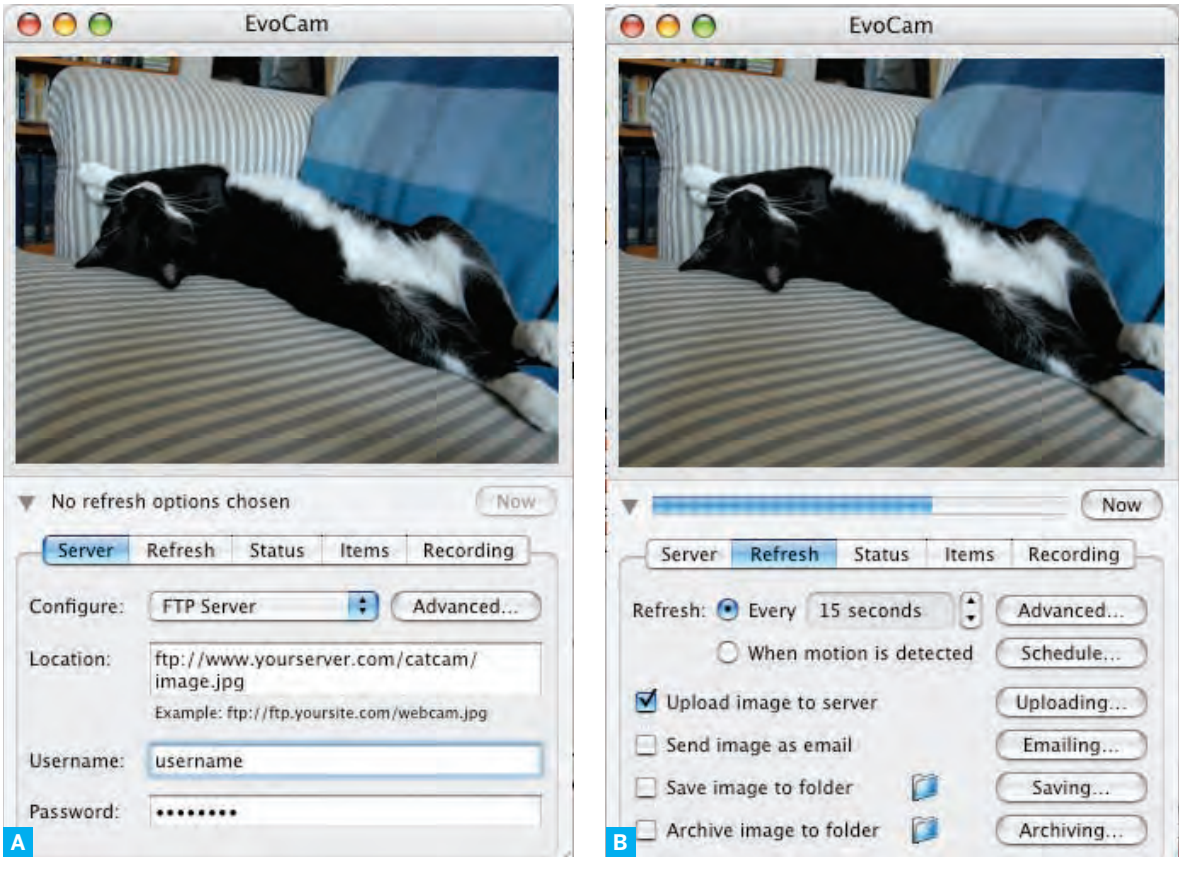


Figure 3-5
The cat email-camera system.

other than the macam program itself, copy the file called **macam.component** to the **/Library/Quicktime** directory of your hard drive. Then open the webcam software you're using and your USB camera should pop up in the list of available cameras.

Once you've got a camera image showing up onscreen, open the FTP settings menu or the configuration panel. Enter the address for your web server, the path to the directory that you want to upload the file to, the filename, and your user name and password. The software will save a picture to a file and then upload it to the server. There is also a setting to control how often a new image is uploaded. Set it for every 15 seconds or so. Don't set it to update too frequently, or you'll create too heavy a load on the server, and viewers won't even get a full image before you're overwriting it with a new one. Remember the first rule of good networking habits: listen more than you speak. Make sure your software isn't uploading so frequently that users can't access the page.

Make a new directory on your server for this project, call it **catcam**, then have the webcam software upload the image to that directory with a clever filename like **image.jpg**. To make sure it got there, open a web browser and see if you can see the image in the directory you set the camera



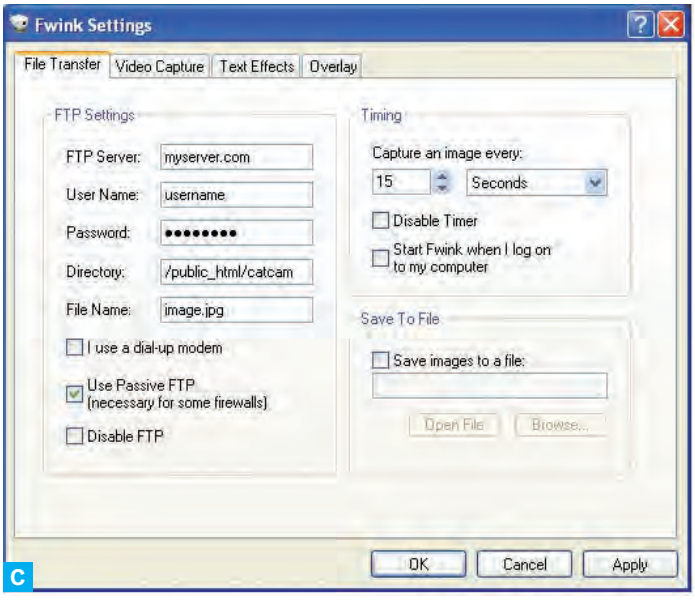


Figure 3-6
Evocam's server panel (A) allows you to set the net address to upload images to. The Refresh panel (2) allows you to set the upload frequency. On Windows, Fwink's Settings panel (C) controls both settings.

to upload to. For example, if your directory is at the root of your website, then the image would be found at www.example.com/catcam/image.jpg.

Once you know the image is there and visible, frame it with a web page in the same directory, called **index.html**. Below is a bare-bones page that will automatically refresh itself in the user's browser every ten seconds. The meta tag in the head of the document causes the browser to refresh the page every ten seconds. Feel free to make the page as detailed as you want, but keep the meta tag in place.

```
<html>
<head>
    <title>noodles</title>
    <meta http-equiv="refresh" content="10">
</head>
<body>
    <center>
        <h2>Cat Cam</h2>
            <img src="http://www.example.com/catcam/image.jpg">
    </center>
</body>
</html>
```

Putting Sensors in the Cat Mat

Now that the catcam is running, it's time to make the system that notifies you when the cat is there to be viewed. How you do this depends on what kinds of force-sensing resistors you use. Interlink's 400 series FSRs include a long, thin model with adhesive backing that mounts nicely on any firm surface. Because this sensor is very flat and has a relatively large surface area, it gives good readings for this project. Mount the FSRs on strips of firm yet flexible wood or cardboard (Masonite works well), and you've got a great sensor.

If you're using smaller FSRs from another company like CUI or FlexiForce, you'll need to make a larger sensing pad. First, cut two pieces of wood or firm cardboard slightly smaller than the cat's mat. Don't use a really thick or hard piece of wood. You just need something firm enough to provide a relatively inflexible surface for the sensors. Attach the sensors to the corners of one of the pieces of wood or cardboard. Sandwich the sensors between the two boards. Tape the two boards together at the edges loosely, so that the weight of the cat can press down to affect the sensors. If you tape too tightly, the sensors will always be under force; too loose, and the boards will slide around too much and make the cat uncomfortable. If the sensors don't give enough of a reaction, get some little rubber feet, available at any electronics store, and position them on the panel opposite the sensors so that they press down on the sensors. If the wood or cardboard panels have some flex in them, position an extra rubber foot or two at the center of the panel to reduce the flex. Figures 3-7, 3-8, and 3-9 show a working version of the sensor board.

Next, attach long wires to the force-sensing resistors to reach from the mat to the nearest possible place to put the microcontroller module. Connect the sensors to an analog input of the microcontroller using the voltage divider circuit shown in Figure 3-9. Because you don't care which of the sensors gets triggered, this circuit makes it possible to react to input from any of the four of them.

X

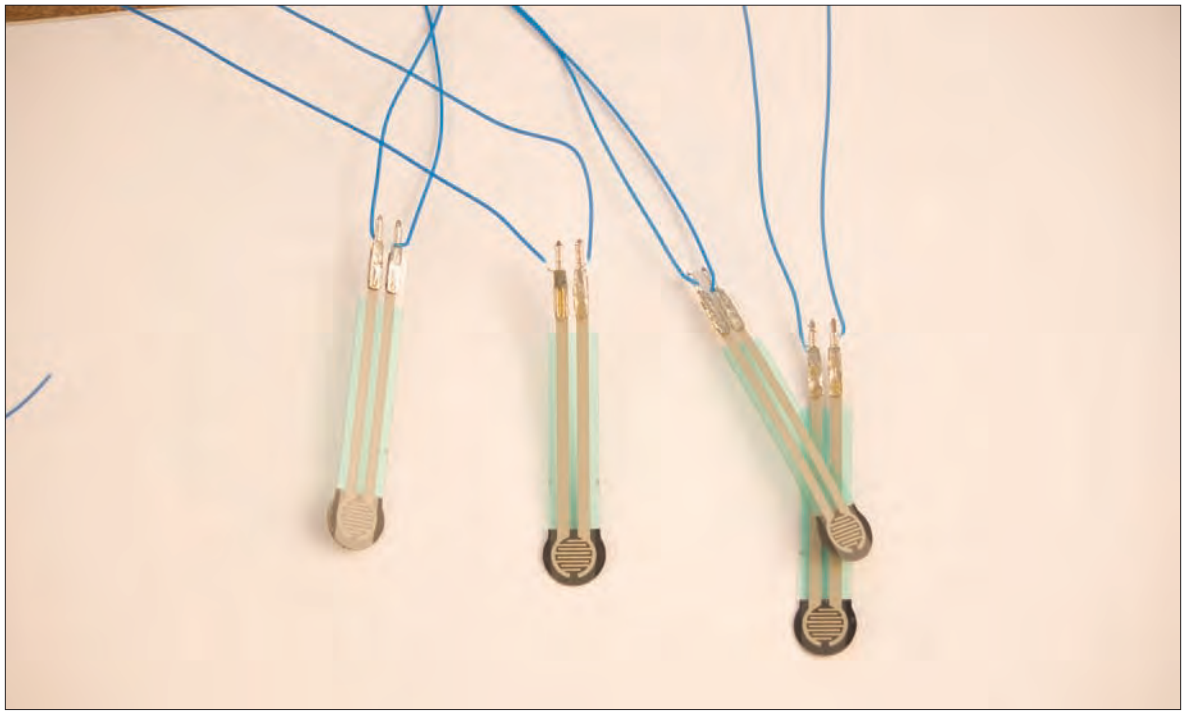
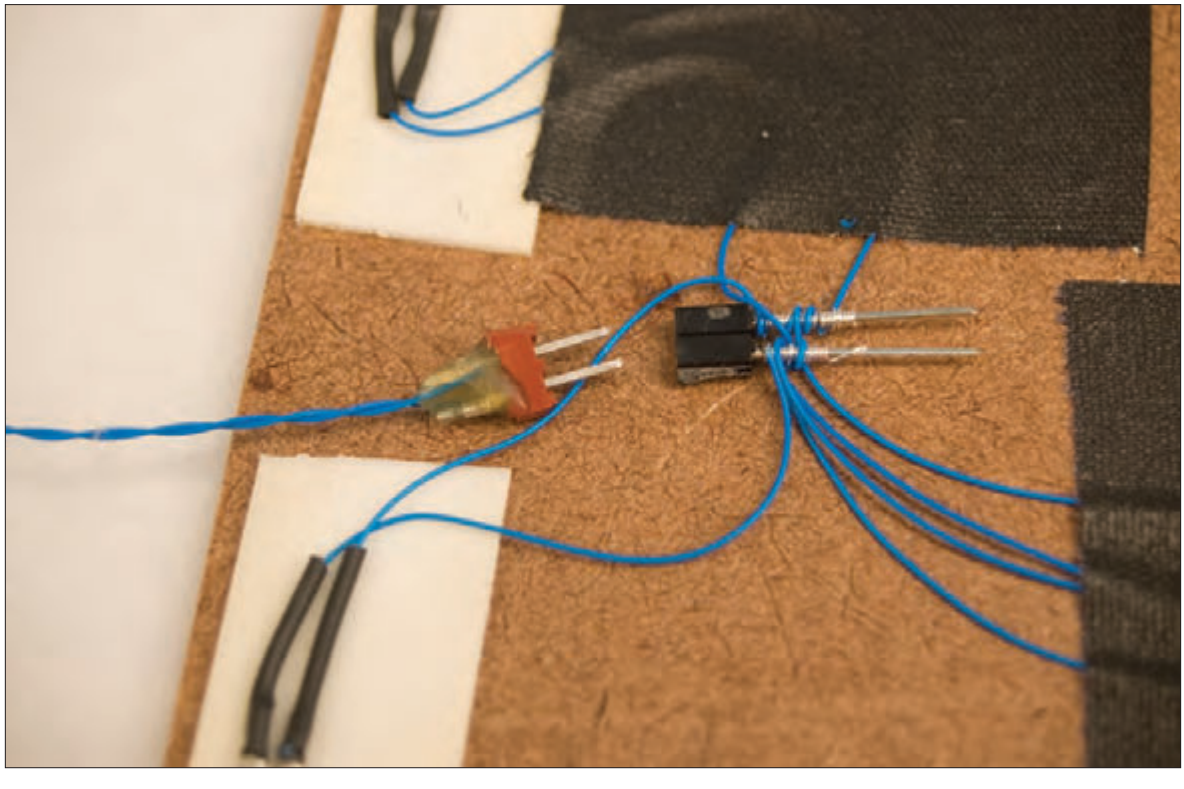
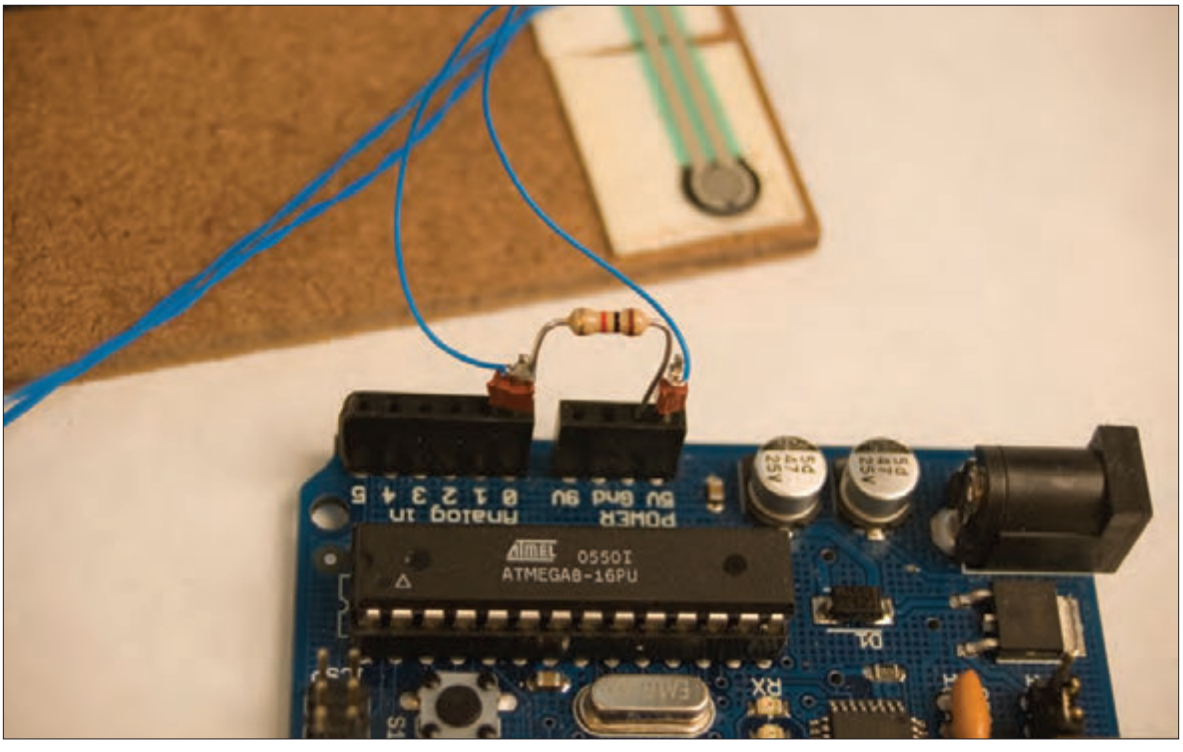


Figure 3-7
Because the force-sensing resistors melt easily, I used 30AWG wire wrap instead of solder. Wire-wrapping tools are inexpensive and easy to use, but make a secure connection. After wire wrapping, I insulated the connections with heat shrink.

Figure 3-8
Cat-sensing panel. The four FSRs are wired in parallel. Note the rubber feet that press down more precisely on the sensors. Make sure to insulate the connections before taping the panels together. The connector is just a pair of female wire wrap headers.





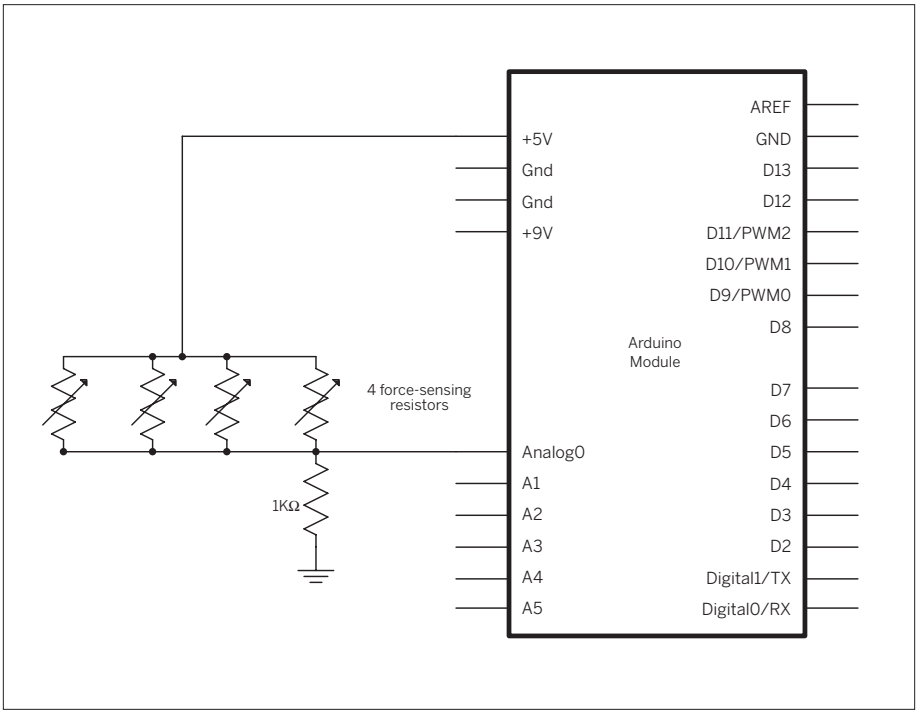


Figure 3-9
The cat-sensing circuit. Because all of the force-sensing resistors are all wired in parallel, there are only two connections for all of them. This circuit is simple enough that you can just solder a resistor to a header pin to make the connection to ground.

Test It

Once you've got the sensor panel together and connected to the microcontroller, run the following on the Arduino or Wiring board to test the sensors:

To see the results, open the Serial Monitor at 9600 bits per second. Now position the cat on the panel and note the number change. This can be tricky, as cats are difficult to command. You may want to put some cat treats or catnip on the pad to encourage the cat to stay there. When you're satisfied that the system works and that you can see a significant change in the value when the cat sits on the panel, you're ready to move on to the next step.

NOTE: Once you've got the serial connection between the microcontroller and the computer working, you might want to add in the Bluetooth radio from the Monski pong project in Chapter 2. It will make your life much easier if your computer doesn't have to be tethered to the cat mat in order to program.

```
/*
   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 sensors on Analog in 0
*/
int sensorValue;            // outgoing ADC value

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

void loop()
{
  // read analog input:
  sensorValue = analogRead(0);

  // send analog value out in ASCII decimal format:
  Serial.println(sensorValue, DEC);

  // wait 10ms for next reading:
  delay(10);
}
```

Connect It

As the microcontroller can't connect to the Internet on its own, you'll need another computer for that. Your computer and Processing will do the job well. Here's a Processing program similar the Monski pong program from Chapter 2 to read the sensors:

```
/*
 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.
 */

import processing.serial.*;

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());
```

You don't want Processing sending a message constantly, because you'd get several thousand emails every time the cat sits on the mat. Instead, you want to recognize when the cat's there, send an email, and don't send again until he's left and returned again. If he jumps on and off and on again in a minute or less, you don't want to send again.

What does that look like in sensor terms? To find out, you need to do one of two things: either get the cat to jump on and off the mat on cue (difficult to do without substantial bribery, using treats or a favorite toy) or weigh the cat and use a stand-in of the same weight. The advantage to using the cat is that you can see what happens when he's shifting his weight, preparing the bed by kneading it with his claws, and so forth. The advantage of the stand-in weight is that you don't have to herd cats to finish the project.

Continued from previous page.

```
  // 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 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 off the carriage return and convert the string to an integer:
    sensorValue = int(trim(myString));

    // print it:
    println(sensorValue);
  }
}
```

Refine It

If your system is working correctly, you should notice a difference of several points in the sensor readings when the cat gets on the mat. It helps to graph the results so you can see clearly what the difference looks like. To do that, add an extra variable to the variable list at the beginning of your Processing program:

```
int graphPosition = 0;  // the horizontal position of the latest
                        // line to be drawn on the graph
```

» Then add this method at the end of your program:

```
void drawGraph() {
  // adjust this formula so that lineHeight is always less than
  // the height of the window:
  int lineHeight = sensorValue /2;

  // draw the line:
  stroke(0,255,0);
  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++;
  }
}
```

» Call this method from the serialEvent() method, right after you print the number:

```
    // print it:
    println(sensorValue);
    drawGraph();
  }
```

When you run the program, you'll see a graph of the sensor values. When the cat jumps on the mat, you should see a sudden increase, and when he jumps off, you'll see the graph decrease. You'll also see any small changes, which you might need to filter out. If the changes are small relative to the difference between the two states you're looking for, you can ignore them. You have enough knowledge to start defining the cat's presence on the mat as an event, using the sensor values. To do this, pick a threshold number in between the two states. When the sensor reading goes above the threshold, send a message that he's in place. When the sensor value goes below the threshold, the cat has left the mat, and the event is over. Once you've sent a message, you don't want to send another one right away, even if the cat gets off the mat and back on. Decide on an appropriate interval, wait that long, and start the whole process again.

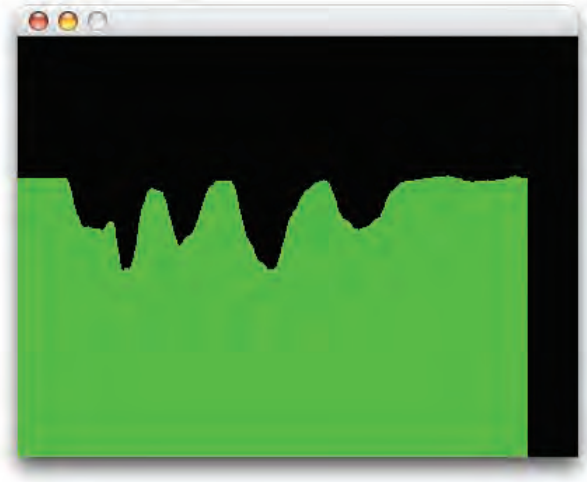


Figure 3-10
Output of the sensor graphing program.

Add the following new variables to the beginning of your program:

```
int prevSensorValue = 0;    // the previous sensor reading
boolean catOnMat = false;   // whether the cat's on the mat;
int threshold = 320;        // above this number, the cat is on the mat.
```

Then put this code in your draw() method, which Processing runs in a continuous loop. So far, you haven't put any code in the draw() method, so this is the only code there:

```
  if (sensorValue > threshold ) {
    // if the last reading was less than the threshold,
    // then the cat just got on the mat.
    if (prevSensorValue <= 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;
```

Make the following change at the beginning of the drawGraph() method; change this:

```
    // draw the line:
    stroke(0,255,0);
```

to this:

```
  // draw the line:
  if (catOnMat) {
    // draw green:
    stroke(0,255,0);
  }
  else {
    // draw red:
    stroke(255,0,0);
  }
```

Finally, add a method that sends mail. For now, it will just print a placeholder to the message window. After the next section, you'll write code to make it send mail for real. Add this method to the end of your program:

```
void sendMail() {
  println("This is where you'd send a mail.");
}
```

When you run the program, the graph should draw in red when the cat's off the mat, and green when it's on. Sometimes, if the cat takes his time getting settled, you can get several mail messages in a second or two. You may notice that the graph switches from red to green several times as it crosses the threshold. This happens because the sensor fluctuates slightly, due to electrical noise. Not all sensors are noisy, and you may be lucky and find out that yours isn't. If it is, you can smooth the transition out slightly by changing your code. Change the first block of the draw() method to read as follows:

```
if (sensorValue > threshold ) {
  // if the last reading was less than the threshold,
  // then the cat just got on the mat.
  if (prevSensorValue <= threshold) {
    // wait a a bit, then check again to see whether the reading
    // is still above the 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;
  }
}
```

When you've got very noisy readings, this method, called debouncing, is very useful. But one hundred milliseconds is a long time in sensor terms, and you're changing the whole system's reaction time by adding this debounce delay. You may even notice the graph pausing slightly at the transitions. Adjust the delay to be as short as possible and still deliver reliable results.

NOTE: Even with the debounce routine in your program, it's possible to get several mail messages a minute, if the cat is fickle. What's needed is a minimum time threshold between mail messages.

Tame It

Every time the program sends a mail message, it should take note of the time, and not send a message again no matter what, unless the time threshold has passed. To make this happen, add a couple new variables at the beginning of the program:

```
int timeThreshold = 1;     // minimum number of minutes between emails
int timeLastSent[] = {
  hour(), minute() - 1 };  // time the last message was sent
```

Now modify the sendMail() method as follows:

Once you're sure it works, adjust timeThreshold to an appropriate minimum number of minutes between emails.

```
void sendMail() {
  // calculate the current time in minutes:
  int[] presentTime = {hour(), minute()};

  // print the sensor value, the current time,
  // and the last time you sent a message, separated by tabs:
  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 that 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();
    }
  }
}
```

Sending Mail from the Cat

Once you've got the Processing program recognizing when the cat lies on the mat, you need to get it to send an email. You could write a program to send an email directly from Processing, using the text strings described in the section on email at the beginning of this chapter, but it's easier to send mail using PHP. The next section shows you how to pass the message on from Processing to PHP to do that. The same technique shown here can be used to call any PHP script from Processing. First, you need to program PHP to send a mail message. The PHP script below takes advantage of PHP's ability to read parameters from the HTTP request:

```
<?php
/*
   Cat On Mat
   Language: PHP

   Expects a parameter called SensorValue, an integer.
   Prints a custom message depending on the value of SensorValue.
*/

$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";
?>
```

▸▸ This value should match the value of the threshold variable from the Processing sketch.

▸▸ In order for the PHP scripts to run, you'll need to install them on a web server that supports PHP. There are many web hosting companies with inexpensive (less than $10 a month) web hosting plans that support PHP.

Save it to your server with the name **cat-script.php**. Test it from a browser with this HTTP request, using different values for sensorValue. Use a URL similar to the one shown here, replacing www.example.com with your server name, and adjusting the path to the script as needed: http://www.example.com/catcam/cat-script.php?sensorValue=12. Any value above 320 should result in a message that the cat is on the mat. When you're satisfied that it works, change your script to make it send an email (change yourname@example.com to your real email address), as shown in the code section below.

Call the script again and then check your mail to see whether the message went through. Some mail servers may require that you send mail only from your proper account name. If that's the case, replace cat@example.com in the send_mail function with your account name on the server that the script is running on.

```
<?php
/*
   Mail sender
   Language: PHP

   Expects a parameter called SensorValue, an integer
   Sends an email if sensorValue is above a threshold value.
*/

$threshold = 320;    // minimum sensor value to trigger action.
                     // change this value to whatever your sensor threshold is.

// 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");
}
?>
```

You'll need to change this number.

You'll need to change this URL.

You'll need to change these email addresses.

Now that you're sending emails from a program, you need to be very careful about how often it happens. You really don't want 10,000 messages in your inbox because you accidentally called the mail command in a repeating loop.

Putting It All Together

Finally, it's time to get Processing to call the PHP script and complete the connection from the cat to your inbox. To do this, you're going to use the net library in Processing. Like the serial library, it adds some functions to the core of Processing. The serial library allowed you to access the serial ports, and the net library allows you to make network connections. Here's an example that uses the net library to make an HTTP call to the PHP script you just wrote. Use it to confirm that Processing can contact your server:

```
/*
    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);
```

» You'll need to change the hostnames and the path.

Continued from previous page.

```
        // note that the request is over:
        requestInProgress = false;
        // reset the string  for future requests:
        responseString = "";
      }
    }
  }
}
```

If you check your mail, you should have a message from the PHP script indicating that the cat is on the mat. Once that's working, it's time to combine this sketch with the cat-sensing sketch shown earlier. First, add the net library import right after the serial library import:

```
import processing.net.*;  // gives you access to the net library
```

Then add the global variables from the HTTP client script to the variable list at the beginning of the sensing script:

```
// 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
```

To send the mail and check the response from the server, add two new methods at the end of the sketch, makeHTTPCall() and checkNetClient() (be sure to change the references to `example.com` and the path of the requestString):

```
void makeHTTPCall() {
  // do this only if you're not already in the middle of an HTTP request:
  if (requestInProgress == false) {
    // Open a connection to the host:
    client = new Client(this, "example.com", 80);

    // form the request string:
    String requestString = "/catcam/cat-script.php?sensorValue=" + sensorValue;

    // send the HTTP GET request:
    client.write("GET  " + requestString + " HTTP/1.0\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());
```

» You'll need to change the hostnames and the path.

Continued from opposite page.

```
    // add to a line of |'s on the screen (crude progress bar):
    print("|");
  }
  // if there are 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 = "";
      }
    }
  }
}
```

» Call makeHTTPCall() in the sendMail() method, in the two places where you're currently printing out "This is where you'd send a mail," like so:

```
      println("This is where you'd send a mail.");
      makeHTTPCall();
```

» Call checkNetClient() at the end of the draw() method like so:

```
  if (requestInProgress == true) {
    checkNetClient();
  }
```

Now run the sketch. When the sensor reading goes above the threshold, if an appropriate number of minutes has passed, you should see a mail message go out. Now the cat can send you email when he's curled up in his bed.

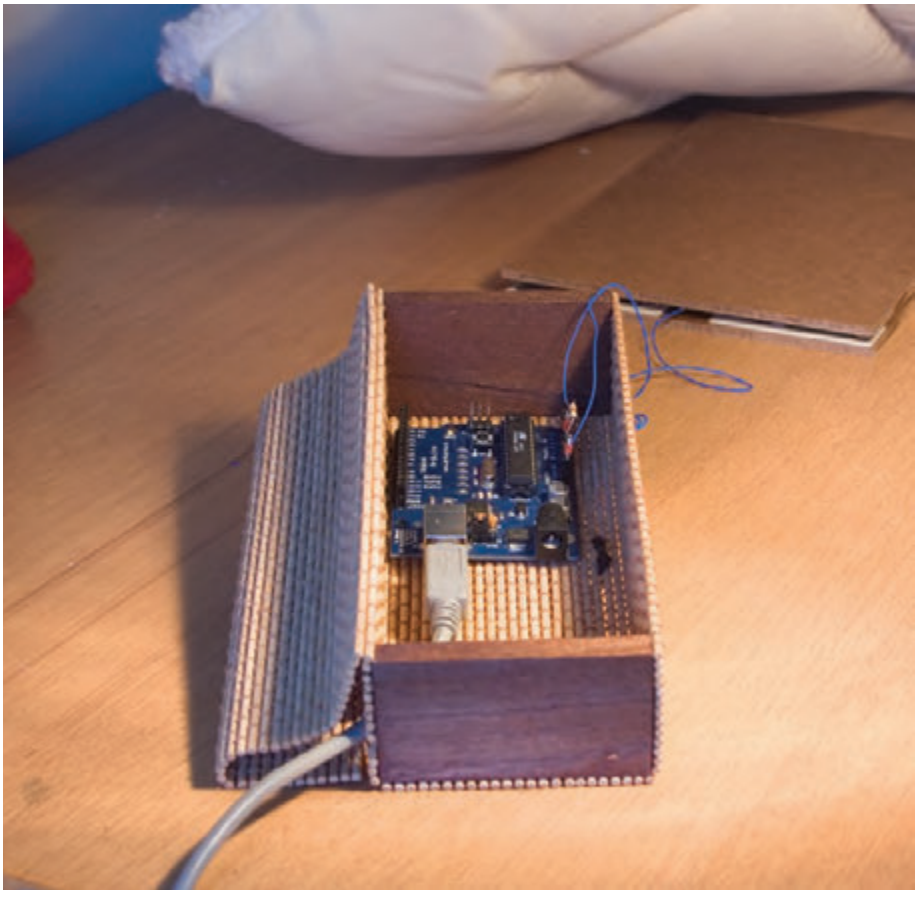


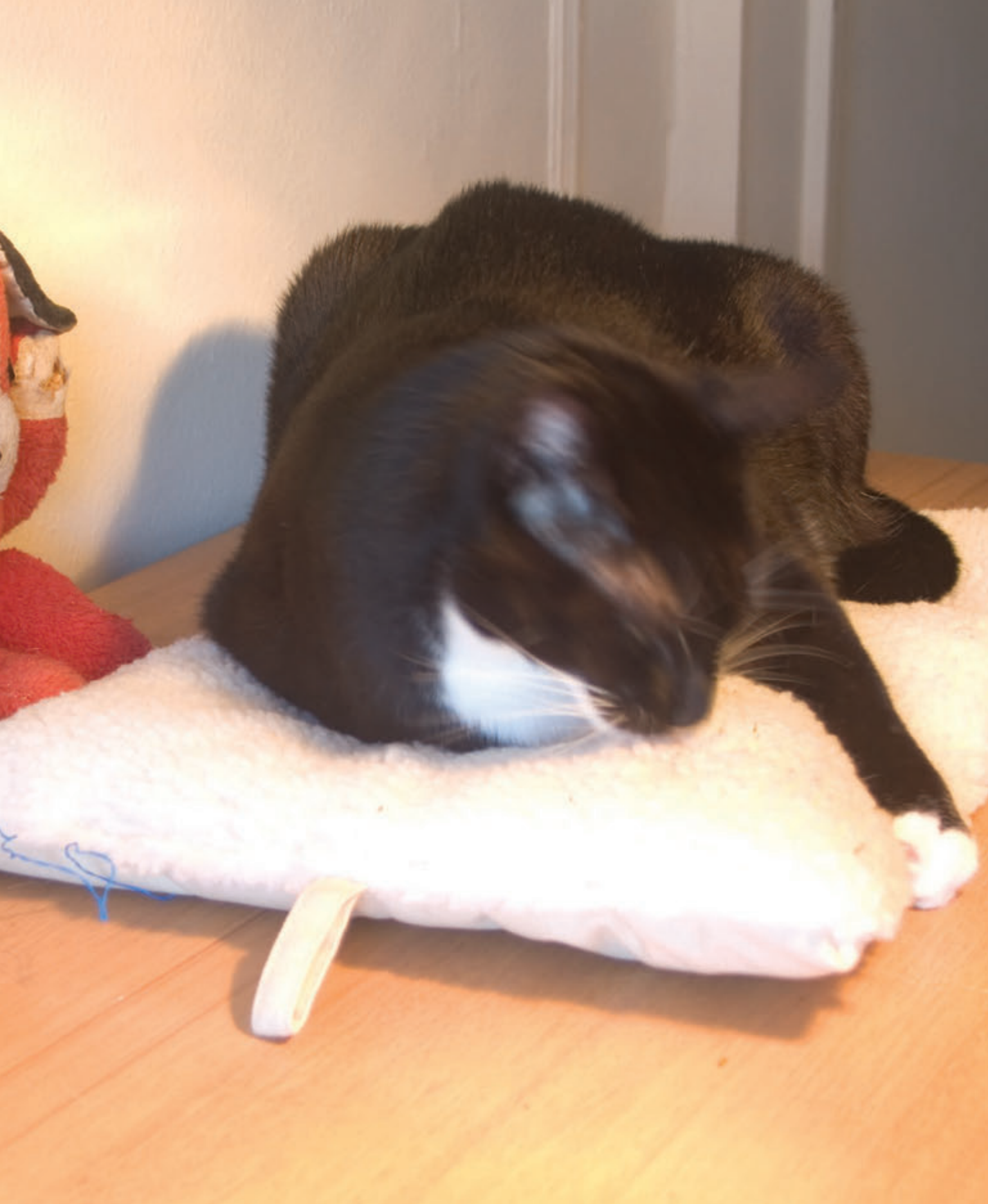
Figure 3-11
The finished cat bed (at right) and a detail of the sensor pad, which sits under the cat bed itself. A bamboo jewelry box from a nearby gift store houses the electronics, and matches the furniture. The USB cable runs to the computer. Make sure to secure the wires throroughly, or the cat may try to chew on them.

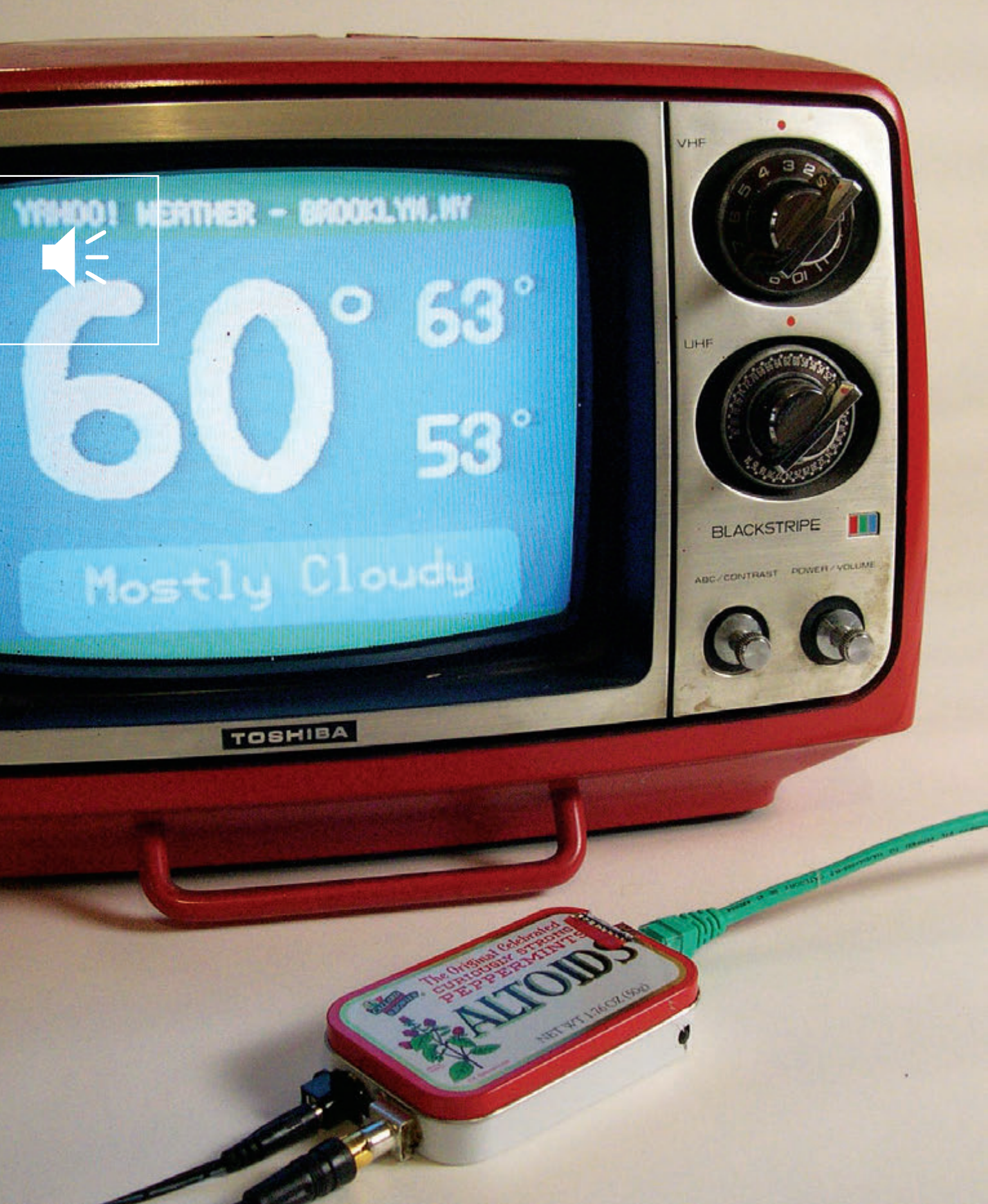
Conclusion

Now you've got an understanding of the structure of the Internet, and how networked applications do their business.

The Internet is actually a network of networks, built up in multiple layers. Successful network transactions rely on there being at least one reliable route through the Net from client to server. Client and server applications exchange strings of text messages about the files they want to exchange, transferring their files and messages over network ports. To communicate with any given server, you need to know its message protocols. When you do, it's often possible to test the exchange between client and server using a telnet session and typing in the appropriate messages. Likewise, it's possible to write programs for a personal computer or microcontroller to send those same messages, as you saw in the cat bed project. Now that you understand how simple those messages can be, you'll get the chance to do it without a personal computer in the next chapter, connecting a microcontroller to the Internet through a serial-to-Ethernet converter that's not much bigger than the microcontroller itself.

X




YAHOO! WEATHER - BROOKLYN, NY
60° 63°
53°
Mostly Cloudy
VHF
UHF
BLACKSTRIPE
ABC / CONTRAST
POWER / VOLUME
TOSHIBA
The Original Celebrated
Curiously Strong
PEPPERMINTS
ALTOIDS

4

MAKE: PROJECTS

Look, Ma, No Computer! Microcontrollers on the Internet

The first response that comes to many people's minds after building a project like the networked cat bed in Chapter 3 is: "Great, but how can I do this without needing to connect to my computer?" It's cumbersome to have to attach the microcontroller to a laptop or desktop computer just to enable it to connect to the Internet. After all, as you saw in Chapter 3, Internet message protocols are just text strings, and microcontrollers are good at sending short text strings. So in this chapter, you'll learn how to connect a microcontroller to the internet through a device that's not much more complex than the Bluetooth radio modem you used in Chapter 2.

◀◀ **Uncommon Projects' YBox** (http://ybox.tv) puts RSS feeds on your TV using an Xport serial-to-ethernet module and a Propeller microchip. *Image courtesy of Uncommon Projects.*

In the past few years, a wide array of commercial appliances has come on the market that can connect directly to the Internet without the aid of a personal computer. Companies like D-Link, Sony, Axis, and others make security cameras with network interfaces, both Ethernet and Wi-Fi. Ceiva, eStarling, and others make picture frames with Wi-Fi connections to which you can upload images from the Net. Ambient Devices makes lamps and displays of various sorts that connect to the Net and change their appearance based on changes in information like stock market data, weather, and other scalar quantities. Cable television set top boxes are computers in a small box, capable of routing streams of audio, video, and data all at the same time. In fact, the operating system in your set-top box might even be a variation of the same Linux operating system that's running on your network provider's web hosting machine. Home alarm systems are made up of networks of microcontrollers that talk among themselves, with one that can communicate with a central server, usually over phone lines using a modem.

All of these appliances engage in networked communication. The simplest ones handle only one transaction at a time, requesting information from a server and then waiting for a response, or sending a single message in response to some physical event. Others manage multiple streams of communication at once, allowing you to surf the Web while watching television. The more processing power a given device has, the more it can handle. For many applications, however, you don't need a lot of processing power, because the device you're making has only one or two functions.

x

Introducing Network Modules

It's possible to write a program for a microcontroller that can manage all the steps of network communication, from the physical and data connections to the network address management to the negotiation of protocols like SMTP and HTTP. A code library that encompasses all the layers needed for network connections is called a network stack, or TCP/IP stack. However, it's much easier to use a network interface module to do the job.

There are many such modules on the market, with varying prices and features. Just as you can choose how technical you want to get when you pick a microcontroller platform, you can also choose your technical level when you pick a network controller. Some modules, like Rabbit Semiconductor's RabbitCore processors, come with all the source code for a TCP/IP stack, and expect you to modify it for your needs and program the device yourself. This module is powerful, but has a steep learning curve. Others have a stack programmed into their firmware, and present you with a serial, telnet, or web-based interface. These are much simpler to use. The web interface gives you access from the browser of your personal computer; the telnet interface gives you access from a server or your personal computer; and the serial interface gives you access from a microcontroller.

The projects in this chapter, and some of the others later in the book, use a family of network modules from Lantronix: the Micro, the XPort, the WiMicro, and the WiPort. These are serial-to-Ethernet modems; as you'll see shortly, they work much like the Bluetooth modems you used in Chapter 2 but have a different serial protocol. The Micro and the XPort require a wired connection on both the serial side and the Ethernet side. The WiMicro and WiPort are wireless counterparts of the Micro and XPort, respectively. Of the four, the XPort is the least expensive and the smallest, about the size of a normal Ethernet jack that you'd find in a computer. It's not designed to be used with a solderless breadboard, though, so you have to make your own printed circuit board for it, or buy one. The WiPort also requires you to make your own board. The Micro and the WiMicro are in the middle of the price range, and both have a more convenient physical interface that can be connected to a breadboard with relative ease.

x

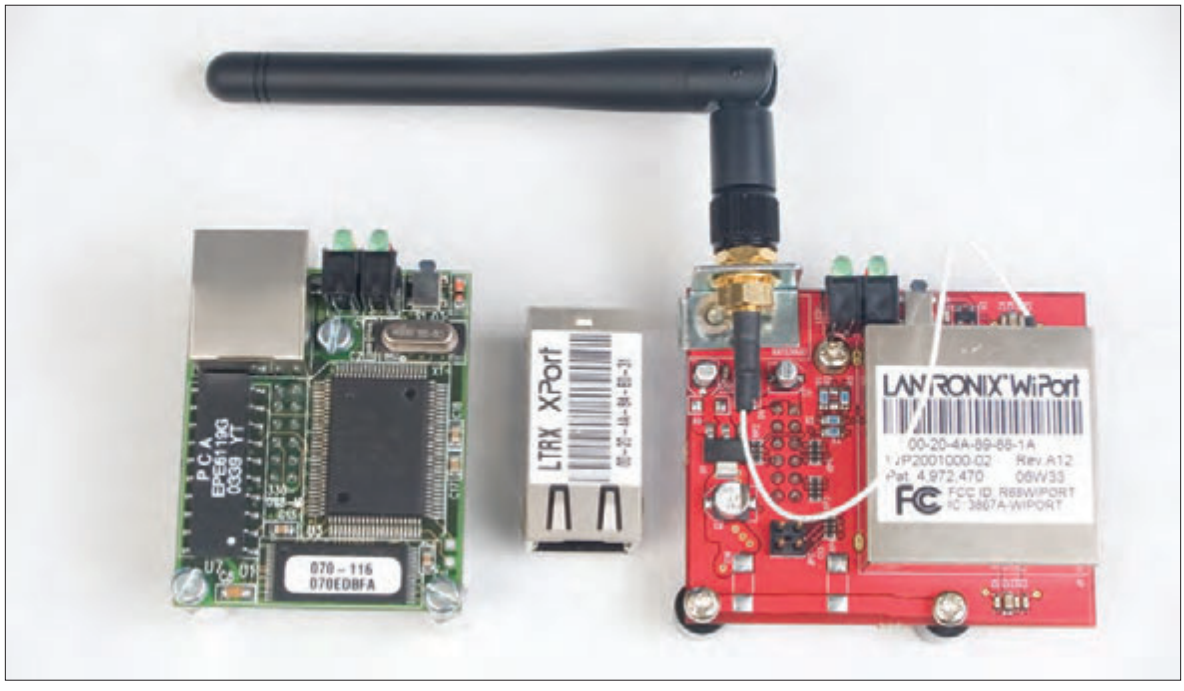


Figure 4-1
The Lantronix Micro (left), XPort (center), and WiMicro (left) serial-to-Ethernet modules. The WiPort is the square silver part of the WiMicro.

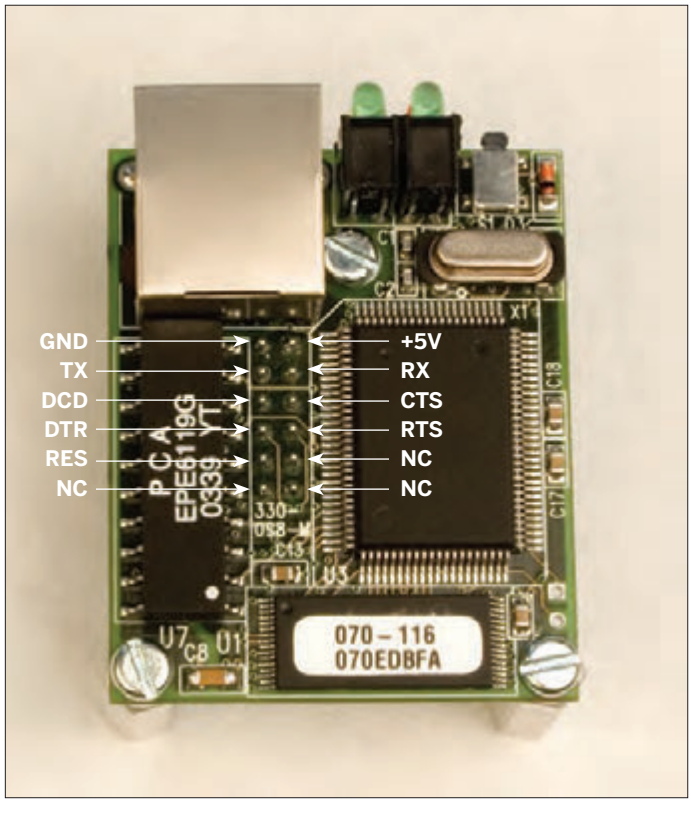


Figure 4-2
Pin configurations for the Micro. The configuration for the WiMicro is identical.

Project 5

Hello Internet!

The first thing you need to do in order to use any network module is to get it to connect to the network and to return any messages it gets through its serial port. That's the goal of this project.

MATERIALS

- **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)
- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, Jameco (www.jameco.com) part number 20601
- **1 micro-to-breadboard connector**
- **1 USB-to-serial circuit** such as the FT232RL or RS-232-to-serial circuit such as the MAX3323, as shown in Chapter 2. Use the second of these if you don't have a spare USB port and plan to use an RS-232 serial port. Otherwise, use the first one.
- **1 7805 5V voltage regulator**
- **1 10µF capacitor**
- **1 1µF capacitor**

For micro-to-breadboard connector:

Option 1 A quick-and-dirty adapter:

- **2 rows of right-angle male headers** (such as Samtec TSW series, TSW-112-08-T-S). Most retailers carry some version of these. Jameco's equivalent is part number 103351.
- **2 rows of straight female headers** (such as Samtec SSA series, SSA-106-S-T). Most retailers carry some version of these. Jameco's equivalent is part no. 308567. I prefer the Samtec ones, because they're easier to break off.

Option 2 A DIY ribbon cable adapter:

- **14-pin IDC DIP plug** Jameco part number 42658
- **IDC 14-pin socket** Jameco part number 153948
- **14-conductor rainbow-colored ribbon cable** Jameco part number 105672

Making the Circuit

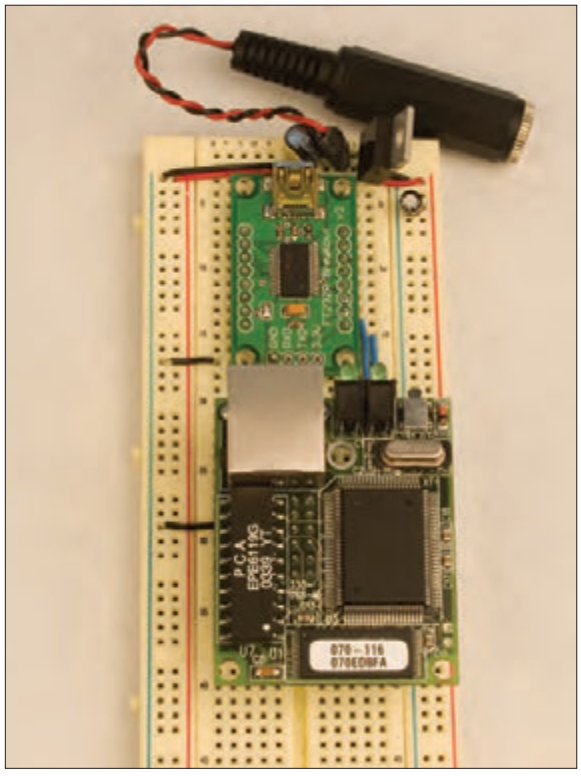
All of the Lantronix modules shown in Figure 4-1 have at least the following connections:

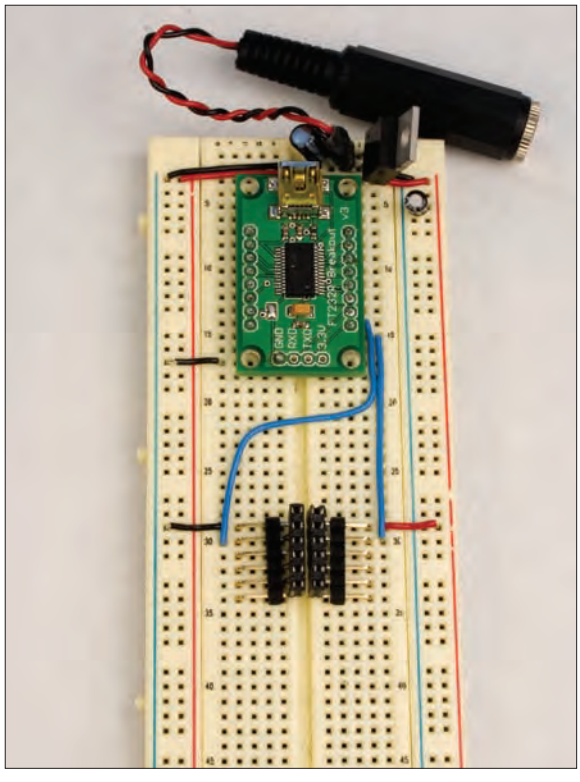
- Power
- Ground
- Serial receive (RX)
- Serial transmit (TX)
- Reset

These are the connections you'll use to connect to your microcontroller. The additional pins are for things like a second serial port (Micro, WiPort, and WiMicro) or user-configurable I/O pins (XPort), which you won't need for the projects in this book. As you might expect by now, serial receive (RX) connects to the microcontroller's serial transmit (TX), and vice versa. Figure 4-2 shows the pin configurations for the Micro. This chapter features the Micro, and later chapters feature the XPort. Once you know the circuit for each, they can be used interchangeably for most of the projects in this book. All of the Lantronix modules use virtually the same firmware interface settings.

Before connecting the Lantronix module to a microcontroller, you should configure it and confirm that the module is communicating properly through its serial port. The easiest way to do this is to connect it to a personal computer serially. You can use the FT232RL USB-to-serial module, or the MAX3323 RS-232-to-TTL serial circuit from Chapter 2. Figure 4-3 shows these circuits connected to the Micro module. The Micro module is shown with its own power supply, because it draws more current than the USB connection can supply.

To connect the Micro to the breadboard, you have two choices: you can make a ribbon cable, or you can make a connector out of header pins. You can use a ribbon cable with a connector to match the Micro's two rows of pins. This type of connector is called an IDC connector. In a pinch, you can pull a two-row ribbon connector out of an old PC, chop off one end, solder headers onto the other end, and use that. You can also buy IDC connectors and IDC DIP sockets, as listed earlier. Assembling these connectors is tricky without an IDC crimp tool, however. As an alternative, you can solder two rows of straight female header sockets to two rows of right-angle male header pins. Figure 4-4 shows how to align the header pins and sockets for soldering.





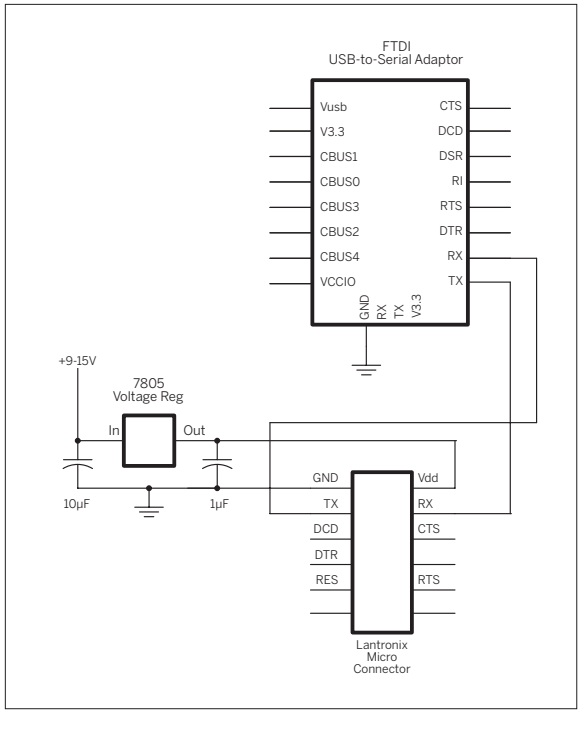
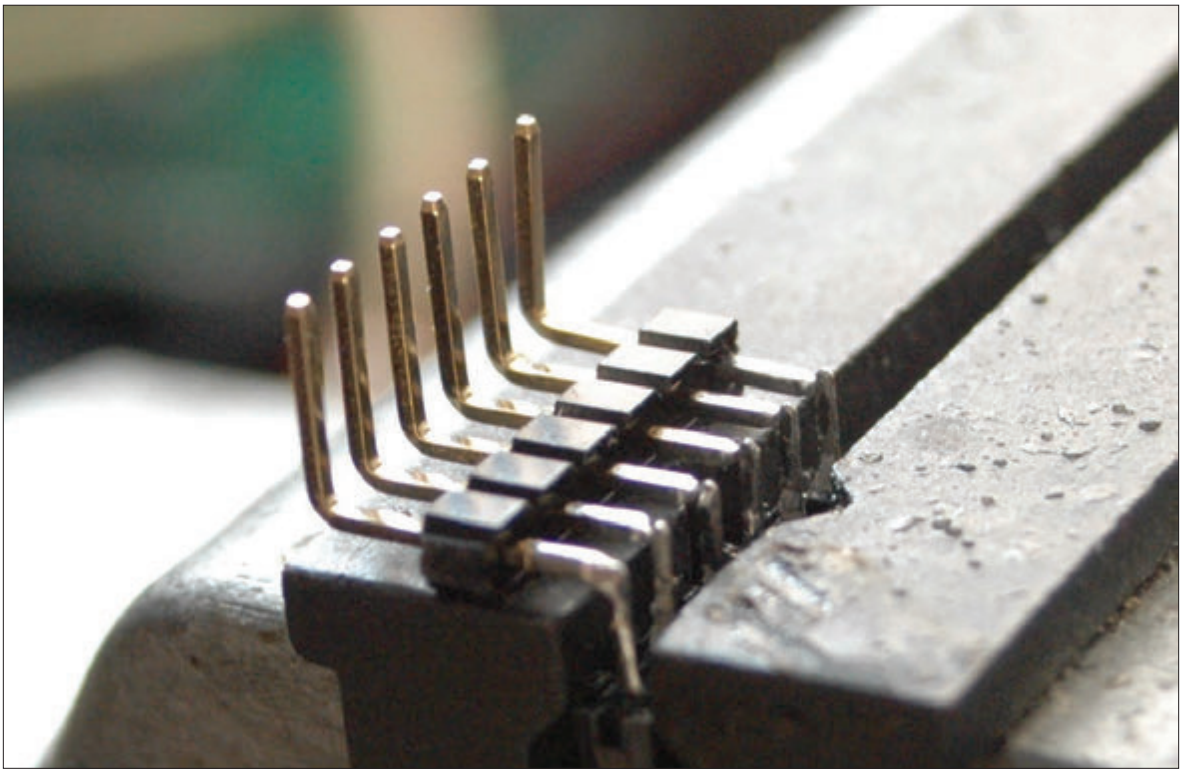


Figure 4-3
The Lantronix Micro connected to an FTDI USB-to-serial module. The circuit is shown with and without the micro attached, to show the wiring underneath. This same circuit can be used for the Micro or WiMicro modules. Note that the FTDI chip doesn't connect to +5V. It takes its power from the USB connection. Only a common ground is needed.



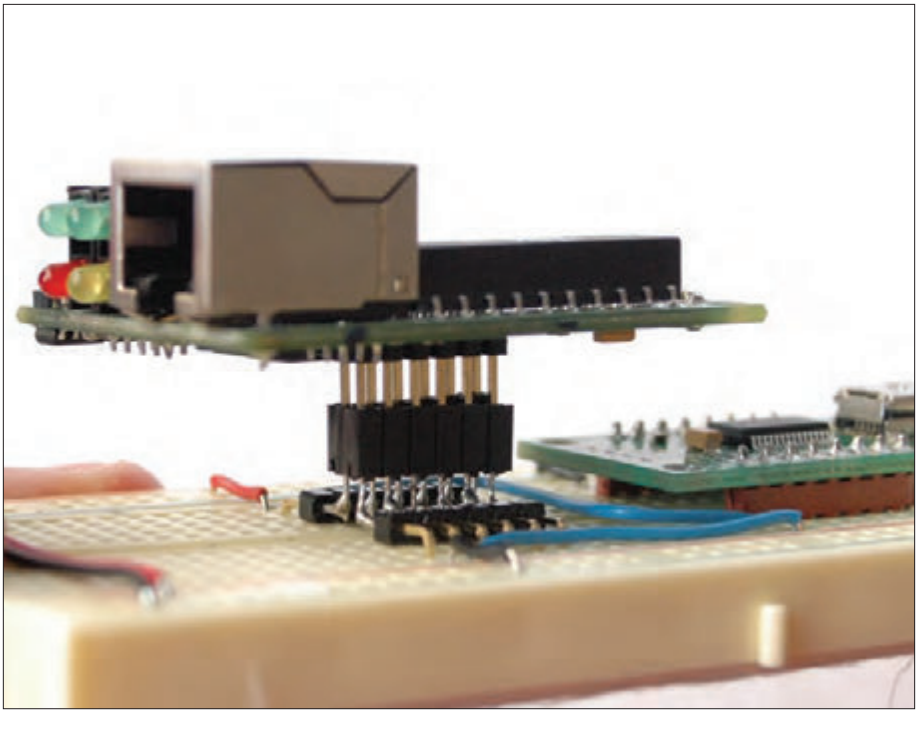


Figure 4-4
Soldering straight female headers to right-angle male headers. Slip the female sockets onto your module. The male header pins should now slip into a breadboard nicely, as shown in the second image, where the finished header assembly is used to mount the Micro module on a breadboard.

Configuring the Micro

Once you've got the circuit assembled, connect it to your PC, connect it to power, and open the serial port using your serial terminal program: GNU screen in Terminal on Mac OS X or Linux/Unix, PuTTY on Windows (see Chapter 2 for instructions on finding your serial ports). Press the reset button of the Micro module, hold down the x key, and release the reset button. This step forces the Lantronix module to go into setup mode. The module should return a configuration menu, like the following:

```
*** Lantronix Universal Device Server ***
Serial Number 6643485  MAC address 00204A66B9DD
Software version 05.2 (030423) LTX
Press Enter to go into Setup Mode
```

» Press Enter, and you'll get the menu shown at the right:

This menu allows you to configure all the settings of the module. The first menu choice controls the network settings. The second controls the serial settings. The others configure advanced settings, which you can leave at the default. Start by configuring the network settings. Type `0`, then press Enter. Then respond as shown below, entering an available IP address on your network.

NOTE: The Lantronix module can find its own address using DHCP, but if you don't know its address, you have no way of contacting it from other devices on the same network. That's why you're entering a fixed address for the module. Pick an address that you know is available on your network. If you're using it on a home network, or any private network, use an address appropriate to the local network.

```
Web Server is        enabled
ECHO is              disabled
Enhanced Password is disabled

*** Channel 1
Baudrate 9600, I/F Mode 4C, Flow 00
Port 10001
Remote IP Adr: --- none ---, Port 00000
Connect Mode : D4
Disconn Mode : 00
Flush   Mode : 00

*** Expert
TCP Keepalive    : 45s
ARP cache timeout: 600s

Change Setup:
  0 Server configuration
  1 Channel 1 configuration
  5 Expert settings
  6 Security
  7 Factory defaults
  8 Exit without save
  9 Save and exit           Your choice ?
```

» This is the address that the module will have on the network.

NOTE: Sample responses are shown in blue here. Change the numbers as needed for your network.

```
IP Address : (0) 192.(0) 168.(0) 1.(0) 20
```

» Next you'll set the address of the router through which your module contacts the rest of the Net. Fill in the router's IP address:

```
Set Gateway IP Address (Y) Y
Gateway IP addr (0) 192.(0) 168.(0) 1.(0)1
```

When a router assigns addresses to devices connected to it, it masks part of the address space so that those devices can use only addresses in the same subnet as the router itself. For example, if the router is going to assign only addresses in the range 192.168.1.2 through 192.168.1.254, it masks out the top three numbers (octets). This is called the netmask, or subnet mask. In your PC's network settings, you'll see it written as a full network address, like so: 255.255.255.0. In the Lantronix modules, tell the module the number of bits for the netmask.

» To set a mask equivalent to 255.255.255.0, use the value 8:

```
Netmask: Number of Bits for Host Part (0=default) (0)8
```

For now, don't worry about the telnet config password. You can set it later if you want to.

```
Change telnet config password (N) N
```

» Once you've entered the previous settings, choose option 1 from the menu. This configures the serial settings, as follows. You'll use the default 9600 bits per second, for now:

```
Baudrate (9600) ?
```

» The I/F (interface) mode sets the additional parameters of the serial communication: 8 bits per byte, no parity, one stop bit. These settings are covered by the default value, 4C:

```
I/F Mode (4C) ?
```

» You're not going to use hardware flow control, so the default, 00, turns it off:

```
Flow (00) ?
```

» Following is the number the network port number that will be connected to the device's serial port. Unless you need a specific port, use the default port 10001:

```
Port No (10001) ?
```

» Next comes the ConnectMode. This parameter controls how the module reacts when it gets network connections. Setting D4 tells it to allow all incoming connections, and when it gets an incoming connection, it should send a response out the serial port. You'll see this in action shortly.

```
ConnectMode (C0) ?D4
```

» With every new outgoing connection, the Lantronix modules can be configured to choose a new outgoing port. You don't need this, so when your module asks, reply with a no:

```
Auto-increment source port (N) ? N
```

» The Lantronix modules can be configured to connect automatically to a remote address and a remote port number on startup. In this chapter, you won't use this feature, so enter the default values (0 or 000) for all of these:

```
Remote IP Address : (000) .(000) .(000) .(000)
Remote Port  (0) ?
```

» The Disconnect mode determines how the module handles disconnection from remote addresses. The default value, 00, will meet your needs:

```
DisConnMode (00) ?
```

» The FlushMode controls how the module flushes its serial port buffers. The default value, 00, tells it to clear both transmit and receive buffers whenever it disconnects from a remote address:

```
FlushMode   (00) ?
```

» The Disconnect Time sets how long a remote device can be connected to the module with no activity. 00:00 sets the time to infinite:

```
DisConnTime (00:00) ?:
```

» The SendChar settings allow you to set a string of characters that can be sent to the module either from a remote device or from the local serial port to force it to disconnect. For now, you won't need this feature:

```
SendChar 1  (00) ?
SendChar 2  (00) ?
```

» Finally, choose option 9 to save your settings and exit setup.

NOTE: For full details on all the settings of the Lantronix modules, check out each module's user guide, found online at www.lantronix.com/support/documentation.html. I'll cover only the settings needed for the projects detailed here. Once you're comfortable using the modules, it's worth exploring some of the other features.

```
Change Setup:
  0 Server configuration
  1 Channel 1 configuration
  5 Expert settings
  6 Security
  7 Factory defaults
  8 Exit without save
  9 Save and exit           Your choice ? 9

Parameters stored ...
```

Now you're ready to connect to the module from the network. Connect the module to your network with an Ethernet cable. Leaving your serial terminal connection open, reset the module by pressing its reset button. After a few seconds, it should return the letter D in the serial window, indicating that it's disconnected from any other device.

Now open your terminal program (xterm on Linux/Unix or Terminal on Mac OS X) and attempt to make a telnet connection to the module's IP address on port number 10001. On Windows, you can run the telnet program from the command prompt or use PuTTY to connect.

```
telnet 192.168.1.20 10001
```

You'll get a response like this:

```
Trying 192.168.1.20...
Connected to 192.168.1.20.
Escape character is '^]'.
```

Back in your serial terminal program, the module will return C for "connected", I for "incoming connection," and the address of the machine that connected to it:

```
CI192.168.1.45
```

Now type messages back and forth. Whatever you type in the telnet session should show up in the serial window, and vice versa. Hello Internet! You've made your first connection. To disconnect, close the telnet session. The Micro module should respond with a D for "disconnected."

Once a Lantronix module is on the network, you can also configure it via telnet. Telnet to the device's IP address on port 9999 and you'll get the same configuration menu that you got in the serial terminal when you reset the module and held down the x button. You can also configure it via a web browser. Open a browser and enter the module's IP address. You'll get a Java-based configuration screen. Sometimes it can be useful to reconfigure your module via the Web or telnet. For initial configuration, however, it's easiest to configure via the serial port, because you don't need to know whether it's successfully obtained an IP address in order to do so.

Connecting Through the Network Module

Now that you've confirmed that your Micro works and that you can connect to it, it's time to make connections to the rest of the Net from it. With the serial terminal connection open, press the reset button on the module. It should respond with a D for "disconnected." To force a Lantronix device to connect to a remote address, type `C` followed by the remote address (numeric only), followed by a slash (`/`) followed by the port number. For example, to connect to O'Reilly's web server, you'd type:

```
C208.201.239.37/80
```

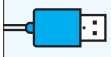
Once you're connected, you'd request a page just like you did from the command line in Chapter 3. Type:

```
GET /index.html HTTP/1.0
HOST: www.oreillynet.com
```

Then hit the Return key twice.

You won't see what you type, because the Lantronix modules don't echo your characters back to you, but you will see everything the server sends in response. In this case, you'll get the header and full HTML text of O'Reilly's web page, something like this:

```
HTTP/1.1 302 Found
Date: Thu, 15 Mar 2007 21:51:02 GMT
Server: Apache
Location: http://www.oreillynet.com/index.csp
Content-Length: 287
Connection: close
Content-Type: text/html; charset=iso-8859-1
```



Modem Responses

The module's responses parallel those of the Bluetooth modem in Chapter 2. When in command mode, it sends messages reporting on its status: the Micro module's D response corresponds to the NO CARRIER response from the Bluetooth module's AT command set. The CIaddress response corresponds to the Bluetooth modem's CONNECT,address response, and so forth. In fact, it's possible to make the Lantronix modules operate using an AT-style command set by changing the ConnectMode to D6. For the programs in this book, however, it's easier to use the less-verbose protocol afforded by setting the ConnectMode to D4.

Finding a Host's IP Address

The Lantronix modules' connect **command takes only numerical IP addresses, so you can't give them host names. For example, typing** Cwww.google.com/80 **won't work. If you need to find a host's IP address, use the** ping **command mentioned in Chapter 3. For example, if you open a command prompt and type** ping –c 1 www.oreillynet.com, **you will get the following response:**

```
PING www.oreillynet.com (208.201.239.37): 56 data bytes
64 bytes from 208.201.239.37: icmp_seq=0 ttl=45 time=97.832 ms
```

And there, in the first line, is the numerical IP address you need. Remember, ping **is your friend.**

... and so forth. The web server will close the connection when it's sent you the whole page.

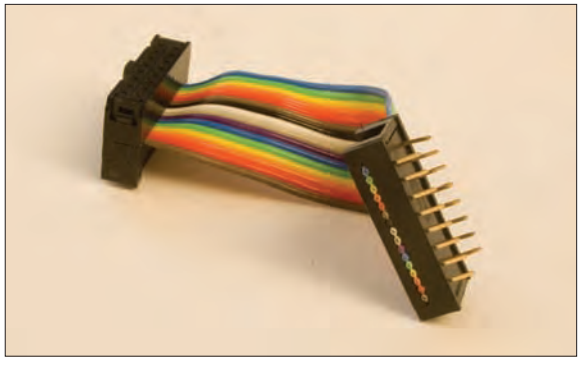
You've made two connections now. In the first, the Lantronix module acted as a server. You telnetted into it and saw what you typed come out the serial port. In the second, the module acted as a client, connecting to a remote host, making an HTTP request, and delivering the results through the serial port. Now it's time to write a program to make HTTP requests directly from a microcontroller.

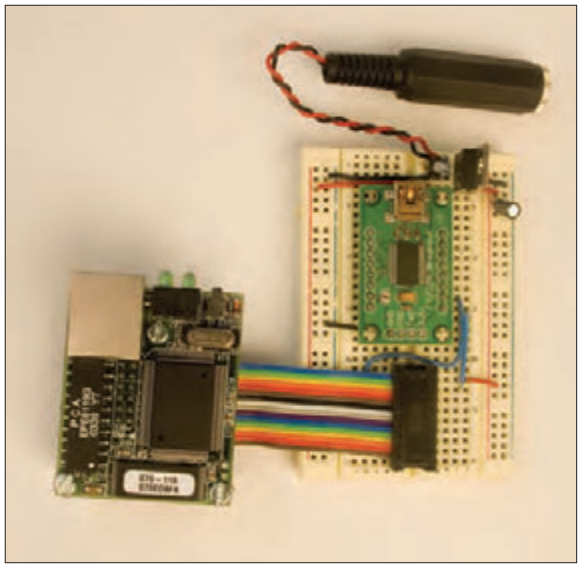
X

If this doesn't work, your serial program might not be sending a carriage return followed by a line feed at the end of each line. You can either reconfigure your serial program to do this, or type Control-M (shown as ^M next) followed by Control-J (^J) instead of pressing Return (once at the end of each line, and twice at the end of the last line):

```
C208.201.239.37/80<Return>
GET /index.html HTTP/1.0^M^J
HOST: www.oreillynet.com^M^J^M^^J
```

You will probably also find this easier to work with if you copy the contents of each line into a text file so that you can cut and paste them (without the end-of-line characters, because you're typing them yourself) into the serial program.





▼ A DIY ribbon cable adaptor for the Micro. Shown below that is the Micro-to-USV-adaptor circuit using an adaptor made from an IDC DIP plug and an IDC socket. The circuit is the same as in Figure 4-3.

An Embedded Network Client Application

By now, you should be pretty good at making connections through your module. It's time to build a full application. This project is an embedded web scraper. It takes data from an existing website and uses it to affect a physical output. It's conceptually similar to devices made by Ambient Devices, Nabaztag, and others.

Project 6

Networked Air Quality Meter

In this project, you'll make a networked air quality meter. You'll need an analog panel meter, like the kind you find in speedometers and audio VU meters. I got mine at a yard sale, but you can often find them in electronics surplus stores or junk shops. The model recommended in the parts list here is less picturesque than mine, but it will do for a placeholder until you find one you love.

On the following page, Figure 4-5 shows how it works: the microcontroller makes a network connection to a PHP script through the Lantronix module. The PHP script connects to another web page, reads a number from that page, and sends the number back to the microcontroller. The microcontroller uses that number to set the level of the meter. The web page in question is AIRNow, www.airnow.gov, the U.S. Environmental Protection Agency's site for reporting air quality. It reports hourly air quality status for many U.S. cities, listed by ZIP code. When you're done, you'll have a meter you can set anywhere in your home or office to see the state of the air quality in your city at a glance (assuming you live in the U.S.).

MATERIALS

- **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).
- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601.
- **1 USB-to-serial circuit** such as the FT232RL, or RS-232-to-serial circuit such as the MAX3323, as shown in Chapter 2. Use the second of these if you don't have a USB port and have to use an RS-232 serial port. Otherwise use the first.
- **The micro-to-breadboard connector** from the previous project (or make another one if needed).
- **Arduino module** or other microcontroller.
- **1 voltmeter** Get a nice-looking antique one if you can. For a placeholder, you can use part number 48J6151 from Newark (www.newarkinone.com). Ideally, you want a meter that reads a range from 0–5V, or 0–10V at most.
- **5 LEDs**

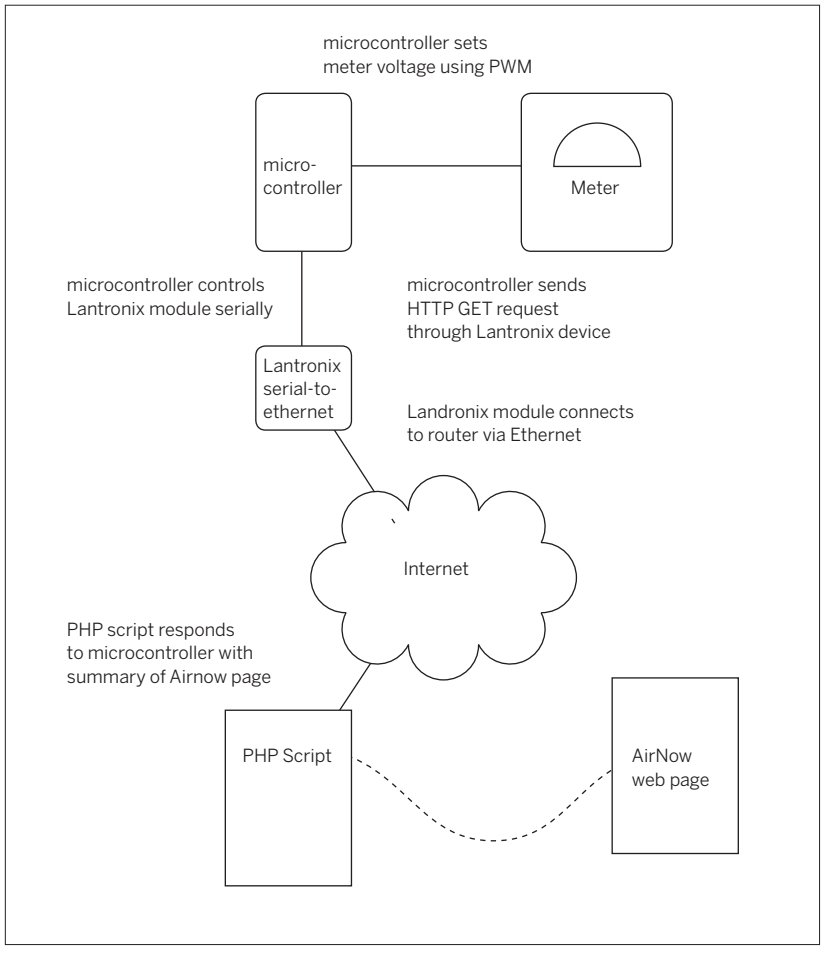


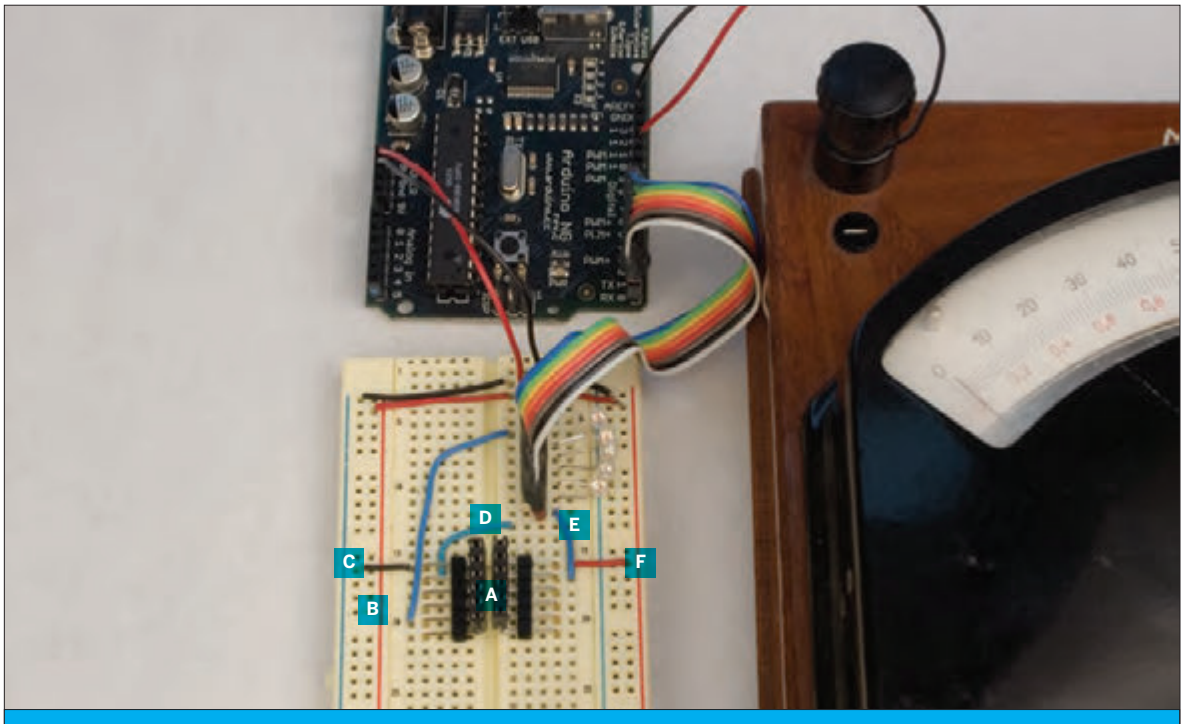
Figure 4-5
The networked air quality meter.

Control the Meter Using the Microcontroller

First, you need to generate a changing voltage from the microcontroller to control the meter. Microcontrollers can't output analog voltages, but they can generate a series of very rapid on-and-off pulses that can be filtered to give an average voltage. The higher the ratio of on-time to off-time in each pulse, the higher the average voltage. This technique is called pulse width modulation (PWM). In order for a PWM signal to appear as an analog voltage, the circuit receiving the pulses has to react much more slowly than the rate of the pulses. For example, if you pulse width modulate an LED, it will seem to be dimming, because your eye can't detect the on-off transitions when they come faster than about 30 times a second. Analog voltmeters are very slow to react to changing voltages, so PWM works well as a way to control these meters. By connecting the positive terminal of the meter to an output pin of the microcontroller and the negative pin to ground and pulse width modulating the output pin, you can easily control the position of the meter.

Figure 4-6 shows the whole circuit for the project. The Lantronix module is connected to the microcontroller's serial pins. You'll use it and the LEDs in the steps that follow.

If you've hooked this circuit up, you probably won't be able to program the microcontroller, because the Lantronix is using the same serial pins that are used by the USB interface. To program the board, you'll need to disconnect the RX pin (pin 0 on the Arduino).



A. Micro module connector B. Micro Reset C. GND D. Micro TX E. Micro RX F. +5V

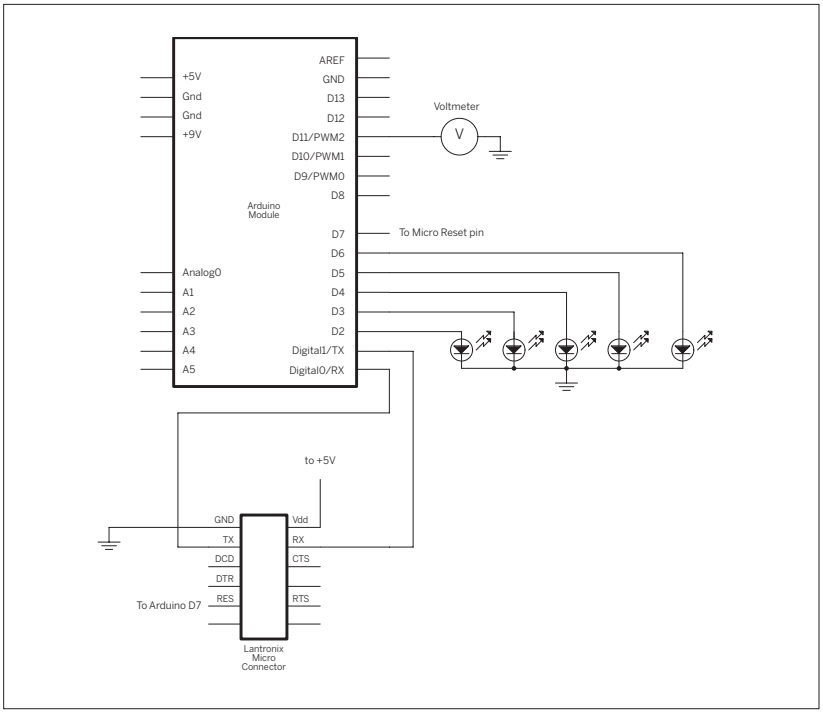


Figure 4-6
The circuit for a networked meter. The Micro itself has been removed to show the wiring beneath it. It plugs into the connector.

Test It

Here's a program to test whether you can control the meter:

You will need to adjust the range of pwmValue depending on the sensitivity of your meter. The meters used to design this project had different ranges. The meter from Newark in the parts list responds to a 0 to 5 volt range, so the preceding program moves it from its bottom to its top. The antique meter, on the other hand, responds to 0 to 3 volts, so it was necessary to limit the range of pwmValue to 0 – 165. When it was at 165, the meter reached its maximum. You'll see in the code that appears later how to limit the range using a maximum value.

```
/*
    Voltmeter Tester
    Uses analogWrite() to control a voltmeter.
    Language: Wiring/Arduino
*/
// the output pin that the meter is attached to:
#define meterPin 11

int pwmValue = 0;  // the value used to set the meter

void setup() {
  // nothing here
}

void loop() {
  // move the meter from lowest to highest values:
  for (pwmValue = 0; pwmValue < 255; pwmValue ++) {
    analogWrite(meterPin, pwmValue);
    delay(10);
  }
  delay(1000);
  // reset the meter to zero and pause:
  analogWrite(meterPin, 0);
  delay(1000);
}
```

Write a PHP Script to Read the Web Page

Next, you need to get the data from AIRNow's site in a form the microcontroller can read. The microcontroller can read in short strings serially, and converting those ASCII strings to a binary number is fairly simple. Parsing through all of the text of a web page using a microcontroller is difficult, but it's the kind of task that PHP was made for. The program that follows reads the AIRNow page, extracts the current AQI reading, and passes that value on to the microcontroller. The Lantronix module is the microcontroller's gateway to the Net, allowing it to open a TCP connection to your web host, where you need to install this PHP script.

NOTE: You could also run this script on one of the computers on your local network. As long as the microcontroller is connected to the same network, you'll be able to connect to it and request the PHP page. For information on installing PHP or finding a web hosting provider that supports PHP, see www.php.net/manual/en/tutorial.php#tutorial.requirements.

Figure 4-7 shows AIRNow's page for New York City (airnow.gov/index.cfm?action=airnow.showlocal&cityid=164). AIRNow's page is formatted well for extracting the data. The AQI index number is clearly shown in text, and if you remove all of the HTML tags, it appears on a line by itself, always following the line AQI observed at hh:mm AM/PM:.

NOTE: One of the most difficult things about maintaining applications like this, which scrape data from an existing website, is the probability that the designers of the website could change the format of their page. If that happens, your application could stop working, and you'll need to rewrite your code. This is a case where it's useful to have the PHP script do the scraping of the remote site. It's more convenient to rewrite the PHP than it is to reprogram the microcontroller once it's in place.

X

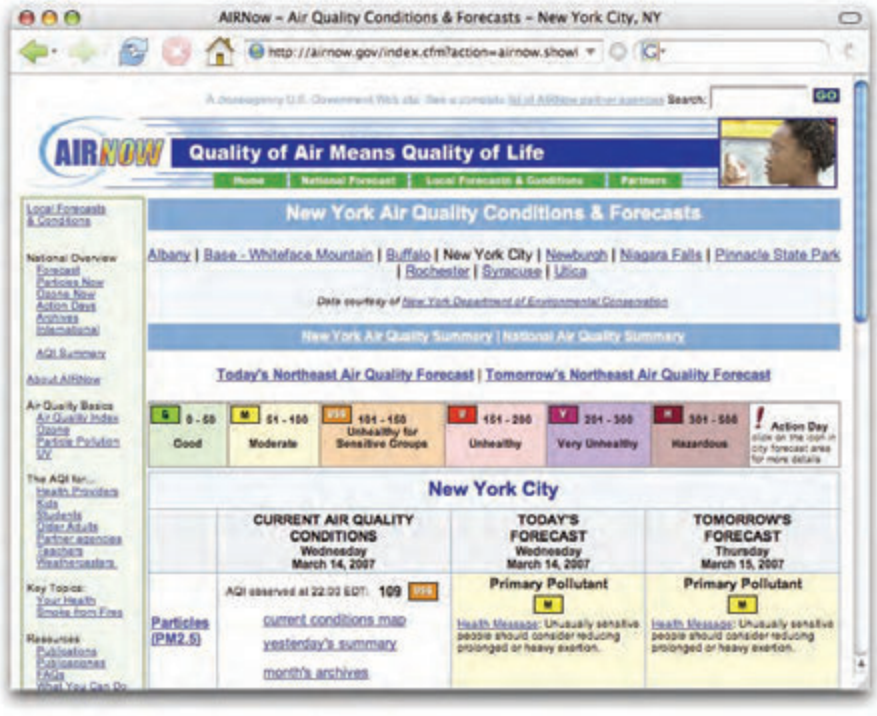


Figure 4-7
AIRNow's page is nicely laid out for scraping. The PHP program used in this project ignores the ozone level.

Fetch It

The PHP script shown here opens the URL of the AIRNow web page and, as long as there are more lines to read, it prints the latest line. The fgetss() command reads a line of text and removes any HTML tags.

When you save this file on your web server and open it in a browser, you should get the text of the AIRNow page without any HTML markup or images. It's not very readable in the browser window, but if you view the source code (click the View→Source menu item in your web browser), you'll see that the text is nicely separated into lines. Scroll down and you'll find some lines like this:

```
AQI observed at 12:00 EDT:
28
```

These are the only two lines you care about.

```
<?php
/*
    AIRNow Web Page Scraper
    Language: PHP
*/
    // Define variables:
    // url of the page with the air quality index data 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 $line;
    }
    // close the file at the URL, you're done:
    fclose($filePath);
?>
```

Scrape It

To extract the data you need from those lines, you'll need a couple more variables. Add these lines at the top of the program (but after the line starting with <?php):

```
$readParticles = 0;    // flag telling you the next line
                       // is the particle value
$particles = -1;       // the particles value
```

» Then replace the command echo $line; in the program with this block of code:

This block uses the preg_match() command to look for a string of text matching a pattern you give it. In this case, it looks for the pattern AQI observed at. You know that when you see that line, the next line is the number you want. When the PHP script finds the "observed at" line, it sets the variable $readParticles to 1.

```
// if the current line contains the substring "AQI observed at"
// then the line following it is either the particle reading
// or the ozone reading:
if (preg_match('/AQI observed at /', $line)) {
    // if $particles == -1, you haven't gotten
    // a value for it yet:
    if ($particles == -1) {
        $readParticles = 1;
    }
}
```

» Now, add the following block of code *before* the one you just added (just before the comment that begins with "if the current line"):

This code checks to see if $readParticles is equal to 1. If it does, it reads the current line of text, trims off any excess characters, and prints it out. The result in your web browser should look like this:

```
< AQI: 43>
```

Now you've got a short string of text that your microcontroller can read. The next step program your microcontroller to read your PHP script over the Net (be sure to reconnect the Lantronix after you program it). To see the PHP script in its entirety, see Appendix C.

```
// if the previous line was the "observed at line" preceding
// the particle matter reading, then $readParticles = 1 and
// you should get this line, trim everything but the number,
// and save the result in $particles:
if ($readParticles == 1) {
    $particles = trim($line);
    echo "< AQI: $particles>";
    $readParticles = 0;
}
```

Read the PHP Script Using the Microcontroller

The microcontroller can communicate through the Lantronix module, just like you did from the serial terminal window. First, you send a connect string telling it the numerical address of the server and the port number. When a connection is made, the Lantronix device returns a "C" to let you know it's connected. After that, you send an HTTP GET request for the PHP script. Then the script returns the Air Quality Index string.

Before you start programming, plan the sequence of messages. Using the Lantronix module as a network client is very similar to using Processing as a network client. In both cases, you have to know the correct sequence of messages to send and how the responses will be formatted. You have to write a program to manage the exchange of messages. Whether you're writing that program in Processing or whether you're writing it in Arduino or in another language on another microcontroller, the steps are still the same:

1. Open a connection to the web server
2. Send an HTTP GET request
3. Wait for a response
4. Process the response
5. Wait an appropriate interval and do it all again

Each of these steps involves sending a message and then waiting for a response, so it's worthwhile to keep track of the state that the program is in. Figure 4-8 is a flowchart of what happens in the microcontroller program. The states of the program — disconnected, connecting, connected, requesting, reading, and complete — are laid out on the left of the chart. The actions taken to move from one state to the next follow from the states. You can see that in each state, there's a loop where you take action, then wait. Based on the outcome of the action, you either keep looping, or go to the next state. Laying out the whole program in a flowchart like this will help you keep track of what's going on at any given point.

Because you need to use the serial port to send and receive messages to and from the server, you can't use Serial.print() statements to check what's going on in the program. Instead, you can use LEDs to keep track of the part of the program that you're in. LEDs attached to I/O pins will indicate which state your program is in.

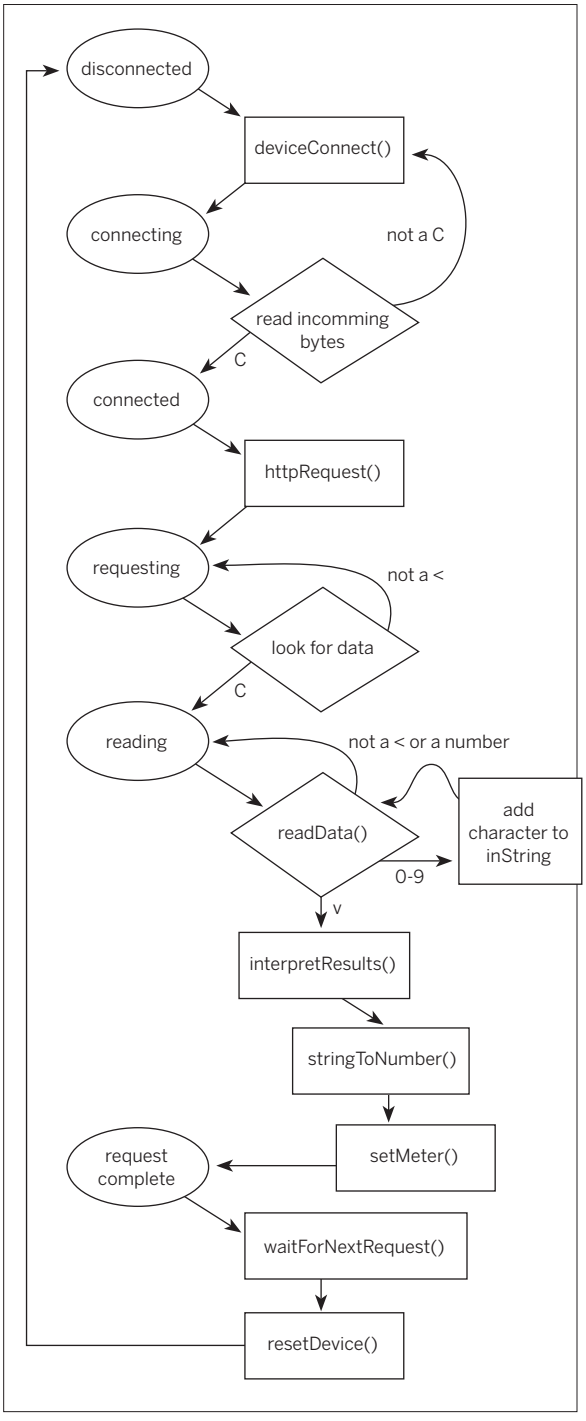


Figure 4-8
A flowchart of the Arduino program for making and processing an HTTP GET request.

Connect It

The program starts out by defining each of the states as numerical values, so that you can refer to them later on in a case statement. Then it defines the pin numbers for the various inputs and outputs:

```
// 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; has to be one of
                              // the PWM pins (9 – 11)
```

» A couple of constants are defined for converting the AQI reading to a meter level:

```
// defines for voltmeter:
#define meterMax 130          // max value on the meter
#define meterScale 150        // my meter reads 0 - 150
```

» Next, the global variables are declared and initialized:

```
// 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
```

» In the setup() method, set the state of all the I/O pins, initialize the serial port, call a custom method to reset the Lantronix device, then blink the program reset LED to signal that the main loop is about to begin:

```
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);
}
```

You can see from the schematic back in Figure 4-6 that digital pin 7 is connected to the Lantronix module's reset pin. Here's the routine that resets the module:

```
// 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);
}
```

The blink() method called in the setup is a method to blink the LED on pin 6, so you know the microcontroller's main loop is about to begin:

```
// 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);
  }
}
```

The loop() just calls two routines, one to check the state of the program and take appropriate action, and another to set the LEDs depending on the state of the program. Here it is:

```
void loop() {
  stateCheck();
  setLEDs();
}
```

The stateCheck() method is a switch-case statement that checks the value of the status variable. In each case, it takes whatever action is appropriate to its current value:

```
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();
```

Continued from opposite page.

```
    break;
  case requesting:
    lookForData();
    break;
  case reading:
    readData();
    break;
  case requestComplete:
    waitForNextRequest();
  }
}
```

In the disconnected state, the deviceConnect() method sends the connect string to the Lantronix module, like so:

```
void deviceConnect() {
  //   send out the server address and
  //   wait for a "C" byte to come back.
  //   fill in your server's numerical address below:
  Serial.print("C82.185.179.43/80\n");
  status = connecting;
}
```

Fill in the numerical IP address of your web server here.

In the connecting state (back in the stateCheck() method, which you've already seen), you've already sent a connect string, and you have to wait for the module to connect and return a "C". If you get anything else, you have to try again:

```
    // 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;
```

Once you're connected, you send an HTTP GET request. Here's the httpRequest() method:

```
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 and 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;
}
```

The server replies to httpRequest() like so:

```
HTTP/1.1 200 OK
Date: Fri, 14 Apr 2006 21:31:37 GMT
Server: Apache/2.0.52 (Red Hat)
Content-Length: 10
Connection: close
Content-Type: text/html; charset=UTF-8

< AQI: 65>
```

The stuff you see at the top is the HTTP header. When you call the PHP script from a browser, you don't see all of the header, because the browser strips it out for you. The lookForData() method reads strip the header out by ignoring all the bytes before the < sign:

```
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;
    }
  }
}
```

After you get the < sign, the readData() method takes only the numeric characters from the remaining string and saves them to the inString[] array. When it gets the > symbol indicating the end of the string, it calls the interpretResults() method:

```
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();
    }
  }
}
```

The interpretResults() method converts the string to a number and sets the meter. It also takes note of the time when the meter was last successfully set, so you can count one minute before the next request:

```
void interpretResults() {
  // convert the string to a numeric value:
  int airQuality = atoi(inString);
  // set the meter appropriately:
  setMeter(airQuality);
  lastCompletionTime = millis();
  status = requestComplete;
}
```

The setMeter() method takes the number and scales it to set the meter appropriately. You will need to adjust the formula if your meter is different from the ones shown here.

You will need to adjust meterMax and meterScale to values that work for your meter. You can determine these by using the meter testing program in the last section.

```
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);
  }
}
```

When the request is complete, waitForNextRequest() counts time for two minutes, then sets the status to disconnected to initiate a new request:

You can see that you're waiting two full minutes in between successful reads of the web page. In fact, you could wait even longer, as the page is updated only about once an hour. There's no need to check constantly. It creates undue demand on the server, and wastes energy. Remember, one of the cardinal rules of love and networking is to listen more than you speak.

```
void waitForNextRequest() {
  if (millis() - lastCompletionTime >= 120000) {
    // reset Lantronix device before next request:
    resetDevice();
    status = disconnected;
  }
}
```

The last thing you need to do in the main loop is to set the indicator LEDs so that you know where you are in the program. Because you gave each of the status settings a numeric value up at the top of the program, you can use those numbers to set the LEDs. The setLEDs() methods does this:

That's the whole program. Once you've got this program working on the microcontroller, the controller will make the HTTP GET request once a minute, and set the meter accordingly.

```
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);
    }
  }
}
```

The Finished Project

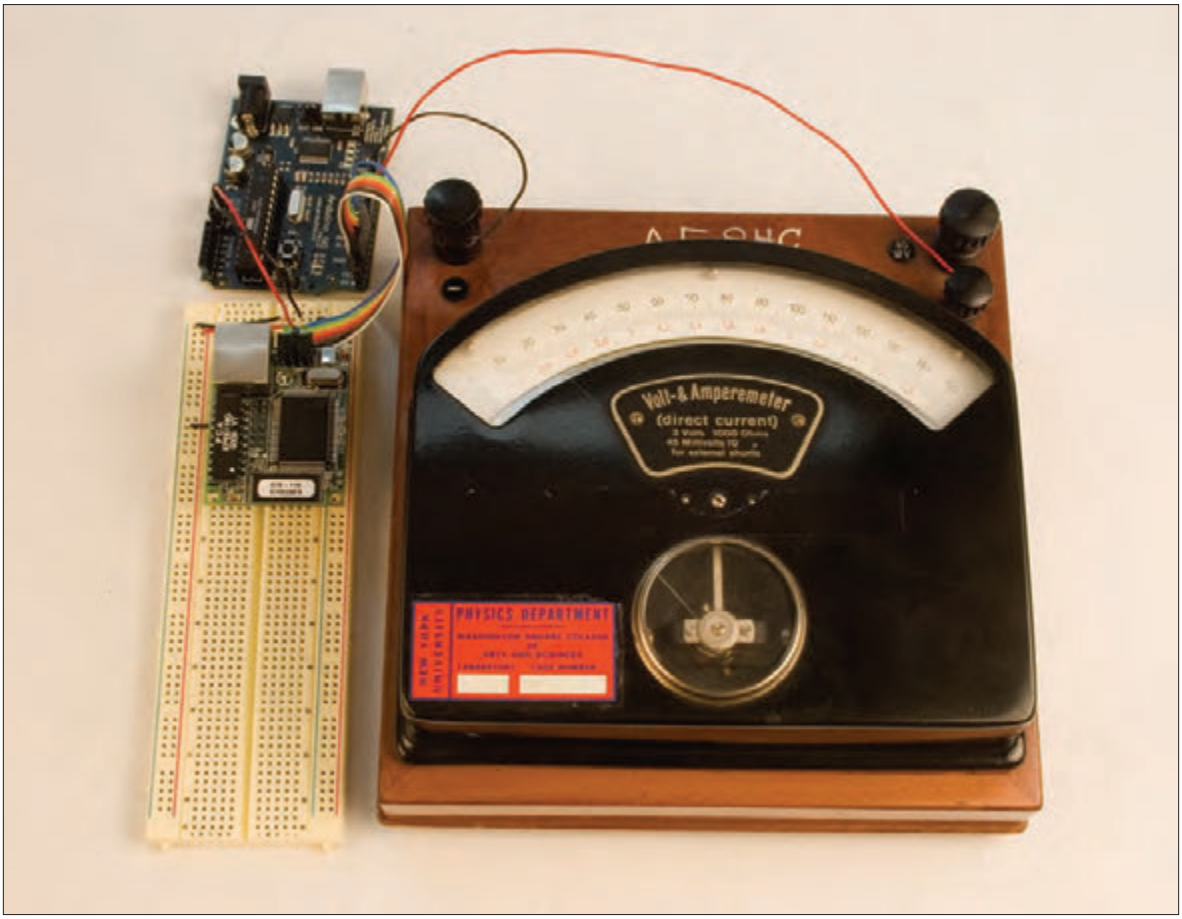


Figure 4-9
The completed networked air quality meter.

Before you run the final program for this project, you must change at least three lines of code (emphasized in the code listing), which are in various places in the example:

```
Serial.print("C82.165.199.35/80\n");
Serial.print("GET /netobj/code/php/scraper.php HTTP/1.1\n");
Serial.print("HOST:example.com\n\n");
```

You may also need to change meterScale **and** meterMax**, depending on the sensitivity of your meter.**

Serial-to-Ethernet Modules: Programming and Troubleshooting Tools

You probably hit a number of problems in making the connections in the last section. Perhaps you wired the transmit and receive connections backwards, or perhaps you got the IP configuration wrong. Probably the most challenging thing about troubleshooting your problems was that there was no clear indication from the Micro module that anything had happened at all. This is the norm when you're working with these modules, or with just about any embedded modem-style module that you build yourself. This section covers a few things you should always check, and a few tools that will help you solve problems. These principles apply whether you're using the Lantronix modules or some other embedded network module.

The Three Most Common Mistakes

Power and Ground

Always check whether you have made the power and ground connections correctly. If you're lucky, the module you're using will have indicator LEDs that light up when it's working properly. Whether it does or not, check the voltage between power and ground with a meter to make sure you've got it powered correctly.

Transmit and Receive

Confirm that you've got the module's transmit pin wired to the receive pin of your serial port or microcontroller, and vice versa. If you don't, you'll get nothing either way.

Configuration

If you're sure about the hardware connections, check the device's configuration to make sure it's all correct. In the case of the Lantronix modules, is the IP address you entered one that's on your subnet? Is the router address correct? Is the netmask?

The Lantronix devices, like many modems, can be configured from either end. So far you've seen how to configure them through the serial port, by holding down the x key in the serial terminal while you reset the module. Remember, you can also telnet into port 9999 to change the configuration settings. If you have problems with the serial connection, try checking the configuration via the network connection.

Diagnostic Tools and Methods

Once you know the modem's working, you have to program the sequence of messages that constitutes your application. Depending on the application's needs, this sequence can get complex, so it's useful to have a few simple programs around to make sure things work the way you want them to.

Use a Second Serial Output for Debugging Messages

One of the most difficult aspects of debugging a microcontroller speaking to a Lantronix device is that you can't send serial debugging messages to the Arduino or Wiring Serial Monitor over the serial port, because it's attached to the Lantronix device. Any serial messages you send will interfere with the communication with the server! This is a common problem in networked microcontroller projects. You can use the Wiring and Arduino SoftwareSerial library to send serial debugging messages on another set of pins. SoftwareSerial can send and receive data only at speeds up to 9600 baud, but it's useful for debugging.

NOTE: On a Wiring module, you can use the second serial port instead of using SoftwareSerial.

To use SoftwareSerial, you'll need two spare digital I/O pins and either a USB-to-serial module or an RS-232-to-TTL module to which you can connect them. In the previous project you never used pins 8 and 9, so this example uses those pins. Figure 4-10 shows a modified version of the meter schematic with the FT232RL USB-to-serial module added to pins 8 and 9 for use with SoftwareSerial.

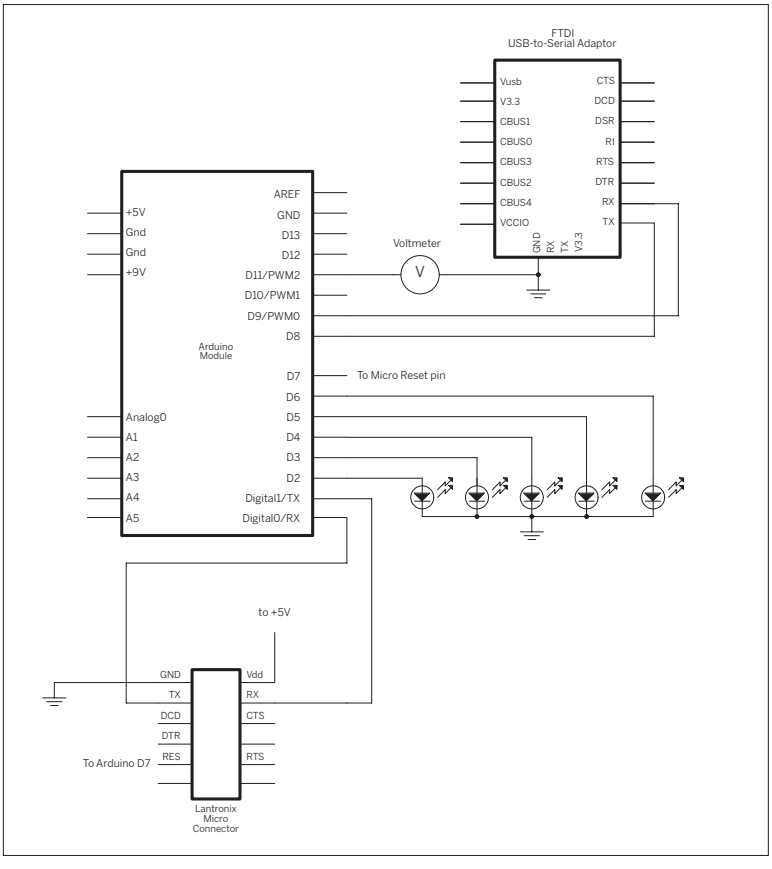
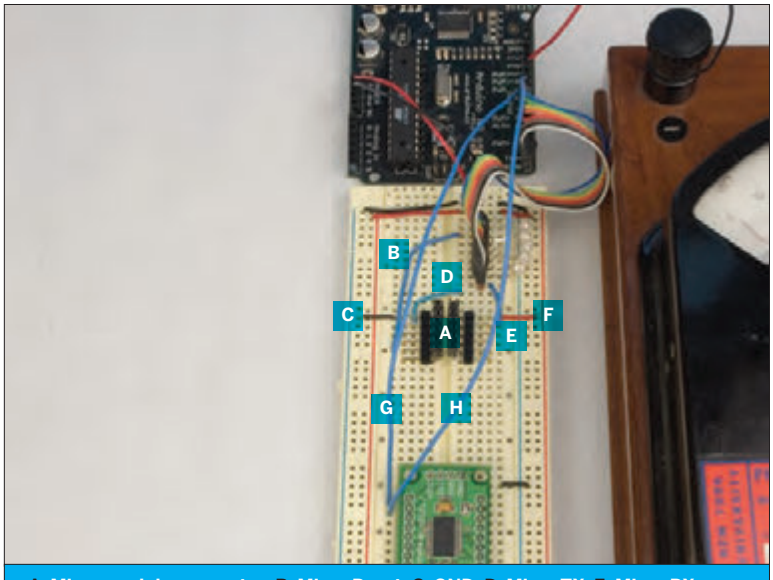


Figure 4-10
Modified network meter circuit, with serial adaptor added.



A. Micro module connector B. Micro Reset C. GND D. Micro TX E. Micro RX F. +5V G. SWserial TX H. SWserial RX

Debug It Wherever you want have a debugging message printed out, use SoftwareSerial instead of Serial. Open your serial terminal program (GNU screen or PuTTY) to see the messages. Here's a simple program that shows how to use the SoftwareSerial library:

```
/*
  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);
}
```

» Here's an abbreviated version of the meter program, showing where you might insert SoftwareSerial debugging messages. Most of the code has been removed here, to show you debugging messages at the beginning or end of each function. New code is shown in blue:

```
// 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:
```

Continued from previous page.

```
  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

  // print out a debugging message:
  mySerial.println("wait");
}


void setMeter(int desiredValue) {
  // print out a debugging message:
  mySerial.println("set");

  // the rest of setMeter() code goes here
}


void resetDevice() {
  // print out a debugging message:
  mySerial.println("reset");

  // 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);
  }
}
```

Write a Test Client Program in Processing

It's easiest to work through the steps of the program if you can step through the sequence of events. More expensive development environments allow you to step through a program one line at a time, but you can make your own version of a step-by-step program in Processing (you'll need to hook the Lantronix device to your computer as shown back in Figure 4-3). The following code goes through each of the steps needed to command a Lantronix module to connect to a remote server. Every time you press any key on the keyboard, it takes the next step. The serialEvent() method waits for data to be returned from each step and prints it out, so you can decide when to take the next step. It's an excellent tool for diagnosing whether you're getting the responses you want from the remote server.

Test It The handy thing about this program is that you can test the exchange of messages without having a microcontroller. Once you know you have the sequence right, you can translate it into code for the Arduino module:

```
/*
 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 modules
 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 apprropriate 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());
}
```

Continued from previous page.

```
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;
  }
}
```

» You'll need to replace this address and path with the address and path of the server you want to contact.

Write a Test Server Program

The previous program allowed you to connect to a remote server and test the exchange of messages. The remote server was beyond your control, however, so you can't say for sure that the server ever received your messages. If you never made a connection, you have no way of knowing whether the module can connect to any server. To test this, you can write your own server program for it to connect to.

» Here is a short Processing program that you can run on your PC. It listens for incoming connections, and prints out any messages sent over those connections. It sends any keystrokes typed out over the open connection.

To use this, first make sure your Lantronix module and your PC are on the same network. Then run this program, and connect to it from the module by connecting to your PC's IP address, port 8080. For example, if your PC has the IP address 192.168.1.45, the connect string would be:

```
C192.168.1.45/8080
```

```
/*
 server_test
 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:
```

»

Continued from opposite page.

```
  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);
    }
  }
}
```

This program uses port 8080, which is a common alternative port for many web servers. If you're running a web server on your PC, you might have to change the port number in this program.

Whether you're connecting from a serial terminal program or have programmed a microcontroller to make the connection, you can use this program to test whether the Lantronix module is making a successful connection. Once you've seen messages coming through to this program in the right sequence, just change the connect string in your microcontroller code to the address of the web server you want to connect to, and everything should work fine. If it doesn't, the problem is most likely with your web server. Contact your service provider for details on how to access any server diagnostic tools they provide, especially any error logs for your server.

X

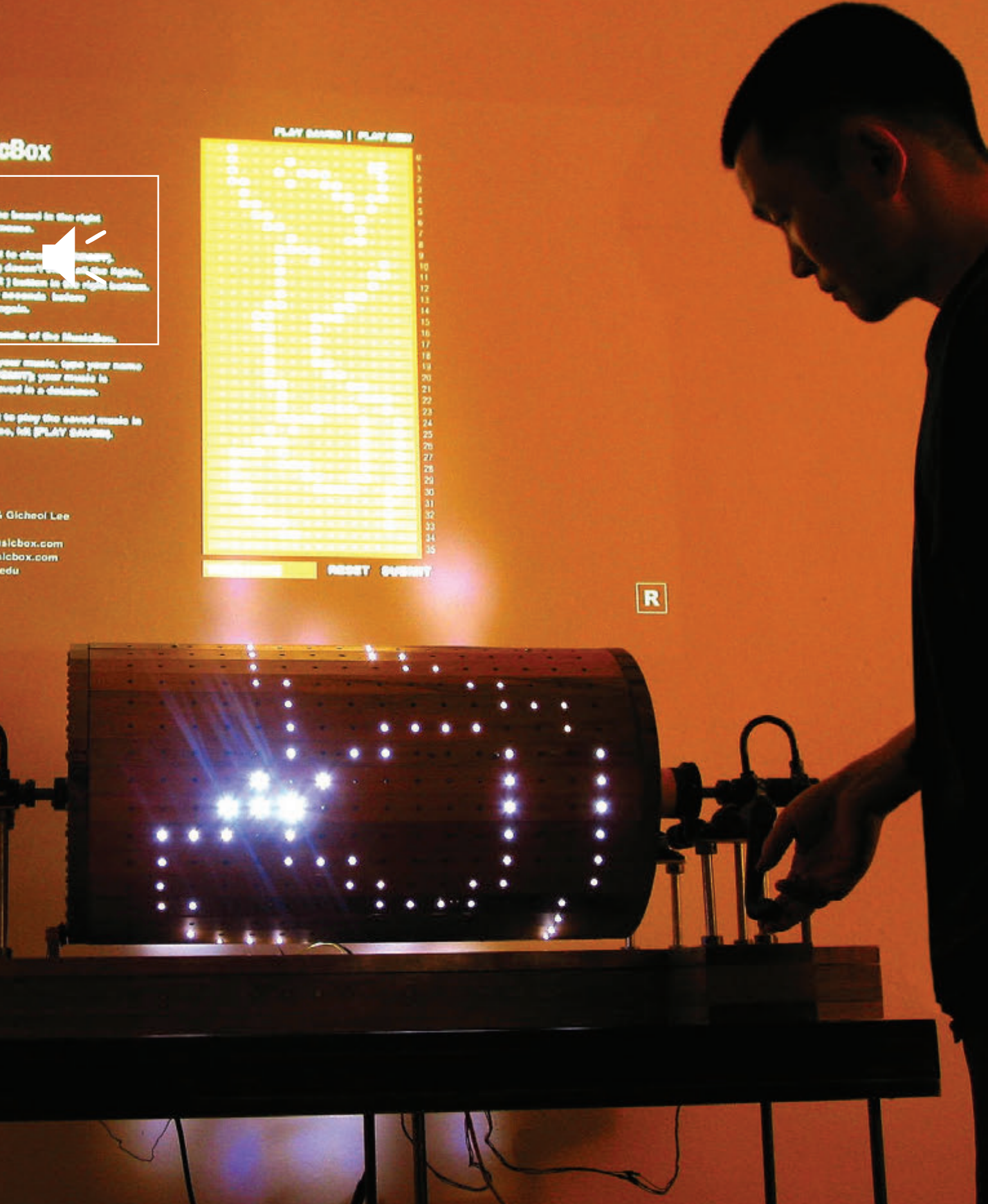
Conclusion

The activities in this chapter show a model for networked objects that's very flexible and useful. The object is basically a browser, requesting information from the Web and extracting the information it needs. You can use this model in many different projects.

The advantage of this model is that it doesn't require a lot of work to repurpose existing web applications. At most, you need to write a variation of the PHP web scraper from this chapter to summarize the relevant information from an existing website. This flexibility makes it easier for microcontroller enthusiasts who aren't experienced in web development to collaborate with web programmers, and vice versa. It also makes it easy to reuse others' work if you can't find a willing collaborator.

The model has its limits, though, and in the next chapter you'll see some ways to get around those limits with a different model. Even if you're not using this model, don't forget the troubleshooting tools mentioned here. Making simple mock-ups of the programs on either end of a transaction can make your life much easier, because they let you see what should happen, and modify what actually is happening to match that.

X


cBox
Gicheol Lee
sicbox.com
icbox.com
edu
PLAY SAVED | PLAY NEW
RESET
SUBMIT
R

5

MAKE: PROJECTS

Communicating in (Near) Real Time

So far, all of the networked communications you've seen worked like a Web browser. Your object made a request to a remote server, the server ran a program and then sent a response. This transaction worked by making a connection to the web server, exchanging some information, then breaking the connection. In this chapter, you'll learn more about that connection, and you'll write a server program that allows you to maintain the connection in order to facilitate a faster and more consistent exchange between the server and client.

Musicbox by Jin-Yo Mok (2004)

The music box is connected to a composition program over the Internet using a serial-to-ethernet module. The composition program changes the lights on the music box and the sounds it will play. Real time communication between the two in order to give the player feedback on what he is playing. *Photo courtesy of Jin-Yo Mok.*

Interactive Systems and Feedback Loops

In every interactive system, there's a feedback loop: you take action, the system responds, you see the response, or a notification of it, and you take another action. In some systems, the timing of that loop can be very loose. In other applications, the timing must be tight.

For example, in the cat bed application in Chapter 3, there's no need for the system to respond in more than a few seconds, because your reaction is not very time-sensitive. As long as you get to see the cat while he's on the bed (which may be true for several minutes or hours), you're happy. Monski pong in Chapter 2 relies on a reasonably tight feedback loop in order to be fun. If it took a half second or more for the paddles to move when you move Monski's arms, it'd be no fun. The timing of the feedback loop depends on the shortest time that matters to the participant.

Any system that requires coordination between action and reaction needs a tight feedback loop. Consider remote control systems, for example. Perhaps you're building a robot that's operated over a network. In that case, you'd need not only a fast network for the control system, but also a fast response from the camera or sensors on the robot (or in its environment) that are giving you information about what's happening. You need to be able to both control it quickly and see the results quickly. Networked action games also need a fast network. It's no fun if your game console reacts slowly, allowing other players with a faster network connection to get the jump on you. For applications like this, an exchange protocol that's constantly opening and closing connections (like HTTP does) wouldn't be very effective.

When there's a one-to-one connection between two objects, it's easy to establish a tight feedback loop. When there are multiple objects involved, though, it gets harder. To begin with, you have to consider how the network of connections between all the objects will be configured. Will it be a star network, with all the participants connected through a central server? Will it be a ring network? Will it be a many-to-many network, where every object has a direct connection to every other object? Each of these configurations has different effects on the feedback loop timing. In a star network, the objects on the edge of the network aren't very busy, but the central one is. In a ring network, every object shares the load more or less equally, but it can take a long time for a message to reach objects on opposite sides of the ring. In a direct many-to-many network, the load is equally distributed, but each object needs to maintain a lot of connections.

In most cases where you have a limited number of objects in conversation, it's easiest to manage the exchange using a central server. The most common program example of this is a text-based chat server like IRC (Internet Relay Chat), or AOL's instant messenger servers (AIM). Server programs that accept incoming clients and manage text messages between them in real time are often referred to as chat servers. The Processing program you'll write in this chapter is a variation on a chat server. The server will listen for new connections and exchange messages with all of the clients that connect to it. Because there's no guarantee how long messages take to pass through the Internet, the exchange of messages can't be instantaneous. But as long as you've got a fast network connection for both clients and server, the feedback loop will be faster than human reaction time.

X

Transmission Control Protocol: Sockets & Sessions

Each time a client connects to a web server, the connection that's opened uses a protocol called Transmission Control Protocol, or TCP. TCP is a protocol that specifies how objects on the Internet open, maintain, and close a connection that will involve multiple exchanges of messages. The connection made between any two objects using TCP is called a socket. A socket is like a pipe joining the two objects. It allows data to flow back and forth between them as long as the connection is maintained. Both sides need to keep the connection open in order for it to work.

For example, think about the exchanges between a web client and server that you saw in the last two chapters. The pipe is opened when the server acknowledges the client's contact, and remains open until the server has finished sending the data.

There's a lot going on behind the scenes of a socket connection. The exchange of data over a TCP connection can range in size anywhere from a few bytes to a few terabytes or more. All that data is sent in discrete packets, and the packets are sent by the best route from one end to the other.

NOTE: "Best" is a deliberately vague term: the optimal route is calculated differently by different network hardware, and involves a variety of metrics (such as the number of hops between two points as well as the available bandwidth and reliability of a given path).

The period between the opening of a socket and the successful close of the socket is called a session. During the session, the program that maintains the socket tracks the status of the connection (open or closed) and the port number; counts the number of packets sent and received; notes the order of the packets and sees to it that packets are presented in the right order, even if the later packets arrive first; and accounts for any missing packets by requesting that they be re-sent. All of that is taken care of for you when you use a TCP/IP stack like the Net library in Processing or the firmware on the Lantronix devices you first saw in Chapter 4.

The complexity of TCP is worthwhile when you're exchanging critical data. For example, in an email, every byte is a character in the message. If you drop a couple of bytes, you could lose crucial information. The error checking of TCP does slow things down a little, though, and if you want to send messages to multiple receivers, you have to open a separate socket connection to each one.

There's a simpler type of transmission protocol that's also common on the net called the User Datagram Protocol, or UDP. Where TCP communication is based on sockets and sessions, UDP is based only on the exchange of packets. You'll learn more about it in Chapter 7.

X

Project 7

A Networked Game

Networked games are a great way to learn about real time connections. This project is a networked variation on pong. In honor of everyone's favorite network status command, let's call it call it *ping* pong. The server will be a Processing program, and the clients will be physical interfaces that connect through Lantronix serial-to-Ethernet (or Wi-Fi) modules. The clients and the server's screen have to be physically near each other so that everyone can see the screen. In this case, you're using a network for its flexibility in handling multiple connections, not for its ability to connect remote places.

From the Monski pong project in Chapter 2, you're already familiar with the methods needed to move the paddles and the ball, so some of the code will be familiar to you. As this is a more complex variation, it's important to start with a good description of the whole system. The system will work like this:

- The game has two teams of multiple players.
- Each player can move a paddle back and forth. The paddles are at the top and bottom of the screen, and the ball moves from top to bottom.
- Players connect to the game server through a TCP connection. Every time a player connects, another paddle is added to the screen. New connections alternate between the top and bottom teams. When a player connects, the server replies with the following string: hi, followed by a carriage return and a line feed (shown as \r\n).
- The client can send the following commands:
  - l (ASCII value 108): move left
  - r (ASCII value 114): move right
  - x (ASCII value 120): disconnect
- When the client sends x, the server replies with the following string, and then ends the socket connection:

```
bye\r\n
```

And that's the communications protocol for the whole game. Keep in mind that it doesn't define anything about the physical form of the client object. As long as the client can make a TCP connection to the server and can send and receive the appropriate ASCII messages, it can work with the server. You can attach any type of physical inputs to the client, or you can write a client that sends all these messages automatically, with no physical input from the world at all (though that would be boring). Later in this chapter, you'll see a few different clients, each of which can connect to the server and play the game.

A Test Server

You need a server to get started. There's a lot of code to control the pong display that you don't need right now (you just want to confirm that the clients can connect). You can start out by using the test server from Chapter 4. It will let you listen for new clients, and send them messages by typing in the applet window that appears when you run the program. Run the server and open a telnet connection to it. Remember, it's listening on port 8080, so if your computer's IP address is, say, 192.168.1.45, then you'd connect like so: `telnet 192.168.1.45 8080`

If you're telnetting in from the same machine, you can use: `telnet localhost 8080`, or `telnet 127.0.0.1 8080`.

Whatever you type in the telnet window will show up in the server's debugger pane, and whatever you type in the server's applet window will show up at the client's command line. This result is useful, as it lets you use telnet as a test client later on. Next, you'll build a client and test it by getting it to connect to this server.

NOTE: If you want to get away with building only one client, you can telnet from a computer to connect to the server (telnet ip-address 8080**). However, you'll have to press Return after each command (r, l, or x), unless you make a change after you connect. On Mac OS X or Linux, press the telnet escape key combination (Control-]), type the following, and then press Return:**

```
mode character
```

On Windows telnet should not require any special configuration, but if you find otherwise, press Control-], type the following, and press Return twice:

```
set mode stream
```

NOTE: You'll now find that the server will accept commands immediately without requiring you to press Return after each one.

The Clients

Like the air quality meter client in Chapter 4, the pong client has a few states that you care about:

Program State	What to do
Disconnected from the server	Try to connect
Connecting to the server	Read bytes in, wait for a Connect message
Connected to the server	Play the game by sending `l`, `r`, or `x`

The client needs a few basic inputs in order to reach any of these states:

- An input for sending a connect message. The same input can be used to send a disconnect message.
- An input for sending a left message.
- An input for sending a right message.

To let the user know what the client device is doing, add some outputs to indicate its state:

- An output to indicate whether the client is connected to the server.
- An output to indicate when the connect/disconnect button is pressed.
- An output to indicate when it's sending a left message.
- An output to indicate when it's sending a right message.

It's always a good idea to put outputs on the client that indicate when it receives input from the user. For example, pressing the connect/disconnect button doesn't guarantee a connection, so it's important to separate the output that acknowledges a button push from the one that indicates successful connection. If there's a problem, this helps the user to determine whether the problem is with the connection to the server, or with the client object itself. In this application, the person using the client can look at the server's screen to see the state of the client. She can see a new paddle appear when her client connects, and she can see her paddle moving back and forth. In many cases, however, there's no way to get confirmation from the server except through the client, so local feedback is essential.

It's also good to indicate the client's status using outputs. You can see in the code that follows that in addition to indicating when the sensors are triggered, you'll also indicate whether the client is connected or disconnected. If this client had a more complex set of potential states, you'd need better indication.

For this project, I built two different clients. They have different methods of physical interaction and different input sensors, but they both behave the same way to the server. You can build either, or both, or use the principles from them to build your own. Building both and comparing the two will give you an idea of how the same protocol can result in very different behavior. One of these clients is much more responsive than the other, but the responsiveness has nothing to do with the communications protocol. It's all in the sensing and in the player's action.

X

Client #1: A Seesaw Client

This client is basically a seesaw, as shown in Figure 5-1. The basic structure is a plank mounted on a piece of pipe. The electronics sit in a case at the center of the plank. The user puts her feet on either end of the plank and balances her weight on the plank. As she tilts to the left, the client sends a left message. As she tilts right, it sends a right message. The pushbutton in the center allows her to connect to the server or disconnect by stomping on it. Use a sturdy button for this, and make sure that the LEDs are out of foot-stomping range.

In Chapter 4, you saw the Lantronix Micro in action. In this chapter and the following ones, you'll see the Lantronix XPort. The XPort is smaller than the Micro module, and operates on 3.3 volts instead of 5 volts. Its input pins can tolerate 5 volts, though, so it's easy to interface to a 5-volt microcontroller. There are no other significant differences, so you can use the Micro if you prefer. For the purposes of these projects, the two are functionally identical — use whichever you please.

The circuit for the seesaw client uses an accelerometer to read the tilt of the seesaw. The one shown in Figure 5-2 is a three-axis accelerometer, but you'll only use one axis. You can use a different accelerometer if you choose, as long as it outputs an analog voltage for tilt along one axis.

Because the XPort's pins don't fit nicely in a breadboard, you need the RJ45 breakout board. Figure 5-3 shows the XPort mounted on the breakout board.

A pencil box from your friendly neighborhood stationery store works well as a case for this project. Drill holes in the lid for the switch and the LED, cut holes in the side for the XPort and Arduino jacks, and you're all set. Figure 5-4 shows the outside of the pencil box case, and Figure 5-5 shows the inside and a detail of the wiring to the switches and LEDs. The configuration for the XPort is identical to the one for the Micro in Chapter 4:

```
*** Channel 1
Baudrate 9600, I/F Mode 4C, Flow 00
Port 10001
Remote IP Adr: --- none ---, Port 00000
Connect Mode : D4
Disconn Mode : 00
Flush   Mode : 00
```

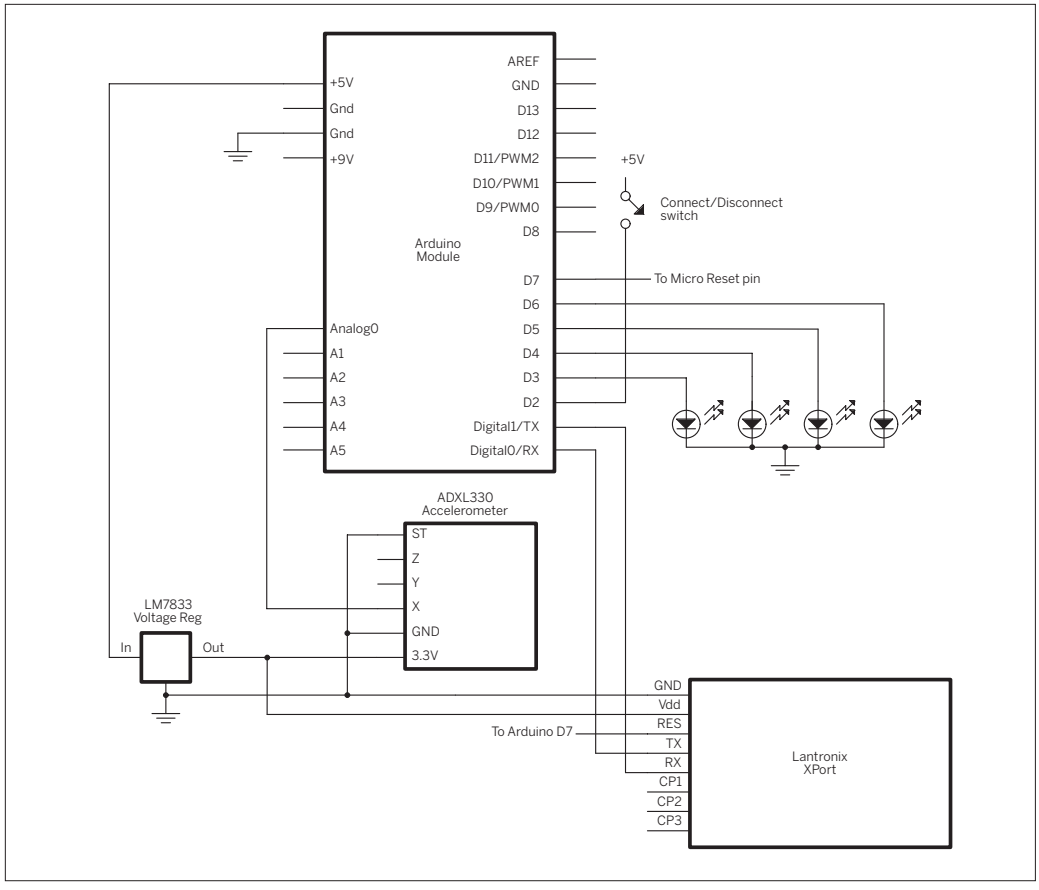
As you're using a Lantronix device again, the code for the client has some elements in common with the code for the air quality meter in Chapter 4. You'll reuse the device Connnect() method, the resetDevice() method, and the blink() method from that application.

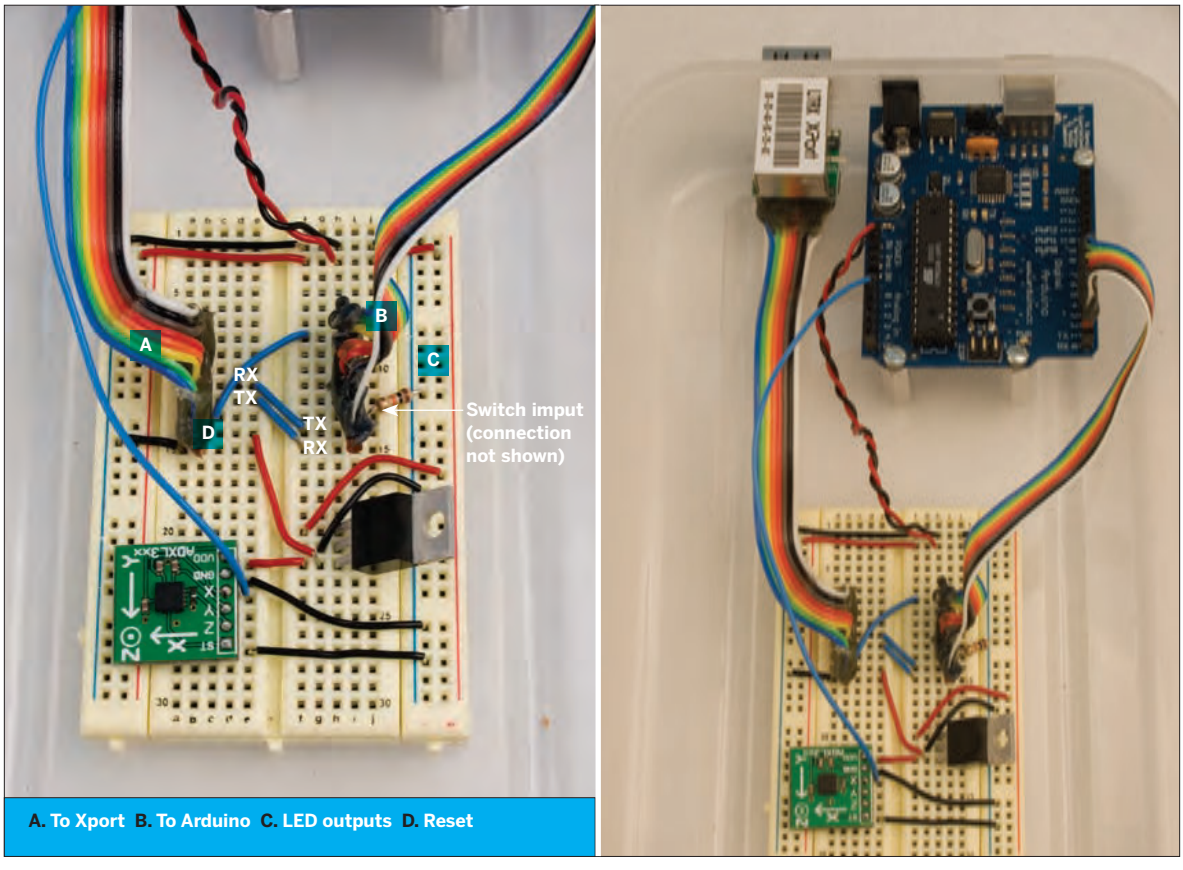
MATERIALS FOR CLIENT 1

- **1 Lantronix embedded device server** available from many vendors, including Symmetry Electronics (www.semiconductorstore.com), part number CO-E1-11AA (Micro) or WM11A0002-01 (WiMicro), or XP1001001-03R (XPort)
- **1 RJ45 breakout board** SparkFun (sparkfun.com) part number BOB-00716 (needed only if you're using an XPort)
- **1 3.3V regulator** The LM7833 from SparkFun, part number COM-00526, or the MIC2940A-3.3WT from Digi-Key, part number 576-1134-ND, will each work well.
- **1 solderless breadboard** such as Digi-Key (digikey.com) part number 438-1045-ND, or Jameco (www.jameco.com) part number 20601
- **1 Arduino** module or other microcontroller
- **4 LEDs** It's best to use at least two colors with established semantics: a big red one, a big green one, and two others of whatever color suits your fancy.
- **1 accelerometer** The circuit shown uses an ADXL330 accelerometer, available in a module from SparkFun (part number SEN-00692), but most any analog accelerometer should do the job.
- **1 push button** Use one that's robust and can stand a good stomping.
- **1 pencil box**
- **1 pipe, 2- to 3-inches in diameter, approximately 6–8 inches long** Wood, sturdy cardboard, or sturdy plastic will work.
- **1 plank of scrap wood**

Figure 5-1
The seesaw client. Note the red and green LEDs to indicate left and right tilt. This client follows nautical tradition port (left) is red, and starboard (right) is green. Feel free to be less jaunty in your own choices, as long as they're clear.

Figure 5-2
The seesaw client schematic. The detail photos on the following page show the wiring on the breadboard.





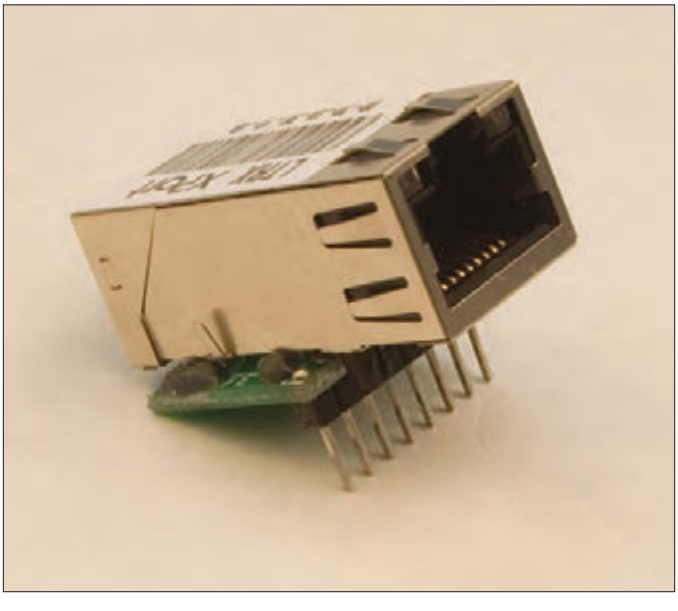


Figure 5-2

» *continued from previous page*

The seesaw client detail photos. The schematic is shown on the previous page.

Figure 5-3

The XPort mounted on an RJ45 breakout board. A couple of dabs of hot glue on the breakout board serve as spacers to keep the metal of the XPort's case from contacting the outside pins on the breakout board.

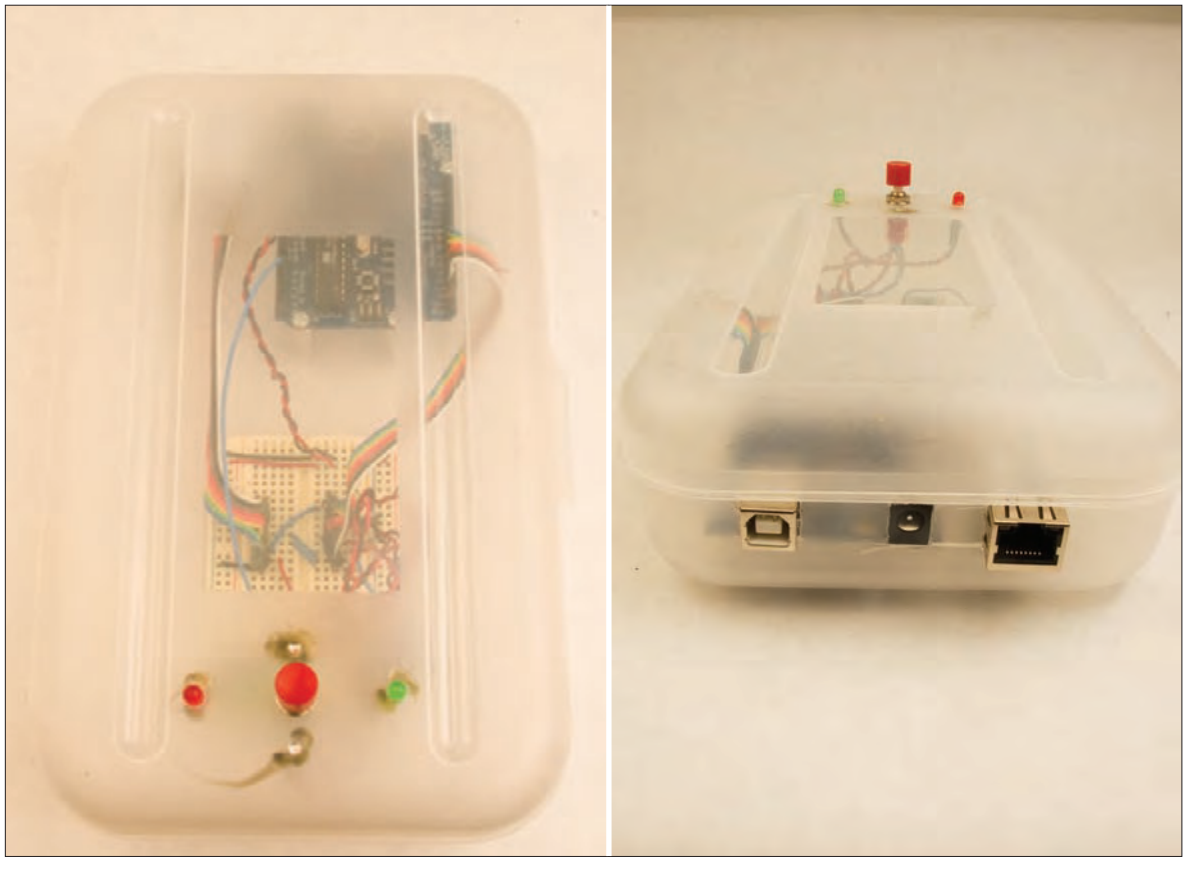


Figure 5-4
The outside of the seesaw control box, and the connectors.

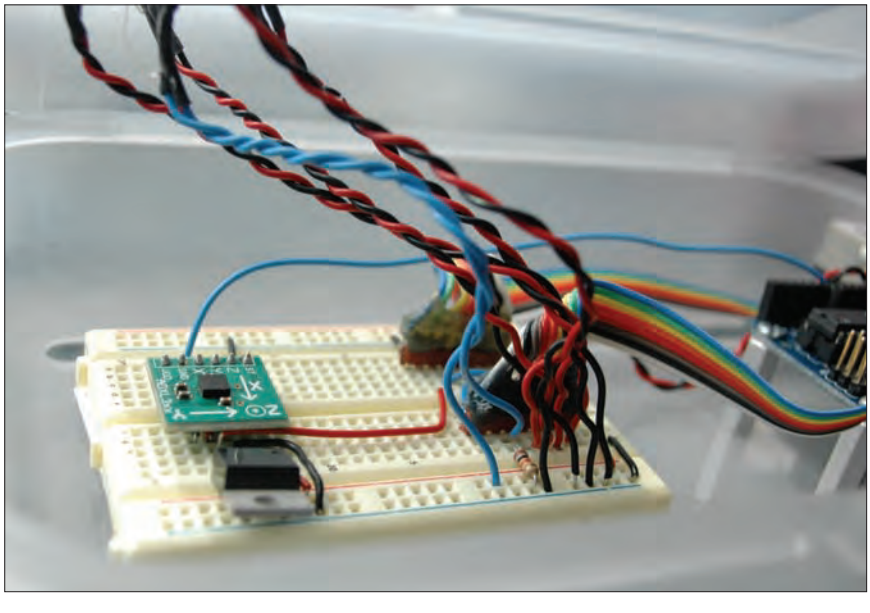


Figure 5-5
Inside the box, showing the wiring to the LEDs and the switch in the box cover.

Try It First, start with the variable declarations and definitions. From the preceding explanation of the program's states and the inputs and outputs earlier, this is all straightforward:

```
/*
  pong client
  Language:  Wiring/Arduino

  This program enables an Arduino to control one paddle in a
  networked pong game.

 */

// 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
```

» You need global variables to handle the serial communication, track the sensors, and keep track of the state of the client:

```
// 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, they are set to 0.
 */
byte paddleMessage = 0;           // message sent to make a paddle move
byte connectMessage = 0;          // message sent to connect or disconnect
```

» The setup() method just sets all the I/O modes, starts the serial port, and resets the Lantronix device:

```
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:
```

»

Continued from opposite page.

```
  Serial.begin(9600);

  // reset the Lantronix device:
  resetDevice();
  // blink the exit button LED to signal that we're ready for action:
  blink(3);
}
```

In the main loop, you read the sensors, set the indicator LEDs, and take an action that depends on the state that the client's in:

```
void loop() {
  // read the inputs:
  readSensors();
  // set the indicator LEDS:
  setLeds();
  // check the state of the client and take appropriate action:
  stateCheck();
}
```

Read the sensors (both the accelerometer and the connect/disconnect switch) using a method called readSensors(). It determines the state of the sensors and indicates the results by setting the values of the connectMessage and paddleMessage variables.

Send a message only when the accelerometer tilts far enough to the left or right. To discover the thresholds for left and right, write a simple program that just reads the analog input that the accelerometer is attached to and print it out. Then get on the seesaw and rock back and forth while watching the values.

The one I used for this example gave a reading of approximately 450 when it was level, and tilted to above 500 to the left, and below 420 to the right. Your results may vary. Based on those values, here's the readSensors() method:

```
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;
  }
```

Change these values to match the values for your own accelerometer.

You can't just send a message every time the connect button is high. You want to send a message only when the button changes from low to high, indicating that the player just pressed it. This code block checks for a low-to-high transition by comparing the state of the button with its previous state:

```
  // 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 he or she hit the button:
      digitalWrite(connectButtonLED, HIGH);
      connectMessage = 'x';
    }
  }
  // save the state of the exit button for next time you check:
  lastConnectButton = connectButton;
}
```

Once you've checked the sensors, you can set the LED indicators:

```
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);
  }
}
```

» Next, take action depending on the state of the client and the messages to be sent:

```
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 tried to connect:
      connectMessage = 0;
    }
    break;
    // if you sent a connect message but haven't connected, 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:
      if (inByte == 'C') {
        status = connected;
      }
      else {  // if you got anything other than a C, try again:
        deviceConnect();
      }
    }
    break;
  }
}
```

Finally, here are the deviceConnect(), resetDevice(), and blink() methods mentioned earlier. Be sure to replace 192.168.1.20 with the IP address of the computer running the Processing server and 8080 with the port number that the server is listening on:

That's the whole client. Once you've assembled the client, compile and run this code on it, and attempt to connect to the test server. You should see it make connections and send messages based on the state of the client and the values from the sensors. If you don't, use the troubleshooting methods at the end of Chapter 4 to determine what's wrong. For the full code, see Appendix C.

```
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);
  }
}
```

You will need to change this to the IP address and port that your Processing server is running on.

Client #2: A Stepper Client

For Client #2, I wanted to see whether there's a big difference between a foot-based input device that requires you to balance and one that you can stand on easily. I found an exercise stepper in the trash that had a missing screen, but the steps still worked. So I attached force-sensing resistors to the foot pads and made a second client. The rest of the client's functionality is the same as the seesaw client.

The stepper client, shown in Figure 5-6, is very similar to the seesaw client. The only difference is that the accelerometer has been replaced by two force-sensing resistors. You can see the FSRs at the back of the foot pads. After much experimenting, I found that I got the most reliable results when the sensors were under the heels. They lift off the pads when you step, giving a nice range of contrast to the readings.

The physical construction for the stepper client is almost identical to the seesaw client, You can use the same pencil box case with the same LED setup. Just drill two holes for the leads to the FSRs and you're all set.

The code for the stepper client is also very similar to the seesaw client. All you have to change is the readSensors() method. Everything else stays the same. That's the beauty of using a clear, simple protocol: it doesn't matter what the physical input is, as long you can map changes recorded by the sensors to a left-right movement, and program the microcontroller to send the appropriate messages. It's worthwhile to try building both these clients, or one of your own, to look at how different physical affordances can affect the performance of different clients, even though it "appears" the same to the server.

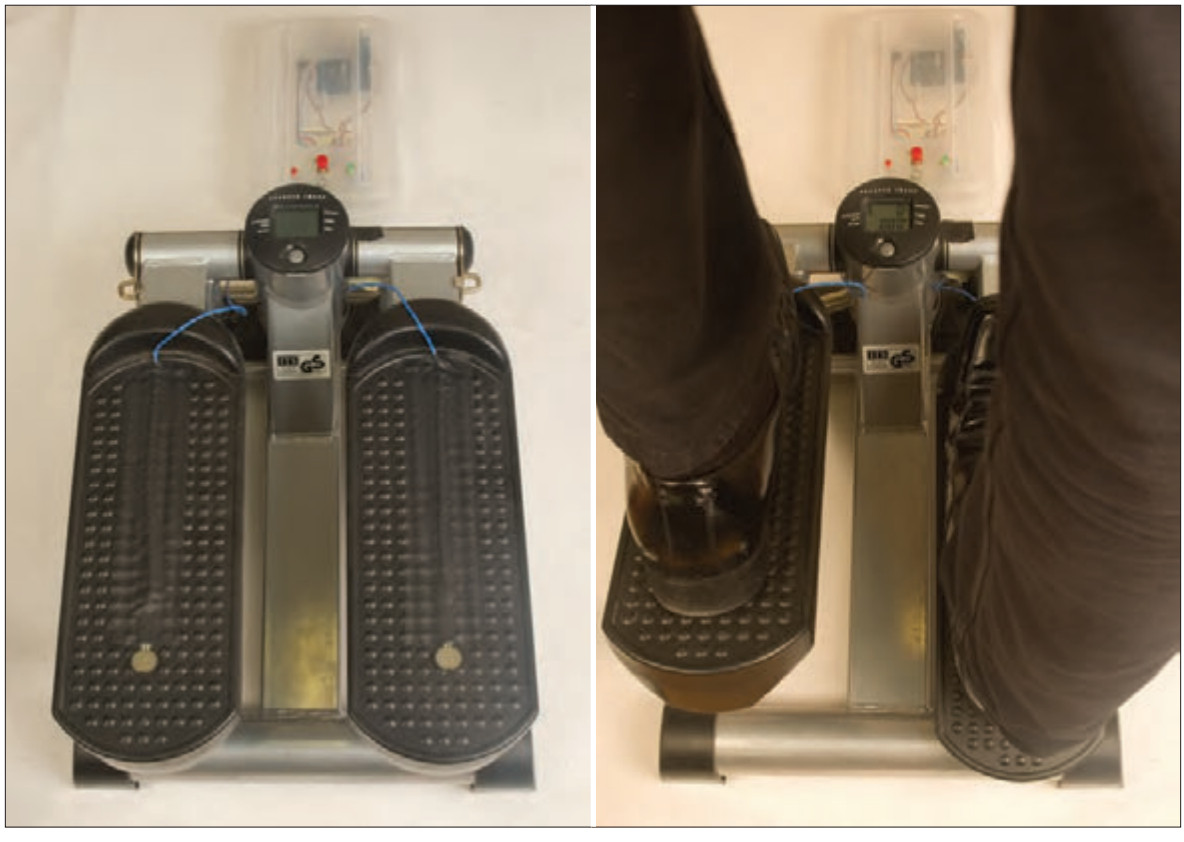


Figure 5-6
The stepper client.

MATERIALS FOR CLIENT 2

- **1 Lantronix embedded device server** Available from many vendors, including Symmetry Electronics (www.semiconductorstore.com) as part number CO-E1-11AA (Micro), WM11A0002-01 (WiMicro), or XP1001001-03R (XPort)
- **1 RJ45 breakout board** SparkFun part number BOB-00716 (needed only if you're using an XPort)
- **1 3.3V regulator** The LM7833 (SparkFun part number COM-00526) or the MIC2940A-3.3WT (Digi-Key part number 576-1134-ND) will work well.
- **1 solderless breadboard** such as Digi-Key part number 438-1045-ND or Jameco (www.jameco.com) part number 20601
- **1 Arduino module** or other microcontroller
- **4 LEDs** For semantic reasons, it's best to use a big red one, a big green one, and two others of whatever color suits your fancy.
- **2 force-sensing resistors (FSRs)**
- **1 pushbutton** Use one that's robust and can stand a good stomping.
- **1 pencil box**
- **1 exercise stepper** The one I found in the trash was a Sharper Image model, but you could also fake it with two foot-size wooden blocks mounted on foam rubber, with the FSRs mounted on the top of the blocks.

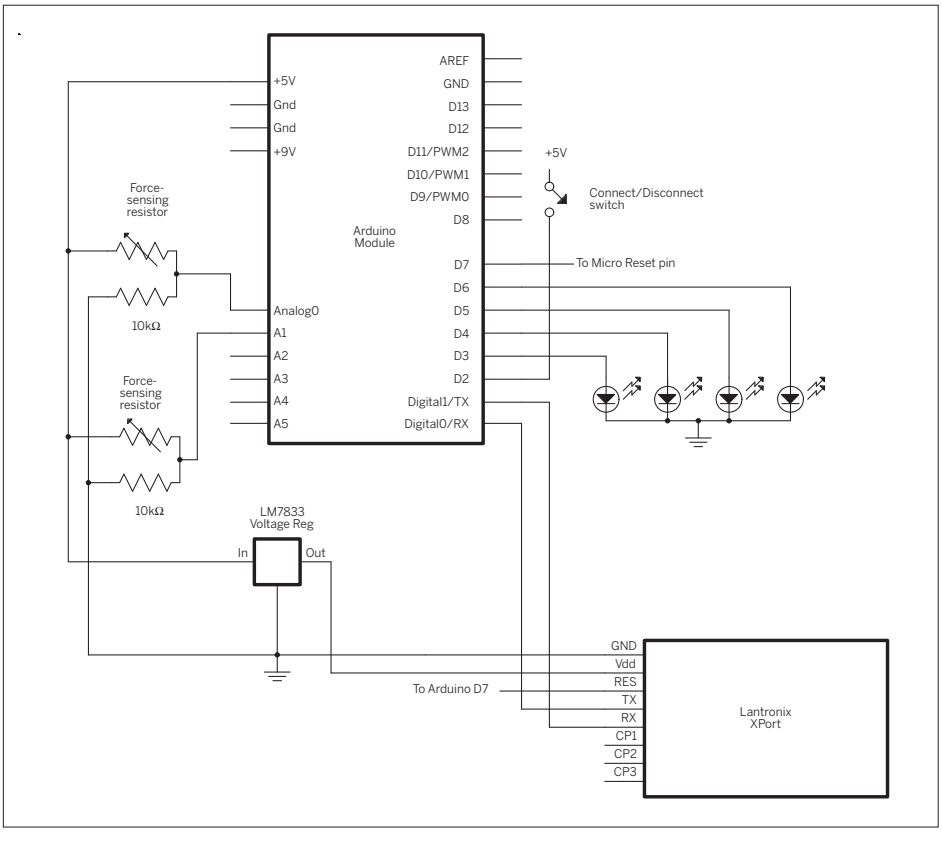
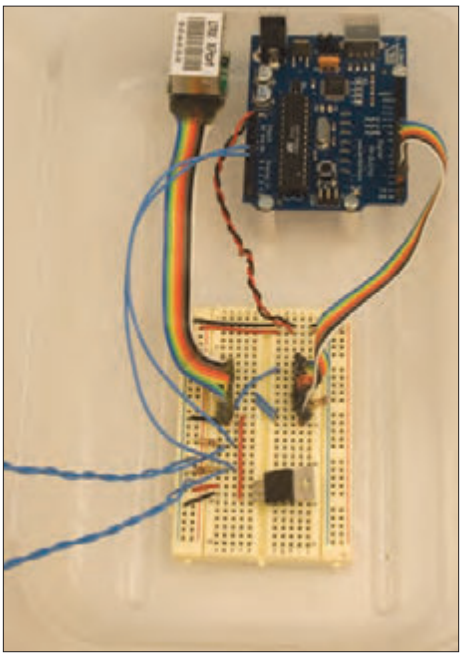
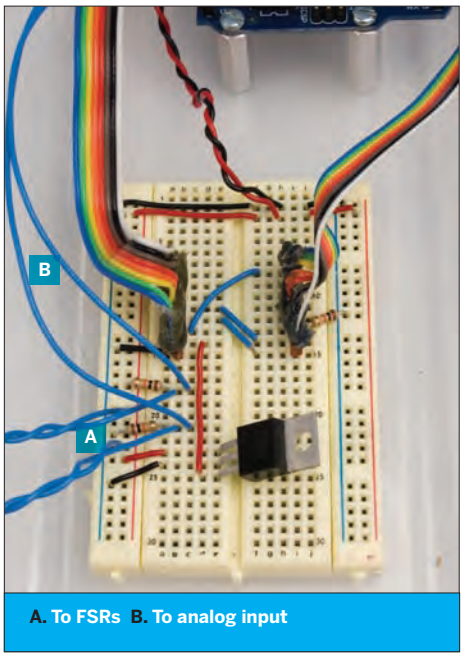


Figure 5-7
The stepper client circuit.





A. To FSRs B. To analog input

The next listing is the readSensors() method for the stepper client. All the rest of the code for this client is identical to the previous client. Just as with the seesaw client, you have to discover experimentally what the thresholds for your sensors are, so you may need to write a short program to do that. In this case, the left and right sensors are electrically identical, so they should produce similar values when subjected to similar forces (for example, your left foot and your right foot). So one threshold value will do the job for both.

You can test this client the same way you did the last one, using the test server program shown earlier. Once you're sure it's sending messages correctly, it's time to write the *ping* pong server.

```
void readSensors() {
  int threshold = 500;
  int  left = analogRead(0);
  delay(10);
  int right = analogRead(1);
  delay(10);
  if (left >= right + threshold) {
    paddleMessage = 'l';
  } else if (right >=left + threshold) {
    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) {
      digitalWrite(connectButtonLED, HIGH);
      connectMessage = 'x';
    }
  }
  lastConnectButton = connectButton;
}
```

Change this value to match the value for your own sensors.

The Server

The tasks to be handled by the server can be divided into two groups: tasks related to the game play, like animating the paddles and the ball and scoring, and tasks related to tracking new clients. To manage it all most effectively, you're going to use an object-oriented programming approach. If you've never done this before, there are a few basics you need to know in advance.

Anatomy of a Player Object

The most important thing to know is that all objects have properties and behaviors. You can think about an object's properties in much the same way as you think about physical properties. For example, a pong paddle has width and height, and it has a location, which you can express in terms of its horizontal and its vertical position. In your game, the paddles will have another important property: each paddle will be associated with a client. Of course, clients have properties as well, so each paddle will inherit from its client an IP address. You'll see all of these in the code that defines a paddle as an object.

A paddle also has a characteristic behavior: it moves left or right. That behavior will be encoded into the paddle as a method called movePaddle(). This behavior will update the properties of the paddle that define its location. A second behavior called showPaddle() will actually draw the paddle in its current location. You'll see later why these are kept separate.

Code It To define an object in Processing (and in Java), create a code block called a class. Here's the beginning of the class that defines a player in the pong server:

```
public class Player {
  // declare variables that belong to the object:
  float paddleH, paddleV;
  Client  client;
```

The variables declared at the beginning of the class as shown in the example are called instance variables. Every new instance of the class created makes its own copies of these variables. Every class has a constructor method. This method gets called to call the object into existence.

You've already used constructors. When you made a new Serial port in Processing, you called the constructor method for the Serial class with something like myPort = new Serial(this, portNum, portSpeed).

Here's the constructor for the Player class. It comes right after the instance variables in your code. As you can see, it just takes the values you give it when you call for a new Player and assigns them to variables that belong to an instance (an individual player) of the class:

```
public Player (int hpos, int vpos, Client someClient) {
  // initialize the local instance variables:
  paddleH = hpos;
  paddleV = vpos;
  client = someClient;
}
```

Next come the two other methods mentioned earlier, movePaddle() and showPaddle(). As you can see, they use the instance variables (paddleH, paddleV, and client) that belong to the object to store the location of the paddle, and to draw it:

```
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 );
}
}
```

This bracket closes the class

That's all the code to define a Player. Put this code at the end of your program (shown next), just as if it were another method. To make a new Player object, write something like Player newPlayer = new Player(h, v, thisClient).

When you do this, the new Player, and all its instance variables and methods, are accessible through the variable called newPlayer (the new Player is not actually stored in this variable; it's stuffed away in a portion of memory somewhere that you can get at through the newPlayer variable). Keep an eye out for this in the program.

The Main pong Server Program

Before you write the code for the server as a whole, it's useful to make a map of what happens. Figure 5-8 shows the main tasks and functions. A few details are left out for clarity's sake, but what's clear are the main relationships between the methods that run the program (setup(), draw(), and serverEvent()) and the Player objects. As with any program, the setup() method kicks things off, then the draw() method takes over. The latter sees to it that the screen is updated and listens to any existing clients. If a new client connects, a serverEvent() message is generated, which causes the method of that name to run. That method creates new Player objects. The draw() method takes advantage of the behaviors inside the Player objects to move and draw their paddles.

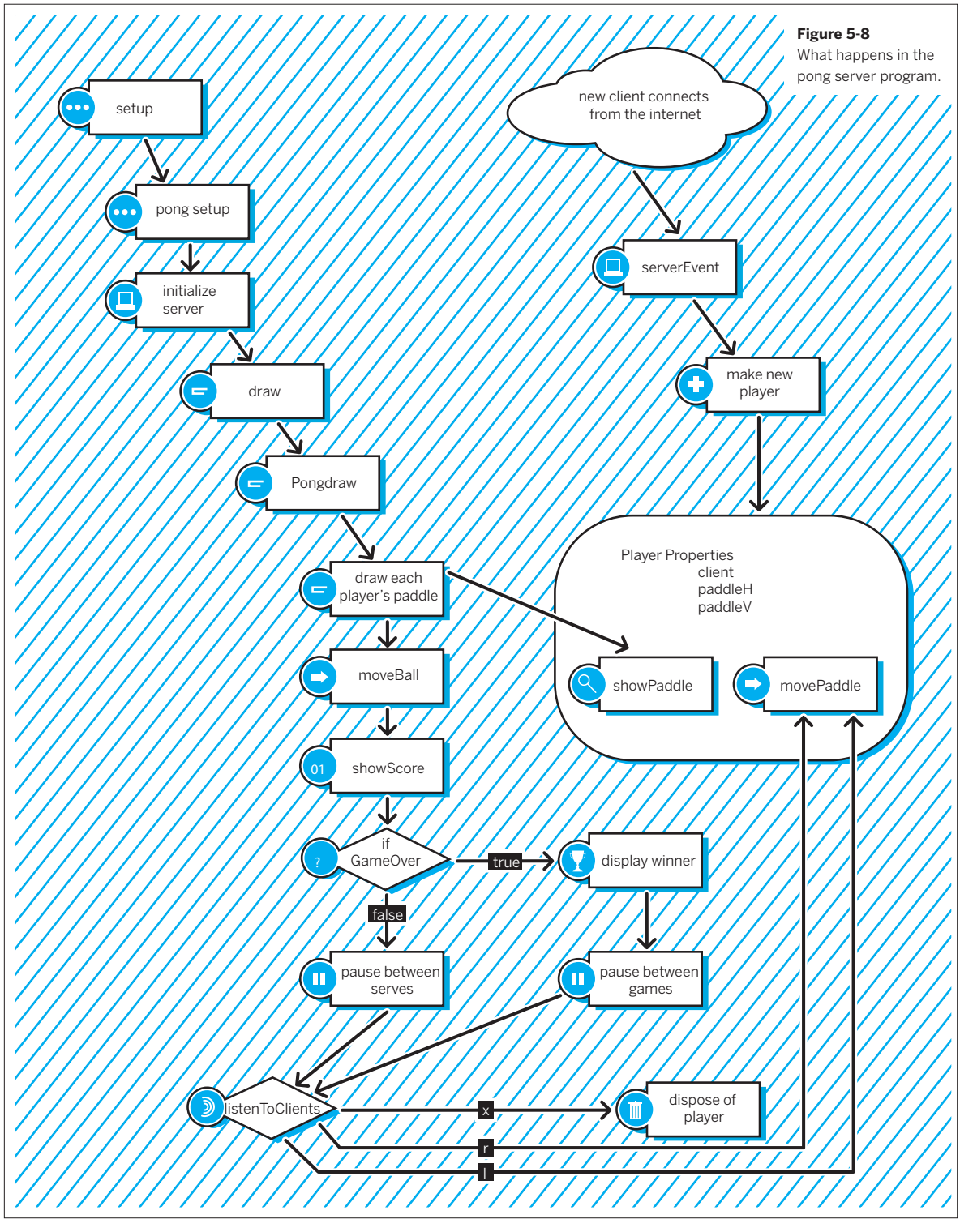


Figure 5-8
What happens in the pong server program.

The first thing to do in the server program itself is to define the variables. They're grouped here by those needed for keeping track of clients versus those needed for managing the graphics of the game play:

```
// 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
```

This program uses a Java class, ArrayList, that's not explained in the Processing reference guide. Think of it as a super-duper array. ArrayLists don't have a fixed number of elements to begin with, so you can add new elements as the program continues. It's useful when you don't know how many elements you'll have. In this case, you don't know how many clients you'll have, so you'll store them in an ArrayList, and add each new client to the list as it connects. ArrayLists include some other useful methods. For more on ArrayList and other Java classes, check out Head First Java (O'Reilly, 2005) or the online documentation at java.sun.com.

```
// 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 or not 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 or not 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
```

The setup() method starts the server and calls pongSetup(), which sets all the initial conditions for the game:

```
void setup()  {
  // set up all the pong details:
  pongSetup();
  // start the server:
  myServer = new Server(this, port);
}
```

Here's the pongSetup() method :

```
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);
}
```

The draw() method updates the screen using a method called pongDraw(), and listens for any messages from existing clients using a method called listenToClients().

```
void draw() {
  pongDraw();
  listenToClients();
}
```

When new clients connect to the server, the net library's serverEvent() method is called automatically; your Processing sketch must implement this method in order to respond to the event. It uses the new client to create a new Player object using a method called makeNewPlayer(). Here's the serverEvent() method:

```
// 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);
    }
  }
}
```

Now that you've seen the draw()and the serverEvent() methods, it's time to look at the methods they call. It's best to start with the creation of a new Player, so here's the makeNewPlayer() method. It checks the number of players so far created by counting the number of Players in the ArrayList called playerList. If there's an even number of Players, the new Player is added to the top team, and is positioned below the last top player. If there's an odd number of Players, the new one goes on the bottom team. The variables nextTopPaddleV and nextBottomPaddleV keep track of the positions for the next players on each team.

Here's the makeNewPlayer()method:

```
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);
    // if the last player's on the top, add to the bottom:
    if (lastPlayerAdded.paddleV == nextTopPaddleV) {
      nextBottomPaddleV = nextBottomPaddleV - paddleHeight * 2;
      v = nextBottomPaddleV;
   }
```

Continued from opposite page.

```
    // if the last player's on the bottom, add to the top:
    else if (lastPlayerAdded.paddleV == nextBottomPaddleV) {
      nextTopPaddleV = nextTopPaddleV + paddleHeight * 2;
      v = nextTopPaddleV;
    }
  } else {  // if there are no players, add to the top:
    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");
}
```

Once a new Player has been created, you need to listen continuously for that Player's client to send any messages. The more often you check for messages, the tighter the interactive loop between sensor and action.

The listenToClients() method, called continuously from the draw() method, listens for messages from clients. If there's data available from any client, this method takes action. First it iterates over the list of Players to see whether each one's client is speaking. Then it checks to see whether the client sent any of the game messages (that is, l for left, r for right, or x for exit). If any of those messages was received, the program acts on the message appropriately.

```
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, 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) {
    case 'x':  // If the client says "exit", disconnect it
```

Continued from previous page.

```
      // 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;
    }
  }
}
```

So far you've seen how the server receives new connections (using serverEvent()), creates new Players from the new clients (using makeNewPlayer()), and listens for messages (using listenToClients()). That covers the interaction between the server and the clients. In addition, you've seen how the Player class defines all the properties and methods that are associated with each new player. Finally, it's time to look at the methods for controlling the drawing of the game. pongDraw(), called from the draw() method, is the main method for this. This method has four tasks:

- Iterate over the playerList and draw all the paddles at their most current positions.
- Draw the ball and the score.
- If the game is over, show a "Game Over" message and pause.
- Pause after each volley, then serve the ball again.

Show It Here is the pongDraw() method:

```
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();
  }
```

You saw earlier that the listenToClients() method actually updates the positions of the paddles using the movePaddle() method from the Player object.

» That method doesn't actually draw the paddles, but this one does, using ach Player's showPaddle() method. This is why the two methods are separated in the object. Likewise, the moveBall() method, called here, checks to see if the ball hit a paddle or a wall, and calculates its new position from there, but doesn't draw the ball itself, as the ball needs to be drawn even if it's not in motion:

```
  // calculate ball's position:
  if (ballInMotion) {
    moveBall();
  }
  // draw the ball:
  rect(ballPosH, ballPosV, ballSize, ballSize);

  // show the score:
  showScore();
```

If the game is over, the program stops the serving and displays the winner for four seconds:

```
  // 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();
  }
```

After each point is scored, the program takes a two-second pause. If there aren't at least two players after that pause, it doesn't serve another ball. This is to keep the game from running when there's no one to play:

That closes out the pongDraw() method itself. It calls a few other methods: moveBall(), which calculates the ball's trajectory; showScore(), which shows the score; and newGame(), which resets the game.

```
  // 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);
    }
  }
}
```

The first thing moveBall() does is to check if the position of the ball intersects any of the Players' paddles. To do this, it has to iterate over playerList, pull out each Player, and check to see if the ball position is contained within the rectangle of the paddle. If the ball does intersect a paddle, then its vertical direction is reversed:

```
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 to see if 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;
```

Continued from previous page.

```
      // 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 above the top of the screen or below the bottom, then one team or the other has scored:

```
  // 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;
  }
```

» Finally, moveBall() checks to see whether the ball hits one of the sides of the screen. If so, the horizontal direction is reversed:

```
  // 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;
}
```

» The newGame() method just stops the game play and resets the scores:

```
void newGame() {
  gameOver = false;
  topScore = 0;
  bottomScore = 0;
}
```

» The showScore() method prints the scores on the screen:

```
public void showScore() {
  textSize(24);
  text(topScore, 20, 40);
  text(bottomScore, 20, height - 20);
}
```

» Finally, moveBall() calls a method called resetBall(), that resets the ball at the end of each point. Here it is:

For the final code, see Appendix C.

```
void resetBall() {
  // put the ball back in the center
  ballPosV = height/2;
  ballPosH = width/2;
  ballInMotion = false;
  delayCounter = millis();
}
```

The beauty of this server is that it doesn't really care how many clients log into it; everyone gets to play *ping* pong. There's nothing in the server program that limits the response time for any client, either. The server attempts to satisfy everyone a soon as possible. This is a good habit to get in. If there's a need to limit the response time in any way, don't rely on the server of the network to do that. Whenever possible, let the network and the server remain dumb, fast, and reliable, and let the clients decide how fast they want to send data across. Figure 5-9 shows a screenshot of the server with two clients.

Once you've got the clients speaking with the server, try designing a new client of your own. See if you can make the ultimate *ping* pong paddle.

X

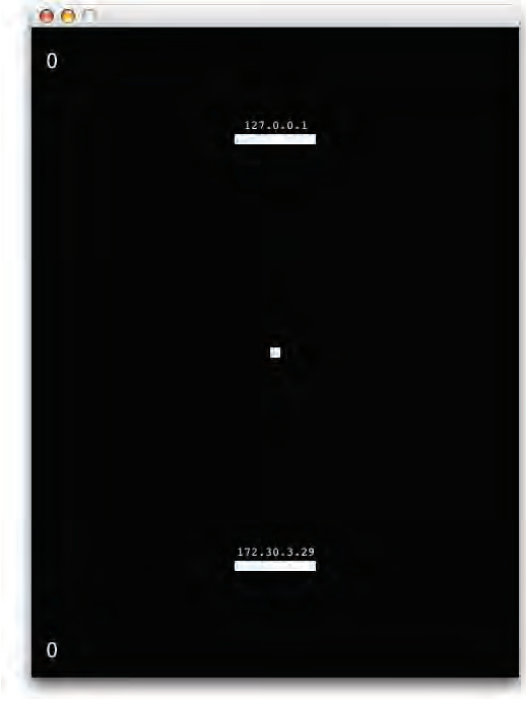


Figure 5-9
The output of the ping pong server sketch.

Conclusion

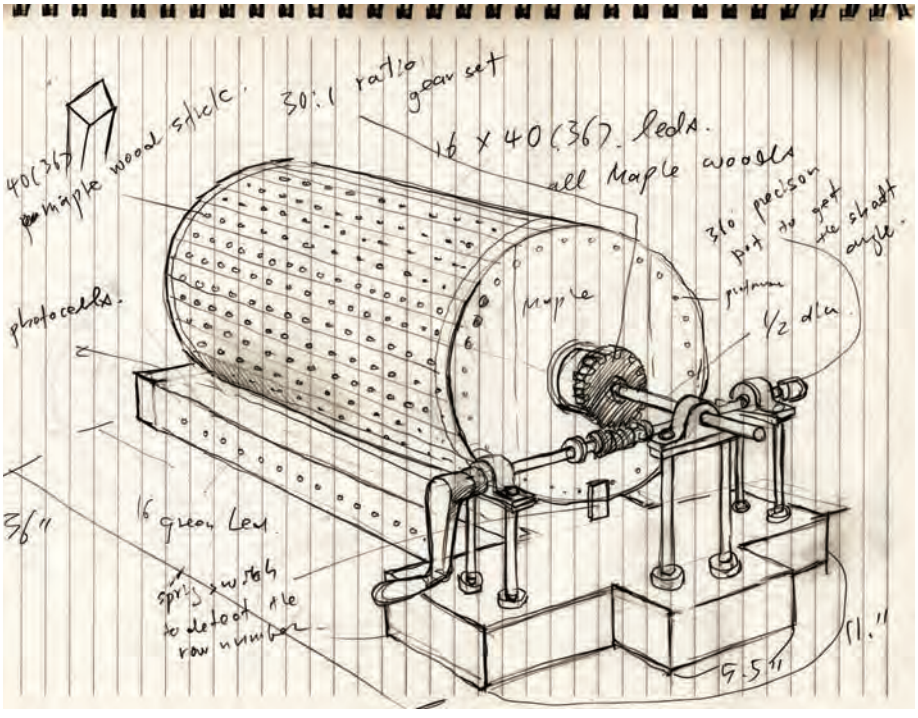
The basic structure of the clients and server in this chapter can be used any time that you want to make a system that manages synchronous connections between several objects on the network. The server's main jobs are to listen for new clients, to keep track of the existing clients, and to make sure that the right messages reach the right clients. It must place a priority on listening at all times.

The client should also place a priority on listening, but it has to juggle listening to the server with listening to the physical inputs. It should always give a clear and immediate response to local input, and it should indicate the state of the network connection at all times.

The protocol that the objects in this system speak to each other should be as simple and as flexible as possible. Leave room to add commands, because you never know when you might decide to add something. Make sure to build in responses where appropriate, like the "hi" and "bye" responses from the server. Keep the messages unambiguous, and if possible, keep them short as well.

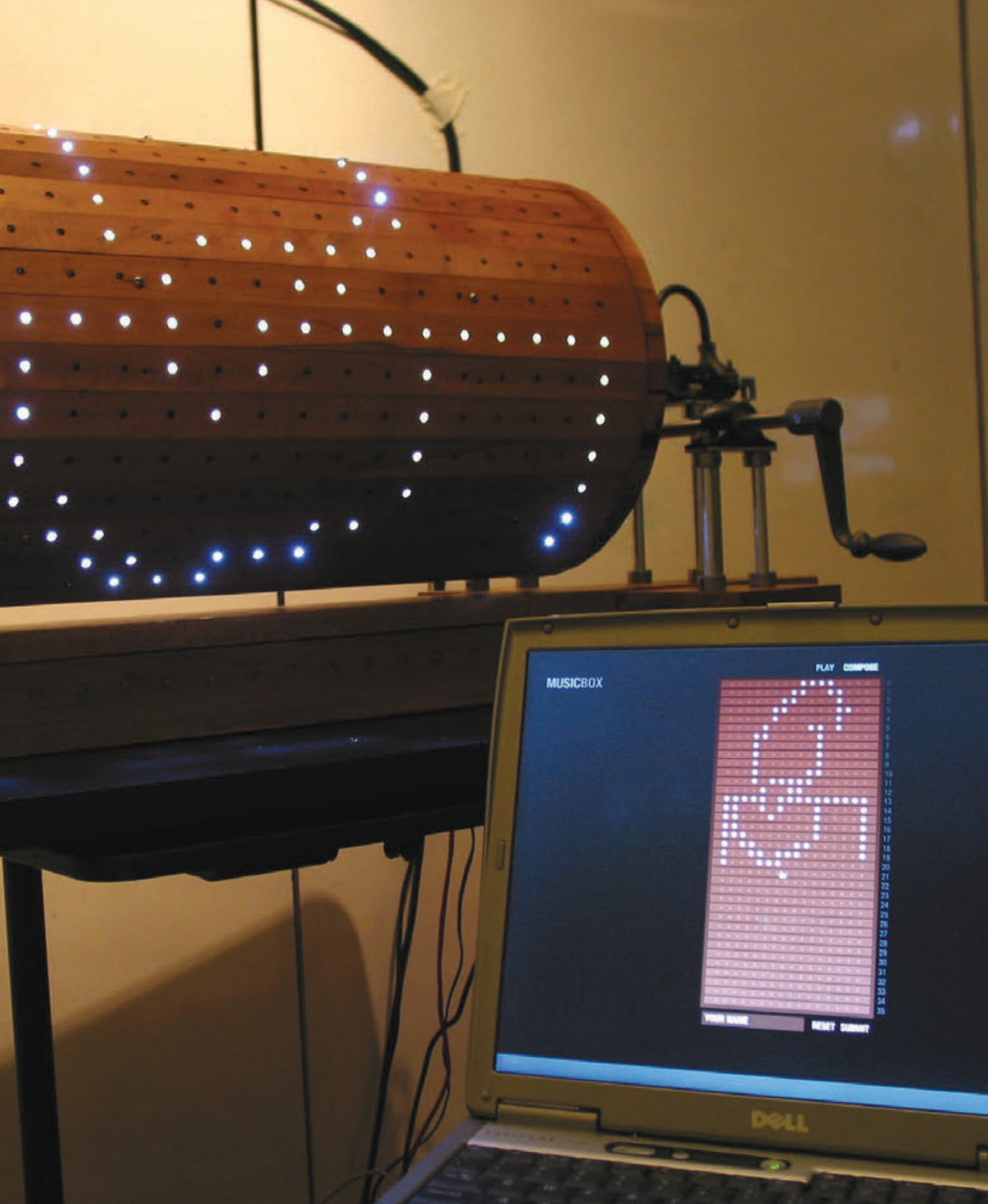
Finally, make sure you've got a reliable way to test the system. Simple tools like the telnet client and the test server will save you much time in building every multi-player server, and help you get to the fun sooner.

Now you've seen examples of both asynchronous client-server exchanges (the HTTP system in Chapter 4) and synchronous exchanges (the chat server here). With those two tools, you can build almost any application in which there's a central server and a number of clients. For the next chapter, you'll step away from the Internet and take a look at various forms of wireless communication.
X



« At right
Jin-Yo Mok's original sketches of the music box.

» At left
The music box composition interface.


MUSICBOX
PLAY
COMPOSE
YOUR NAME
RESET
SUBMIT
DELL

6

MAKE: PROJECTS

Wireless Communication

If you're like most people interested in this area, you've been reading through the early chapters thinking, "but what about wireless?" Perhaps you're so eager that you just skipped straight to this chapter. If you did, go back and read the rest of the book! In particular, if you're not familiar with serial communication between computers and microcontrollers, you'll want to read Chapter 2 before you read this chapter. This chapter explains the basics of wireless communication between objects. In it, you'll learn about two types of wireless communication, and build some working examples.

◀◀ **Alex Beim's Zygotes** (www.tangibleinteraction.com) are lightweight inflatable rubber balls lit from within by LED lights. The balls change color in reaction to pressure on their surface, and communicate with a central computer using ZigBee radios. A network of zygotes at a concert allows the audience to have a direct effect not only on the balls themselves, but also on the music and video projections to which they are networked. *Photo courtesy of Alex Beim.*

The early part of this chapter covers how wireless works, and what makes it stop working, giving you some background and starting places for troubleshooting. The second half of the chapter contains examples. The topic is so broad, even a survey of several different devices only begins to cover the tip of the iceberg. For that reason, the exercises in this chapter will be less fully developed applications than the previous ones. Instead, you'll just get the basic "Hello World!" example for several different forms of wireless device.

X

Why Isn't Everything Wireless?

The advantage of wireless communication seems obvious at first: no wires! This makes physical design much simpler for any project where the devices have to move and talk to each other. Wearable sensor systems, digital musical instruments, and remote control vehicles are all simplified physically by wireless communication. However, there are some limits to wireless communication that you should consider before going wireless.

Wireless communication is never as reliable as wired communication

You have less control over the sources of interference. You can insulate and shield a wire carrying data communications, but you can never totally isolate a radio or infrared wireless link. There will always be some form of interference, so you must make sure that all the devices in your system know what to do if they get a garbled message, or no message at all, from their counterparts.

Wireless communication is never just one-to-one communication

The radio and infrared devices mentioned here broadcast their signals for all to hear. Sometimes that means they interfere with the communication between other devices. For example, Bluetooth, most Wi-Fi radios (802.11b, g, and n) and ZigBee (802.15.4) radios all work in the same frequency range, 2.4 gigahertz. They're designed to not cause each other undue interference, but if you have a large number of ZigBee radios working in the same space as a busy Wi-Fi network, for example, you'll get interference.

Wireless communication does not mean wireless power

You still have to provide power to your devices, and if they're moving, this means using battery power. Batteries add weight, and they don't last forever. The failure of a battery when you're testing a project can cause all kinds of errors that you might attribute to other causes. A classic example of this is the Mystery Radio Error. Many radios consume extra power when they're transmitting. This causes a slight dip in the voltage of the power source. If the radio isn't properly decoupled with a capacitor across its power and ground leads, the voltage can dip low enough to make the radio reset itself. The radio may appear to function normally when you're sending it serial messages, but it will never transmit, and you won't know why. When you start to develop wireless projects, it's good practice to make sure that you have the communication working first using a regulated, plugged-in power supply, and then create a stable battery supply.

Wireless communication generates electromagnetic radiation

This is easy to forget about, but every radio you use emits electromagnetic energy. The same energy that cooks your food in a microwave sends your mp3 files across the Internet. And while there are many studies indicating that it's safe at the low operating levels of the radios used here, why add to the general noise if you don't have to?

Make the wired version first

The radio and IR transceivers discussed here are replacements for the communications wires used in previous chapters. Before you decide to add wireless to any application, it's important to make sure you've got the basic exchange of messages between devices working over wires first.

X

Two Flavors of Wireless: Infrared and Radio

There are two common types of wireless communication in most people's lives: infrared light communication and radio communication. The main difference between them from a user's or developer's position is their directionality.

Television remote controls typically use infrared (IR) communication. Unlike radio, it's dependent on the orientation between transmitter and receiver. There must be a clear line of sight between the two. Sometimes IR can work by bouncing the beam off another surface, but it's not as reliable. Ultimately, the receiver is an optical device, so it has to "see" the signal. Car door openers, mobile phones, garage door remote controls, and many other devices use radio. All of these work whether the transmitter and receiver are facing each other or not. They can even operate through walls, in some cases. In other words, their transmission is omnidirectional. Generally, IR is used for short-range line-of sight applications, and radio is used for everything else.

Transmitters, Receivers, and Transceivers

There are three types of devices common to both IR and RF systems: transmitters, which send a signal, but can't receive one; receivers, which can receive a signal, but can't send one; and transceivers, which can do both. You may wonder why everything isn't a transceiver, as it's the most flexible device. The answer is that it's more complex and more expensive to make a transceiver than it is to make either of the other two. In a transceiver, you have to make sure the receiver is not receiving its transmitter's transmission, or they'll interfere with each other and not listen to any other device. When you buy a transceiver that does this for you, you pay for the convenience. For many applications, it's cheaper to just use a transmitter-receiver pair, and handle any errors by just transmitting the message many times until the receiver gets the message. That's how TV remote controls work, for example. It makes the components much cheaper.

It's increasingly common, especially in radio, to just make everything a transceiver, and incorporate a microcontroller to manage the transmitter-receiver juggling. All Bluetooth, ZigBee, and Wi-Fi radios work this way. However, it's still possible to get transmitter-receiver pair radios, and they are still considerably cheaper than their transceiver counterparts. The first two projects in this chapter use transmitter-receiver pairs.

Keep in mind the distinction between transmitter-receiver pairs and transceivers when you plan your projects, and when you shop. Start by asking yourself whether the communication in your project has to be two-way, or whether it can be one-way only. If it's one-way, ask yourself what happens if the communication fails. Can the receiver operate without asking for clarification? Can the problem be solved by transmitting repeatedly until the message is received? If the answer is yes, then you might be able to use a transmitter-receiver pair and save some money.

How Infrared Works

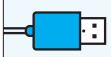
IR communication works by pulsing an IR LED at a set data rate and receiving the pulses using an IR photodiode. It's simply serial communication transmitted using infrared light. Since there are many everyday sources of IR light (the sun, incandescent light bulbs, any heat source), it's necessary to differentiate the IR data signal from other IR energy. To do this, the serial output is sent to an oscillator before it's sent to the output LED. The wave created by the oscillator, called a carrier wave, is a regular pulse that's modulated by the pulses of the data signal. The receiver picks up all IR light, but filters out anything that's not vibrating at the carrier frequency. Then it filters out the carrier frequency so that all that's left is the data signal. This method allows you to transmit data using infrared light without getting interference from other IR light sources, unless they happen to be oscillating at the same frequency as your carrier wave.

The directional nature of infrared makes it more limited than radio, but cheaper than radio, and requires less power. As radios get cheaper, more power-efficient, and more robust, it's less common to see an IR port on a computer or PDA, but it's still both cost-effective and power-efficient for line-of-sight remote control applications.

Data protocols for the IR remote controls of most home electronics vary from manufacturer to manufacturer. To decode them, you need to know both the carrier frequency and the message structure. Most commercial IR remote control devices operate using a carrier wave between 38 and 40 kHz. The frequency of the carrier wave limits the rate at which you can send data on that wave, so IR transmission is usually done at a low data rate, typically between 500 and 2000 bits per second. It's not great for high-bandwidth data transmission, but if you're only sending the values of a few pushbuttons on a remote, it's acceptable. Unlike the serial protocols you've seen so far in this book, IR protocols do not all use an 8-bit data format. For example, Sony's Control-S protocol has three formats: a 12-bit, a 15-bit, and a 20-bit format. Philips' RC5 format, common to many remotes, uses a 14-bit format.

If you have to send or receive remote control signals, you'll save a lot of time by looking for a specialty IR modulator chip to do the job, rather than trying to recreate the protocol yourself. Fortunately, there are many helpful sites on the web to explain the various protocols. Reynolds Electronics (www.rentron.com) has many helpful tutorials, and sells a number of useful IR modulators and demodulators. EPanorama has a number of useful links describing many of the more common IR protocols at www.epanorama.net/links/irremote.html.

If you're building both the transmitter and receiver, your job is fairly straightforward. You just need an oscillator through which you can pass your serial data to an infrared LED, and a receiver that listens for the carrier wave and demodulates the data signal. It's possible to build your own IR modulator using a 555 timer IC, but there are a number of inexpensive modules you can buy to modulate or demodulate an IR signal, as shown in this next project. X



Making Infrared Visible

Even though you can't see infrared light, cameras can. If you're not sure whether your IR LED is working, one quick way to check is to point the LED at a camera and look at the resulting image. You'll see the LED light up. Here's a look at the IR LED in a home remote control, viewed through a webcam attached to a personal computer. You can even see this in the LCD viewfinder of a digital camera. If you try this with your IR LED, you may need to turn the lights down to see this effect.

Having a camera at hand is useful when troubleshooting IR projects.

Project 8

Infrared Transmitter-Receiver Pair

This example uses custom IR transmitter and receiver ICs (integrated circuits) from Reynolds Electronics to establish a one-way link between transmitter and receiver. This transmitter-receiver pair can operate up to 19200 bits per second, much faster than normal household remote controls.

MATERIALS

- **1 solderless breadboard** such as Digi-Key (www.digikey.com) part number 438-1045-ND, or Jameco (www.jameco.com) part number 20601
- **1 Arduino module** or other microcontroller
- **1 USB-to-TTL serial adaptor** SparkFun's BOB-00718 from Chapter 2 will do the job (www.sparkfun.com). If you use a USB-to-RS-232 adaptor such as a Keyspan or Iogear dongle, refer to Chapter 2 for the schematics to convert RS-232 to TTL serial.
- **1 IR transmitter IC** Reynolds (www.rentron.com) part number TX-IRHS
- **1 20MHz ceramic resonator with internal caps** for use with the IR transmitter IC, from Reynolds
- **1 infrared LED** Reynolds part number TSAL7200, or any other IR LED will do the job.
- **1 high-speed IR detector module** Reynolds (www.rentron.com) part number TSOP7000
- **1 100-ohm resistor**
- **1 220-ohm resistor**
- **1 potentiometer**

In this example, you'll connect the IR transmitter and a potentiometer to your microcontroller. The receiver will connect to your personal computer through a USB-to-serial adaptor. The microcontroller will continually send the potentiometer's value to the receiving computer.

You could also build this project with two microcontrollers, of course, but it's good practice to test the modules on a PC first so that can you troubleshoot the circuit and test the range of your IR transmitter and receiver. In testing, I got about 12 feet (nearly 4 meters) of range with this system.

There are two circuits for this project: the transmitter, which is connected to a microcontroller, and the receiver, which is connected to your computer via a USB-to-serial adaptor.

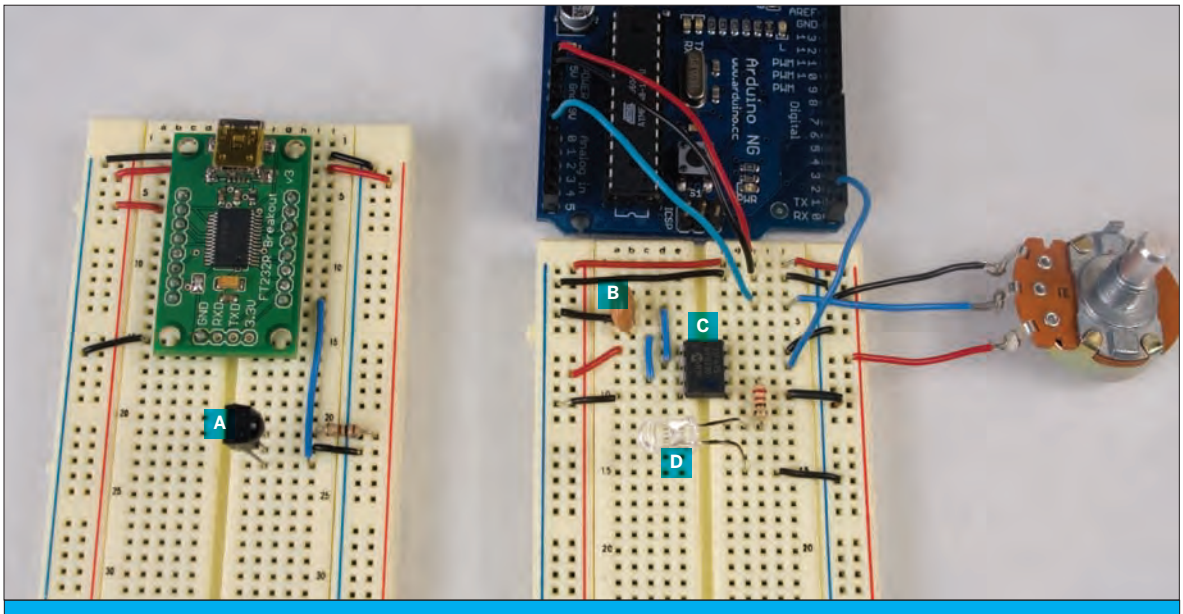
The transmitter's connections:

1. Voltage: to 5V
2. Oscillator 1: to ceramic resonator pin 1
3. Oscillator 2: to ceramic resonator pin 3
4. Duty cycle select: to ground. This sets the duty cycle of the carrier wave.
5. Data inversion select: to ground. This setting specifies whether the transmitter sends data as true (logic 0 = 0V, logic 1 = 5V), or inverted (logic 0 = 5V, logic 1 = 0V).
6. Data out: sends data out to IR LED
7. Data in: to microcontroller TX
8. Ground: to ground

The receiver's connections:

1. Data out: to the USB-to-serial adaptor's receive (RX) line
2. Ground: to ground
3. Voltage: to 5V power through a 100-ohm resistor

Figures 6-1 and 6-2 show the transmitter and receiver.



A. TSOP7000 receiver B. 20MHz resonator C. TX-IRHS transmitter D. IR LED

Figure 6-1
At left, a TSOP7000 IR receiver connected to a USB-to-serial adaptor. At right, a TX-IRHS infrared transmitter connected to a microcontroller. A potentiometer is attached to an analog input of the microcontroller. The microcontroller sends the value of the pot to the serial adaptor via IR communication. You can see the wiring diagrams for this on the opposite page.

Try It Once you've got the circuit connected, transmission is very straightforward. Here's a program that transmits the value of a potentiometer:

```
/*
    IR transmit example
    Language: Wiring/Arduino
*/
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);
}
```

Once you've got this code running on your microcontroller, connect the IR receiver circuit to your computer. Then open your serial terminal program and connect to the circuit at 19200 bits per second. You should see the value of the potentiometer printed in the serial terminal window like so:

```
127
128
128
129
130
129
```

▸▸ As you can see, serial communication over infrared isn't that different than serial communication over a wire. You can send data in only one direction, so you can't use a handshaking protocol, and you're limited to the data rate of your transmitter-receiver pair. Otherwise, you won't have to make any major code changes to use IR.

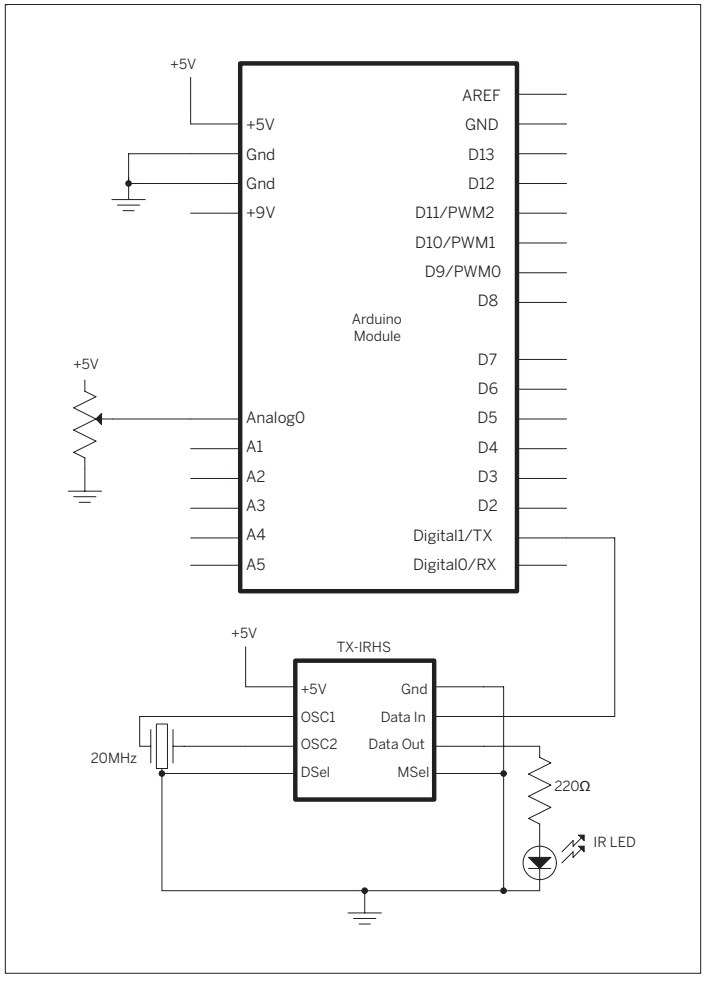
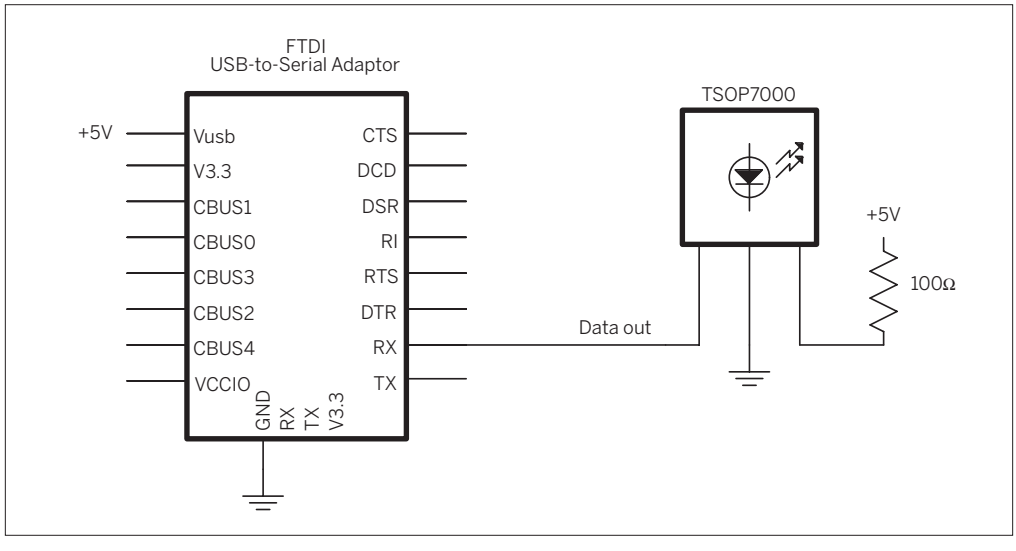


Figure 6-2
Top: the transmitter.
Bottom: the receiver.



How Radio Works

Radio relies on the electrical property called induction. Any time you vary the electrical current in a wire, you generate a corresponding magnetic field that emanates from the wire. This changing magnetic field induces an electrical current in any other wires in the field. The frequency of the magnetic field is the same as the frequency of the current in the original wire. This means that if you want to send a signal without a wire, you can generate a changing current in one wire at a given frequency and attach a circuit to the second wire to detect current changes at that frequency. That's how radio works.

The distance that you can transmit a radio signal depends on the signal strength, the sensitivity of the receiver, the nature of the antennas, and any obstacles that block the signal. The stronger the original current and the more sensitive the receiver, the farther apart the sender and receiver can be. The two wires act as antennas. Any conductor can be an antenna, but some work better than others. The length and shape of the antenna and the frequency of the signal all affect transmission. Antenna design is a whole field of study on its own, so I can't do it justice here, but a rough rule of thumb for a straight wire antenna is as follows:

Antenna length
5,616 in. / frequency in MHz

Antenna length
14,266.06 cm. / frequency MHz

For more information, consult the technical specifications for the specific radios you're using. Instructions on making a good antenna are common in a radio's documentation.

Radio Transmission: Digital and Analog

As with everything else in the microcontroller world, it's important to distinguish between digital and analog radio transmission. Analog radios simply take an analog electrical signal such as an audio signal, and superimpose it on the radio frequency in order to transmit it. The radio frequency acts as a carrier wave, carrying the audio signal. Digital radios superimpose digital signals on the carrier wave, so there must be a digital device on either end to encode or decode those signals. In other words, digital radios are basically modems, converting digital data to radio signals, and radio signals back into digital data.

Radio Interference

Though radio is omnidirectional, it can be blocked by obstacles, particularly metal ones. A large metal sheet, for example, will reflect a radio signal rather than allowing it to pass through. This principle is used not only in designing antennas, but also in designing shields. If you've ever cut open a computer cable and encountered a thin piece of foil wrapped around the inside wires, you've encountered a shield. Shields are used to prevent random radio signals from interfering with the data being transmitted down a wire. A shield doesn't have to be a solid sheet of metal, though. A mesh of conductive metal will block a radio signal as well, if the grid of the mesh is small enough. The effectiveness of a given mesh depends on the frequency it's designed to block. In fact, it's possible to block radio signals from a whole space by surrounding the space with an appropriate shield and grounding the shield. You'll hear this referred to as making a Faraday cage. A Faraday cage is just an enclosure that's shielded to be radio-free. The effect is named after the physicist Michael Faraday, who first demonstrated and documented it.

Sometimes radio transmission is blocked by unintentional shields. If you're having trouble getting radio signals through, look for metal that might be shielding the signal. Transmitting from inside a car can sometimes be tricky because the car body acts as a Faraday cage. Putting the antenna on the outside of the car improves reception. This is true for just about every radio housing.

All kinds of electrical devices emit radio waves as side effects of their operation. Any alternating current can generate a radio signal, even the AC that powers your home or office. This is why you get a hum when you lay speaker wires in parallel with a power cord. The AC signal is inducing a current in the speaker wires, and the speakers are reproducing the changes in current as sound. Likewise, it's why you may have trouble operating a wireless data network near a microwave oven. Wi-Fi operates at frequencies in the gigahertz range. That range is commonly called the microwave range, because the wavelength of those signals is only a few micrometers long. Microwave ovens use those same frequencies, transmitted at very high power, to cook food. Some of that energy leaks from the oven at low power, which is why you get all kinds of radio noise in the gigahertz range around a microwave.

Motors and generators are especially insidious sources of radio noise. A motor also operates by induction; specifically, by spinning a pair of magnets around a shaft in the center of a coil of wire. By putting a current in the wire, you generate a magnetic field, and that attracts or repulses the magnets, causing them to spin. Likewise, by using mechanical force to spin the magnets, you generate a current in the wire. So a motor or a generator is essentially a little radio, generating noise at whatever frequency it's rotating.

Because there are so many sources of radio noise due to the ubiquitous use of alternating currents, there are many ways for a radio signal to be interfered with. It's important to keep these possible sources of noise in mind when you begin to work with radio devices. Knowledge of common interference sources, and knowing how to shield against them, is a valuable tool in radio troubleshooting.

Multiplexing and Protocols

When you're transmitting data via radio, anyone with a compatible receiver can receive your signal. There's no wire to contain the signal, so if two transmitters are sending at the same time, they will interfere with each other. This is the biggest weakness of radio: a given receiver has no way to know who sent the signal it's receiving. In contrast, consider a wired serial connection: you can be reasonably sure when you receive an electrical pulse on a serial cable that it came from the device on the other end of the wire. You have no such guarantee with radio. It's as if you were blindfolded at a cocktail party and everyone else there had the same voice. The only way you'd have of knowing who was saying what is if everyone were polite about not interrupting each other, clear about beginning and ending their sentences, and identifying themselves when they speak. In other words, it's all about protocols.

The first thing everyone at that cocktail party would have to do is to agree on who speaks when. That way they could all share your attention by dividing up the time they get. Sharing in radio communication is called multiplexing, and this form of sharing is called time division multiplexing. Each transmitter gets a given time slot in which to transmit.

Of course, it depends on all the transmitters being in synch. When they're not, time division multiplexing can still work reasonably well if all the transmitters speak much less than they listen (remember the first rule of love and networking from Chapter 1: listen more than you speak). If a given transmitter is transmitting for only a few milliseconds in each second, and if there's a limited number of transmitters, the chance that any two messages will overlap, or collide, is relatively low. This guideline, combined with a request for clarification from the receiver (rule number three), can ensure reasonably good RF communication.

Back to the cocktail party. If every speaker spoke in a different tone, you could distinguish them by their tones. In radio terms, this is called frequency division multiplexing. It means that the receiver has to be able to receive on several frequencies simultaneously. But if there's a coordinator handing out frequencies to each pair of transmitters and receivers, it's reasonably effective.

Various combinations of time and frequency division multiplexing are used in every digital radio transmission system. The good news is that most of the time you never have to think about it, because it's handled for you by many of the radios on the market today, including the ones you'll see shortly.

Multiplexing helps transmission by arranging for transmitters to take turns and to distinguish themselves based on frequency, but it doesn't concern itself with the content of what's being said. This is where data protocols come in. Just as you saw how data protocols made wired networking possible, you'll see them come into play here as well. It's common to use a data protocol on top of using multiplexing methods, to make sure that the message is clear. For example, Bluetooth, ZigBee, and Wi-Fi are nothing more than data networking protocols layered on top of a radio signal. All three of them could just as easily be implemented on a wired network (and in a sense, Wi-Fi is: it uses the same TCP/IP layer that Ethernet uses). The principles of these protocols are no different than those of wired networks, which makes it possible to understand wireless data transmission even if you're not a radio engineer. Remember the principles and troubleshooting methods you used when dealing with wired networks, because you'll use them again in wireless projects. The methods mentioned here are just new tools in your troubleshooting toolkit. You'll need them in the projects that follow.
X

Project 9

Radio Transmitter-Receiver Pair

When your project is simple enough to work with one-way communication, but you need the omnidirectionality that radio affords, RF transmitter-receiver pairs are the way to go. There are several models on the market, from companies like Abacom (www.abacomdirect.com), Reynolds (www.rentron.com), Glolab (www.glolab.com), and others. Most of them are very simple to interface with a microcontroller, requiring nothing more than power, ground, and a connection to the serial I/O lines of the controller. Many of them even come with built-in antennas. This example uses a TX/RX pair made by Laipac, sold by SparkFun. The transmitter and receiver both connect directly to a microcontroller's serial lines as described above, and can operate at voltages in the 3.3V to 5V range.

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 USB-to-TTL serial adaptor** SparkFun's BOB-00718 from Chapter 2 will do the job. If you use a USB-to-RS-232 adaptor such as a Keyspan or Iogear dongle, refer to Chapter 2 for the schematics to convert RS-232 to 5V TTL serial.
- **1 RF transmitter-receiver pair** Available from SparkFun as part number WRL-07813, but similar models from the other retailers will work as well.
- **1 10KΩ potentiometer**
- **2 0.1µF capacitors**

In this example, you'll connect the RF transmitter and a potentiometer to your microcontroller. The receiver will connect to your personal computer through a USB-to-serial adaptor. The microcontroller will continually send the potentiometer's value to the receiving computer.

There are two circuits for this project: the transmitter, which is connected to a microcontroller, and the receiver, which is connected to your computer via a USB-to-serial adaptor. The connections are as follows:

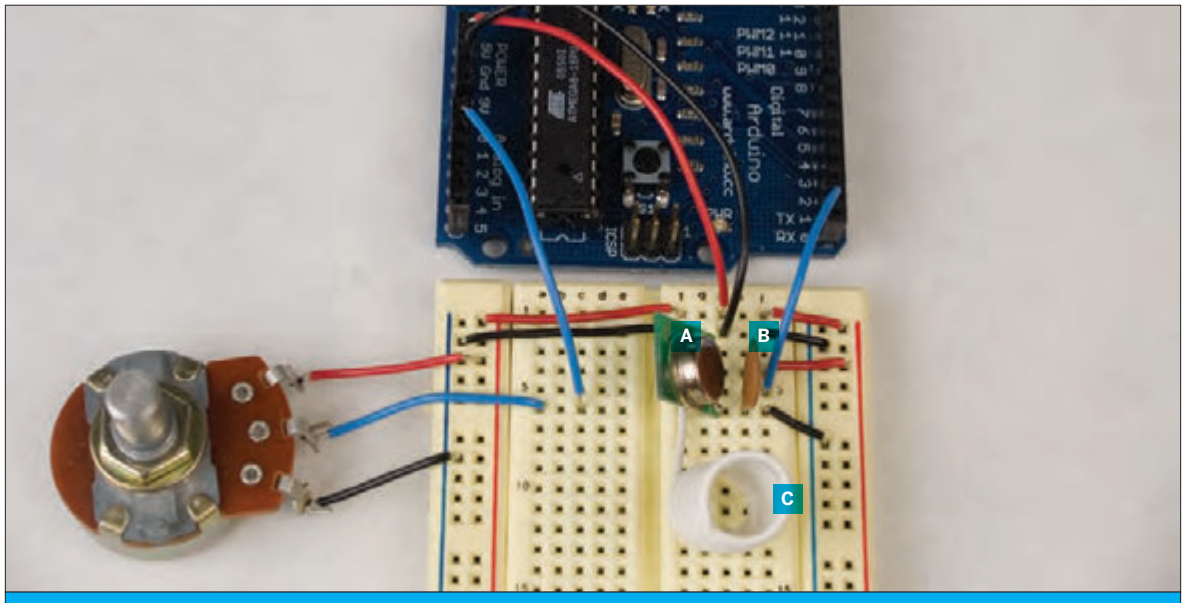
Transmitter

1. Ground
2. Data in: to microcontroller transmit pin
3. Voltage: to 5V
4. Antenna: to a 30cm piece of wire, acting as an antenna

Receiver

1. Ground
2. Data out: to USB-to-serial RX pin
3. Linear out: not connected
4. Voltage: to +5V
5. Voltage: to +5V. Be sure to put a 0.1µF capacitor across voltage and ground to decouple the radio's power supply.
6. Ground
7. Ground
8. Antenna: to a 30cm piece of wire, which acts as an antenna.

Figure 6-3 shows the transmitter, and Figure 6-4 shows the receiver.



A. RF transmitter B. 0.1 µF capacitor C. Antenna (30cm wire)

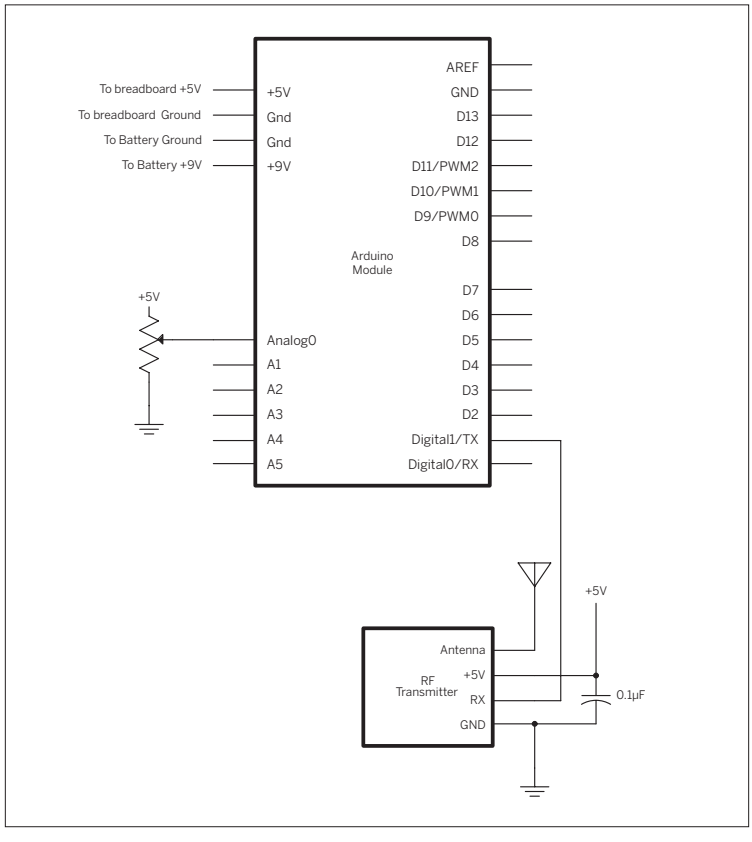
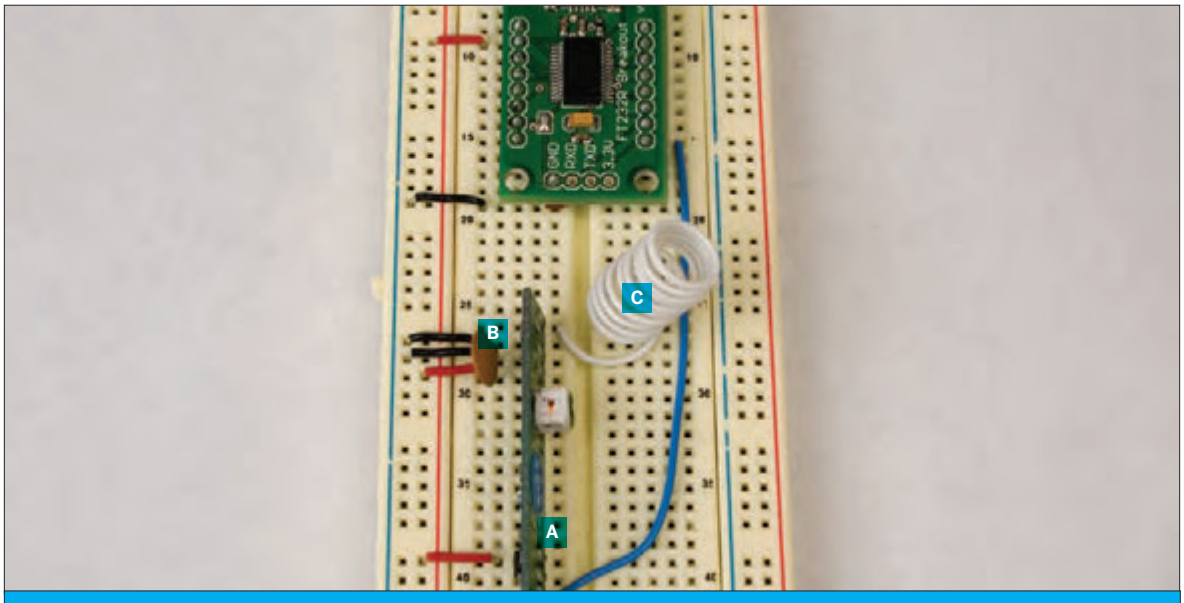


Figure 6-3
RF transmitter connected to a microcontroller. A potentiometer is connected to the microcontroller's analog input. The coil of wire is the antenna.



A. RF receiver B. 0.1 µF capactior C. Antenna (30cm wire)

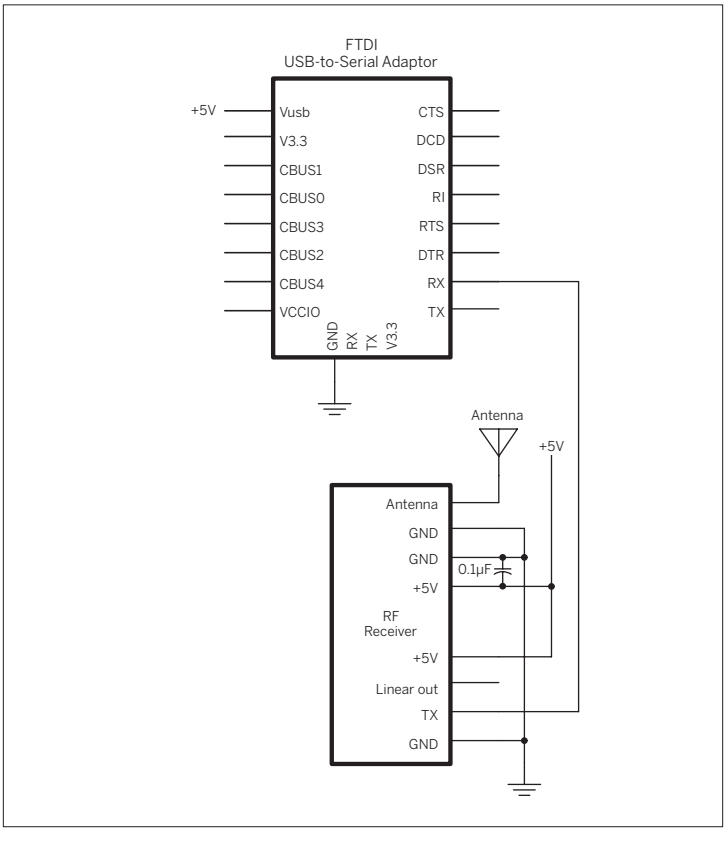


Figure 6-4
RF receiver connected to a personal computer via a USB-to-serial adaptor. The coil of wire is the antenna.

Try It Once you've got the circuit connected, program the microcontroller with the following code, which reads the analog input and sends its value out as an ASCII-encoded string:

```
/*
 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);
}
```

When you've got the microcontroller programmed, open the serial port that the receiver is connected to using your serial terminal program, at 2400 bits per second (thus far you've operated at 9600 or 19200 bps. For this example, just set your serial terminal program to communicate at 2400 bps instead of 9600). You might see a string of garbage characters. That's because the transmitter started sending before the receiver was activated. Reset the microcontroller while the serial port is open. After a few seconds, it should start sending again, and you should see a string of numbers representing the potentiometer's value, like this:

```
127
128
128
129
130
129
```

You may also get garbage characters on the screen, even when the transmitter is turned off. This is because the receiver's picking up random noise in its frequency range. This can be generated by a wide range of sources. All electrical devices emit some RF waves, so if your receiver is sitting beside an LCD or CRT display, for example, it could be picking up noise from that. In the absence of a signal from the transmitter, the receiver will display anything it gets. This is one of the downsides of working with simple transmitter-receiver pairs like this: you need to filter out the garbage in your program. More advanced transceivers like the ones in the next example will do some or all of that filtering for you.

One simple way to filter out the noise is to limit your transmission to a definite range of values. If you transmit only in ASCII-encoded strings separated by commas, linefeeds, or return characters, you can ignore any bytes received that don't fit within those values. Furthermore, if you send the data in a particular format every time, your receiving program can ignore any strings it receives that don't match the pattern. The bytes you're receiving from the example program are always in the ASCII numeral range ("0" through "9", or ASCII values 48 through 57), or a linefeed and carriage return (ASCII 10 and 13). In addition, the strings are never more than four digits long, because the analog input value never exceeds 1023. So you can test whether the bytes match the accepted values, and whether the string is of the appropriate length.

Tune In Here's a Processing program that does this. Close your serial terminal and try this Processing program instead. Make sure to change the serial port number in this program to what-ever port you're using. All the interesting work is done in the serialEvent() method:

Although the program is written in Processing, the same algorithm can work in other programming environments.

```
/*
  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 will tell what went wrong

  // read the serial buffer:
  String myString = myPort.readStringUntil('\n');

  // make sure you have a valid string:
```

Continued from opposite page.

```
  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);
    }
  }
}
```

Now that you've seen the basics of sending serial data in one direction over a radio link, the next example will demonstrate how to send it in two directions, using a pair of RF transceivers. Even though the transceivers in the next example incorporate their own error checking, you might want to keep this checking method in mind, in case you need it.

Radio Transceivers

In many cases, one-way communication isn't enough. For example, you might have noticed in the previous example that sometimes the message doesn't get through, even when you've got the circuitry and the code fully working. In that case, you might want the PC to be able to query the microcontroller occasionally to see the state of the inputs. Or perhaps you're making an application in which there's input and output on both sides. In that case, transceivers are essential.

There are many different kinds of data transceivers on the market. Some connect directly to the serial I/O of the microcontroller and send the data as is. Some, like the Bluetooth module you saw in Chapter 2, add an additional protocol layer on top of the data communication, so you have to be able to implement that protocol on both sender and receiver. The cost of transceivers varies widely.

Until recently, most digital radio transceivers on the market implemented only the most basic serial communications protocol. For example, the RTF-DATA-SAW transceivers from Abacom do a good job of sending and receiving serial information. They connect directly to the serial transmit and receive pins of your microcontroller. Any serial data that you send out the transmit line goes directly out as a radio signal. Any pulses received by the transceiver are sent into your microcontroller's receive line. The benefit is that you don't have to learn any serial protocol — you can send data in any form you want. The cost is that you have to manage the whole conversation yourself. If the receiving transceiver misses a bit of data, you'll get a garbled message, just like you did with the transmitter-receiver pair in the preceding project. Furthermore, any radio device in the same frequency range can affect the quality of your reception. As long as you're working with just two radios and no interference, transceivers like the RTF-DATA-SAW do a fine job. There are other companies on the market who sell similar transceivers, including Linx Technologies (www.linxtechnologies.com) and Low Power Radio Solutions (www.lprs.co.uk).

There are an increasing number of cheap transceivers on the market that implement networking protocols, handling the conversation management for you. The Bluetooth modem in Chapter 2 ignored signals from other radios that it wasn't associated with, and took care of error checking for you. The XBee radios you'll use in the next project will do the same, and much more, as you'll see in Chapter 7. These particular transceivers are in the same general price range as the plain serial transceivers mentioned earlier. They require you to learn a bit more in terms of networking protocols, but the benefits you gain make them well worth that minor cost.

There's one other difference between the serial transceivers and the networked ones: the networked modules tend to operate at much higher speeds, both in terms of transmission frequency and serial data rate. For example, the Abacom modules mentioned previously operate at 315 MHz and a maximum serial data rate of 9600 bits per second. The XBee modules in the following project operate at 2.4Ghz and up to 115,200 bits per second. Hence, the XBee radios can send a message at nearly 100 times the speed. Even if you're sending only a few bytes per second, this means that your transceiver can spend more time listening and less time speaking, thus reducing the chance that it'll miss a given message from another transceiver.

Project 10

Duplex Radio Transmission

In this example, you'll connect an RF transceiver and a potentiometer to the microcontroller. Each microcontroller will send a signal to the other when its potentiometer changes by more than ten points. When either one receives a message, it will light up an LED to indicate that it got a message. Each device also has an LED for local feedback as well.

MATERIALS

- **2 solderless breadboards** such Digi-Key part number 438-1045-ND or Jameco part number 20601
- **1 USB-to-TTL serial adaptor** SparkFun's PCB-BOB-00718 from Chapter 2 will do the job. If you use a USB-to-RS-232 adaptor such as a Keyspan or Iogear dongle, refer to Chapter 2 for the schematics to convert RS-232 to 5V TTL serial.
- **2 Arduino modules** or other microcontrollers
- **2 Maxstream XBee OEM RF modules** available from www.maxstream.net or www.gridconnect.com, part number GC-WLM-XB24-A
- **2 XBee breakout boards** or 2 XBee Arduino shields

For the breakout boards:

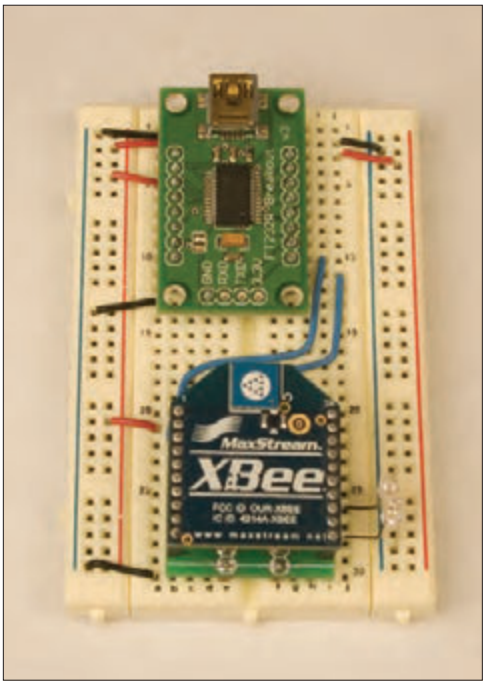
- **2 2mm breakout boards** The XBee modules listed here have pins spaced 2mm apart. To use them on a breadboard, you'll need a breakout board that shifts the spacing to 0.1 inches. You could solder wires on to every pin, or you could make or purchase a printed circuit board that shifts the pins. SparkFun's Breakout Board for XBee Module (part number BOB-08276) will do the trick.
- **4 rows of 0.1 inch header pins** available from most electronics retailers
- **4 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 also offers header rows for the XBee (part number PRT-08272).
- **2 potentiometers**
- **2 1µF capacitors**
- **2 10µF capacitors**
- **6 LEDs**

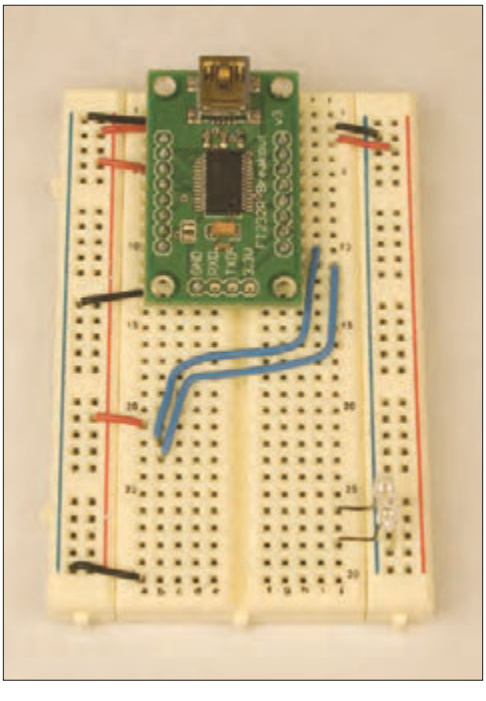
The RF transceivers used in this project implement the 802.15.4 wireless networking protocol on which ZigBee is based. In this example, you won't actually use any of the benefits of ZigBee, and few of the 802.15.4 benefits. 802.15.4 and ZigBee are designed to allow many different objects to communicate in a flexible networking scheme. Each radio has an address, and every time it sends a message, it has to specify the address to send to. It can also send a broadcast message, addressed to every other radio in range. You'll see more of that in Chapter 7. For now, you'll give each of your two radios the other's address, so that they can pass messages back and forth.

As you may have discovered with the previous project, there are many things that can go wrong with wireless transmission, and as radio transmissions are not detectable without a working radio, it can be difficult to troubleshoot. Because of that, you're going to build this project up in stages. First you'll communicate with the radio module itself serially, in order to set its local address and destination address. Then you'll write a program for the microcontroller to make it send messages when the potentiometer changes, and listen for the message to come through on a second radio attached to your personal computer. Finally, you'll make two microcontrollers talk to each other using the radios.

Step 1: Configuring the XBee Modules Serially

The RF transceivers used in this project implement the 802.15.4 wireless networking protocol on which ZigBee is based. In this example, you won't actually use any of the benefits of ZigBee, and few of the 802.15.4 benefits. 802.15.4 and ZigBee are designed to allow many different objects to communicate in a flexible networking scheme. Each radio has an address, and every time it sends a message, it has to specify the address to send to. It can also send a broadcast message, addressed to every other radio in range. You'll see more of that in Chapter 7. For now, you'll give each of your two radios the other's address, so





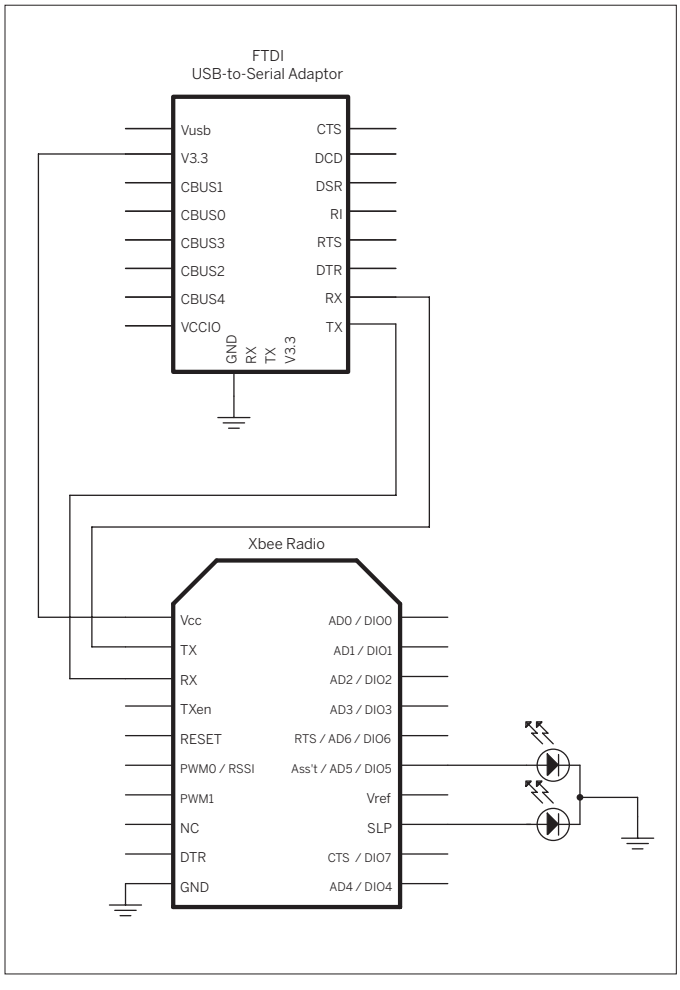


Figure 6-5
XBee radio attached to an FTDI USB-to-serial adaptor. The second photo shows the wiring underneath the XBee board. Note that the LEDs attached to the XBee have no resistors in series with them. The current out of the XBee's output pins is low enough to not burn up the LEDs.

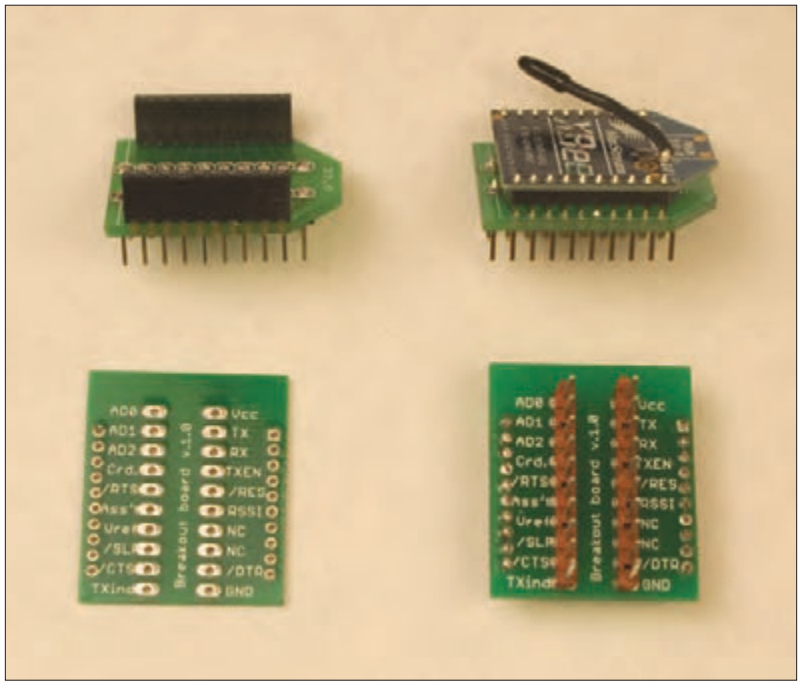


Figure 6-6
XBee printed circuit board, in various stages. *Bottom left:* bare board shown from the bottom. *Bottom right:* board with headers soldered to inner rows. *Top right:* finished board with no female sockets (radio is soldered directly to the board). *Top left:* finished board with female sockets.

Mounting the XBee Radios on a Breakout Board

The XBee radios have pins spaced 2mm apart, which is too narrow to fit on a breadboard. You can either solder wires to each pin to extend the legs, or you can mount the module on a breakout board. SparkFun has such a board: the Breakout Board for XBee Module (part number BOB-08276). The breakout board in Figure 6-6 is a custom-designed board developed before there was a commercially available solution. It's functionally identical to the SparkFun board. You'll need two breakout boards for this project, one for each radio.

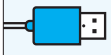
Once you've got the breakout board, solder headers to the inner rows. These will plug into your breadboard. Next, attach the radio to the breakout board. You can either solder it directly or use 2mm female headers to mount it on. If you solder the radio directly to the board, make sure that you leave space between the radio and the inner header pins, so they're not touching. If they are, you will short the radio out.

that they can pass messages back and forth.

As you may have discovered with the previous project, there are many things that can go wrong with wireless transmission, and as radio transmissions are not detectable without a working radio, it can be difficult to troubleshoot. Because of that, you're going to build this project up in stages. First you'll communicate with the radio module itself serially, in order to set its local address and destination address. Then you'll write a program for the microcontroller to make it send messages when the potentiometer changes, and listen for the message to come through on a second radio attached to your personal computer. Finally, you'll make two microcontrollers talk to each other using the radios.

Figure 6-5 shows an XBee module connected to a USB-to-serial adaptor. The USB adaptor draws power from the USB bus, and the radio draws power from the adaptor via its 3.3V voltage output.

Once you've got the XBee module's circuit built and



Arduino XBee Shield

There is an alternative to the breadboard circuit shown in Figure 6-5 for Arduino users. Libelium (www.libelium.com) and PCB Europe (pcb-europe.com) have teamed up to make an XBee shield for the Arduino module. The shield comes with an XBee radio, and connects to the Arduino's TX and RX pins. To connect the radio to the Arduino, set the XBEE/USB jumpers to the left, as shown in Figure 6-7. When you're programming the Arduino, you might want to remove the jumpers, so that the radio's serial communications don't interfere with the program upload.

You can also use your Arduino board as a USB-to-serial converter to configure the XBee radio on the shield. To do this, unplug your Arduino from its power source, then remove the microcontroller chip, as shown in Figure 6-8. Be careful not to bend the pins, so that you can put it back when you are done. Pay attention to the orientation of the microcontroller as well, as you have to put it back the same way. Once you've removed the chip, set the shield's XBEE/USB jumpers to the right, as shown in Figure 6-9, and put the shield on the board. Open a serial terminal connection to the Arduino board's serial port, and send commands as shown in "Step 1: Configuring the XBee Modules Serially." Once you've configured the radio, unplug the Arduino board, replace the microcontroller chip, set the XBEE/USB jumpers to the left, put the shield back on, and you're all set to program the Arduino to talk to the XBee radio.

These shields may change their form by the time this book is published, but even in their initial form, they are a convenient way to combine Arduinos and XBees.

powered, the LED on pin 13 should stay on steadily, and the LED on pin 15 will blink. The former is lit when the module is active (LED on), and the latter blinks whether the radio is associated with another radio (LED blinking) or not (LED on steadily). Make sure that your circuit is connected to your computer (USB port or serial port) and open the port in your favorite serial terminal program:

```
+++
```

Don't type the return key or any other key for at least one second afterward. It should respond like so:

```
OK
```

This step should look familiar to you from the Bluetooth modem you saw in Chapter 2. The XBee is using an AT-style command set like the Bluetooth modem did, and the +++ puts it into command mode. The one-second pause after this string is called the guard time. If you do nothing, the module will drop out of command mode after ten seconds, so if you're reading this while typing, you may need to enter another +++ string before the next stage.

Once you get the OK response, set the XBee's address. The 802.15.4 protocol uses either 16-bit or 64-bit long addresses, so there are two parts to the address, the high word and the low word (two or more bytes in computer memory used for a single value are sometimes referred to as a word). For this project, you'll use 16-bit addressing and therefore get to choose your own address. You'll need only the low word of the address do to this. Type:

```
ATMY1234\r
```

To confirm that you set it, type:

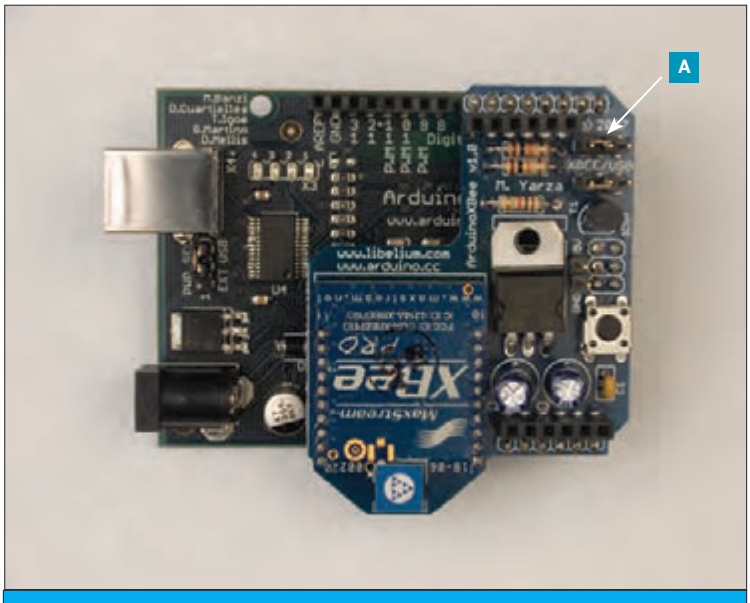
```
ATMY\r
```

The module should respond:

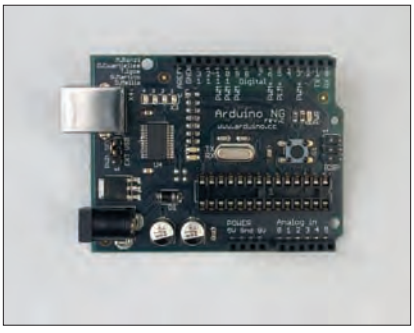
```
1234
```

You'll see that the responses from the XBee overwrite each other, because the XBee sends only a carriage return at the end of every message, not a linefeed.

X



A. XBEE/USB jumpers set to left



» at left, above

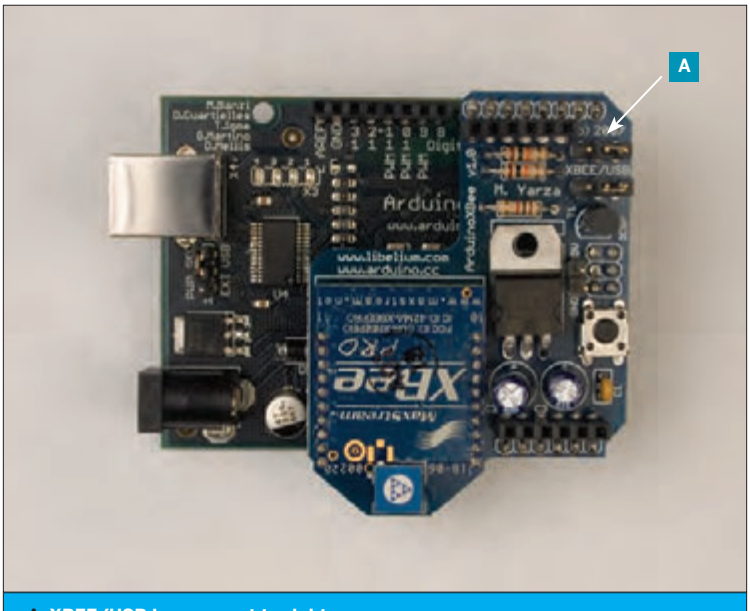
Figure 6-7

Arduino XBee shield. The XBEE/USB jumpers are set to the left to connect the radio's TX to the Arduino's RX, and vice versa.

» above

Figure 6-8

Arduino module with the microcontroller removed. In this configuration, the Arduino can act as a USB-to-serial converter.



A. XBEE/USB jumpers set to right

» at left, below

Figure 6-9

Arduino XBee shield. The XBEE/USB jumpers are set to the right so that the XBee's TX connects directly to the FTDI chip's RX, and vice versa. There is no microcontroller chip on the board underneath the shield.

An XBee Serial Terminal

Because the GNU screen program in Mac OS X, Unix, and Linux doesn't print newlines when the XBees send only a return character, it can be difficult to read the results.

Here's a Processing program that substitutes newlines for return characters when it prints the results onscreen. GNU screen users may find it useful in place of screen for communicating with XBee radios.

Figure 6-10 shows a screenshot of the XBee terminal program.

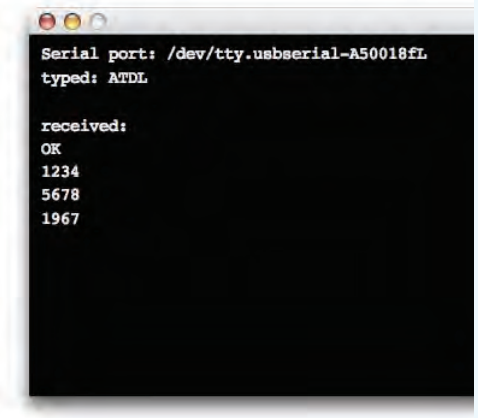


Figure 6-10
The XBee terminal sketch in action.

```
/*
  XBee terminal
  language: processing

  This program is a basic serial terminal program.
  It replaces newline characters from the keyboard
  with return characters. It's designed for use with
  Linux, Unix, and Mac OS X in combination with XBee radios,
  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() {
  // clear the screen:
  background(0);
  // print the name of the serial port:
  text("Serial port: " + portnum, 10, 20);
```

Continued from opposite page.

```
  // 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 OS X, if the user types return, a linefeed is returned. But
    // to communicate with the XBee, you want 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;
    // if any other key is typed, add it to outString:
  default:
    // 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();
```

Continued from previous page.

```
    }
  }
}
// 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);
}
```

Once you have the sketch working, you're ready to set the XBee's destination address. Make sure you're in command mode (+++), then type: ATDL\r

You'll likely get this: 0

The default destination address on these modules is 0. The destination address is two words long, so to see the high word, type:

```
ATDH\r
```

This pair of commands can also be used to set the destination address, like so:

```
ATDL5678\r
ATDH0\r
```

These radios also have a group, or Personal Area Network (PAN) ID. All radios with the same PAN ID can talk to each other, and ignore radios with a different PAN ID. Set the PAN ID for your radio like so:

```
ATID1111\r
```

The XBee will respond to this command, like all commands, with:

```
OK
```

Make sure to add the parameter WR after your last command, to write the parameters to the radio's memory. That way they'll remain the way you want them even after the radio is powered off. For example:

```
ATID1111,WR\r
```

Once you've configured one of your radios, quit the Processing sketch (or disconnect your serial terminal program) and unplug the board from your computer. Next, remove the XBee from the circuit, insert the second one, and configure it using the same procedure. Don't set a radio's destination address to the same value of its source address, or it will only talk to itself! You can use any 16-bit address for your radios. Here's a typical configuration for two radios that will talk to each other (don't forget to add the ,WR to the last command):

	ATMY	ATDL	ATDH	ATID
Radio 1	1234	5678	0	1111
Radio 2	5678	1234	0	1111

You can combine commands on the same line by separating them with commas. For example, to get both words of a module's source address, type this:

```
ATDL, DH\r
```

The module will respond with both words at once. Likewise, to set both destination words and then make the module write them to its memory so that it saves the address when it's turned off, type:

```
ATDL5678, DH0, WR\r
```

The module will respond to all three commands at once:

```
OK OK OK
```

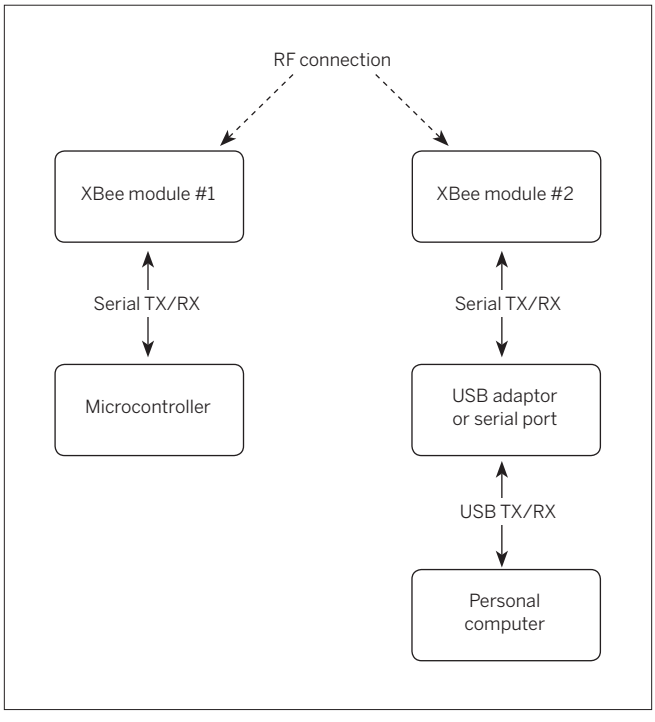
X

Step 2: Programming a Microcontroller to use the XBee Module

Okay! Now you're ready to get two modules to talk to each other. If you happen to have two serial ports, or two USB adaptors, you could duplicate the circuit shown previously and wire the second radio to the second serial port, then open a second serial terminal window to the second port and communicate between the two radios that way. But it's clearer to see what's going on if one of the radios is attached to another device, like a microcontroller. Figure 6-11 shows a diagram of what's connected to what in this step.

Figure 6-12 shows an XBee module attached to a regular Arduino using the XBee shield. Figure 6-13 shows an XBee attached to an Arduino mini along with the circuit diagram. Note the 3.3V regulator. The XBee's serial I/O connections are 5V tolerant, meaning that they can accept 5V data signals, even though the module operates at 3.3V, just like the XPort in Chapter 5. You need to power the module from 3.3 volts, however.

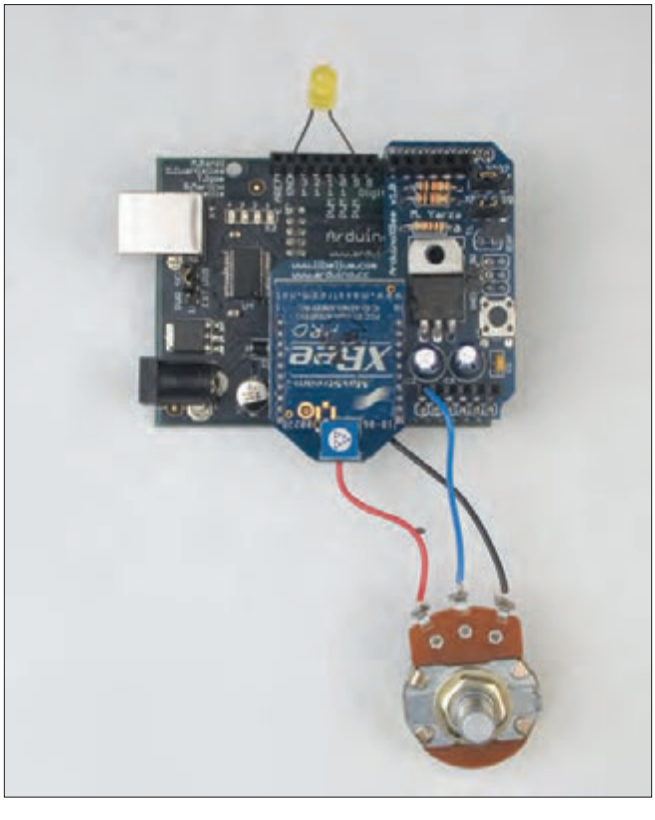
Once your module is connected, it's time to program the microcontroller to configure the XBee, then to send data through it. In this program, the microcontroller will configure the XBee's destination address on startup. Once that's done, it will watch for a switch to change from low to high, and send data across when the switch changes.



» at right, above

Figure 6-11

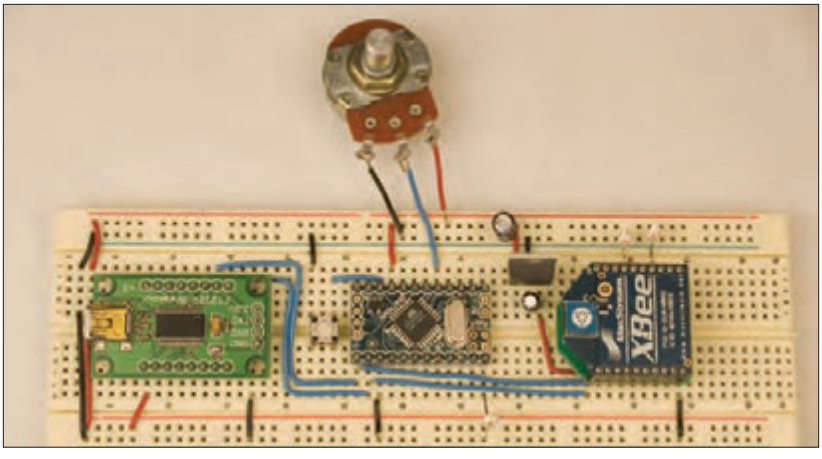
XBee #1 is connected to the microcontroller. XBee #2 is connected via USB or serial to the PC. This enables a wireless link between the PC and the microcontroller.



» at right, below

Figure 6-12

Arduino and XBee shield with potentiometer attached to analog pin 0, and LED attached to digital pin 9. This circuit is the same as the one shown in Figure 6-13, but without the TX and RX LEDs on digital pins 2 and 3.



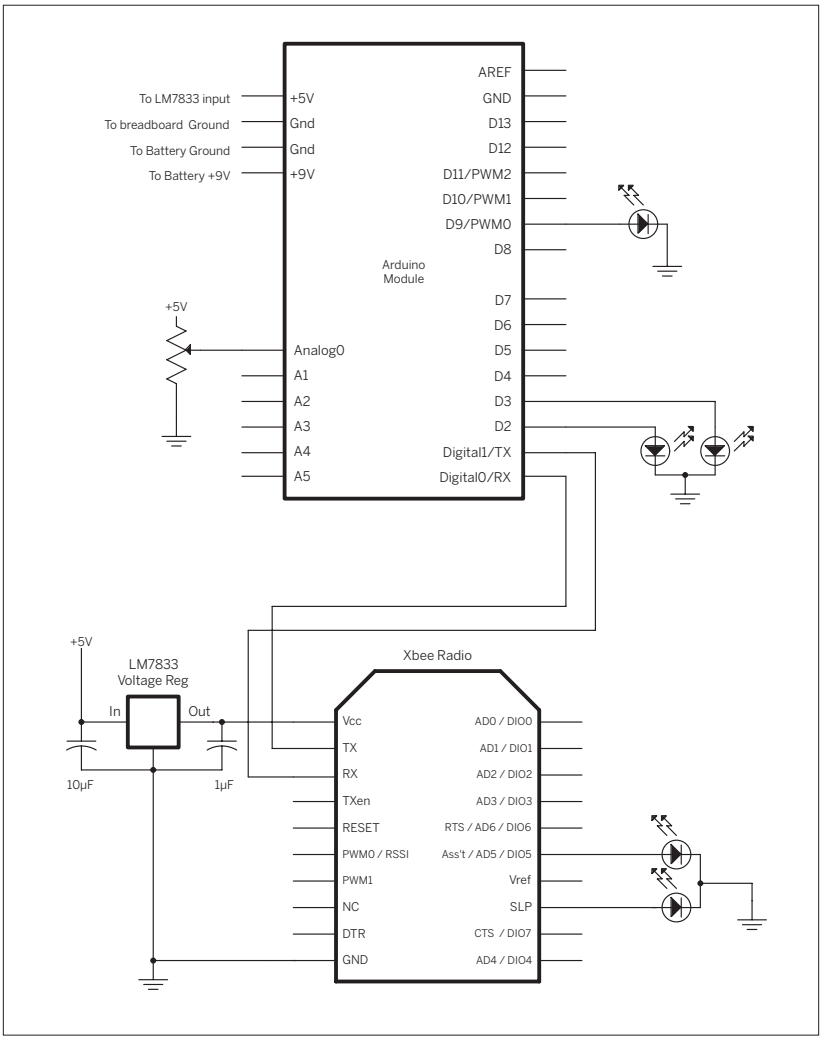


Figure 6-13
Top: XBee connected to an Arduino Mini. This Mini is using a SparkFun version of the USB-to-serial adaptor rather than the Arduino model. The two adaptors are functionally identical. *Bottom:* Circuit diagram for Arduino-Xbee connection.

Make It First, give the I/O pins names and set up some variables for tracking the change in the switch:

```
#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
```

Next, in the setup() method, configure serial transmission, set the modes on the I/O pins, and configure the XBee's destination address:

```
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 indicating that the main program's about to start:
  blink(3);
}
```

The XBee configuration, handled by the setDestination() method, looks just like what you did earlier, only now you're instructing the microcontroller to do it:

```
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 with 16-bit addressing. This radio's
  // destination should be the other radio's MY address and vice versa:
  Serial.print("ATDH0, DL1234\r");
  Serial.print("ATMY5678\r");   // set my address (16-bit addressing)

  // set the PAN ID. If you're in a place where many people
  // are using XBees, choose a unique PAN ID
  Serial.print("ATID1111\r");
  Serial.print("ATCN\r"); // go into data mode:
}
```

Change the destination address to the destination address of the radio you're attaching to your personal computer, not the one that's attached to your microcontroller.

The blink() method is just like ones you've seen previously in the book. It blinks an LED to indicate that setup is over:

```
// 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);
  }
}
```

The main loop handles incoming serial data, reads the potentiometer, and sends data out if there's a sufficient change in the potentiometer's reading:

```
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);
  }
}
```

There are two other methods called from the loop, handleSerial(), which listens for strings of ASCII numerals and converts them to bytes in order to set the brightness of the led on the PWM output, and readSensor(), which reads the potentiometer and checks to see whether the change on it is high enough to send the new value out via radio. Here are those methods:

```
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);
```

NOTE: You might need to disconnect the XBee's receive and transmit connections to the microcontroller while programming, if your microcontroller is programmed serially like the Arduino and Wiring modules are. The serial communications with the XBee can interfere with the serial communications with the programming computer. Once the microcontroller's programmed, you can re-connect the transmit and receive lines.

Continued from opposite page.

```
    // 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;
}
```

Notice that in the main loop, you're not using any AT commands. That's because the XBee goes back into data mode (called idle mode in the XBee user's guide) automatically when you issue the ATCN command in the setDestination() method.

Remember, in data mode, any bytes sent to an AT-style modem go through as is. The only exception to this rule is that if the string +++ is received, the modem switches to command mode. This behavior is the same as that of the Bluetooth module from Chapter 2, and as almost any device that implements an AT-style protocol. It's great, because it means that once you're in data mode, you can send data with no extra commands, letting the radio itself handle all the error corrections for you.

Once you've programmed the microcontroller, set the destination address on the computer's XBee to the address of the microcontroller's radio. (If you did this in the earlier step, you shouldn't need to do it again.) Then turn the potentiometer on the microcontroller. You should get a message like this in your serial terminal window:

120

The actual number will change as you turn the potentiometer. It will overwrite itself in the serial window, because you're not sending a newline character (unless you are using the serial terminal Processing sketch shown earlier). Congratulations! You've made your first wireless transceiver link. Keep turning the potentiometer until you're bored, then move on to step 3.

X

Step 3: Two-Way Wireless Communication Between Microcontrollers

This step is simple. All you have to do is to replace the computer in the previous step with a second microcontroller (connect it to your second XBee module as shown in Figure 6-12 or Figure 6-13). The program for both microcontrollers will be almost identical to each other; only the destination address of the XBee radio will be different. This program will both send and receive data over the modules. Turning the potentiometer causes it to send a number to the other microcontroller. When the microcontroller receives a number in the serial port, it uses it to set the brightness of an LED on pin 9.

First, connect the second XBee module to the second microcontroller using the circuit in Figure 6-13. It's same circuit you created in the previous step. Then program both microcontrollers with the previous program, making sure to set the destination addresses as noted in the program. Look in Appendix C for the program in its entirety.

When you've programmed both modules, power them both on and turn the potentiometer several times. As you turn the potentiometer, the LED on pin 9 of the other module should fade up and down. Now you've got the capability for duplex wireless communication between two microcontrollers. This opens up all kinds of possibilities for interaction.
X

Wireless and Mobile

Now that you're able to communicate wirelessly, you might want to make your microcontroller mobile as well. To do this, all you have to do is to power it from a 9-volt battery. If you're using an Arduino module or a Wiring module, you can do this by connecting a 9-volt battery to the power input terminals as shown in Figure 6-14 (and in the schematic shown earlier). If you're working with a different microcontroller and it's powered by a 5-volt voltage regulator, just connect the battery to the input terminals of the voltage regulator. It's a good idea to keep your microcontroller module connected to a power adaptor or USB power while programming and debugging, however. When a battery starts to weaken, your module will operate inconsistently, and that can make debugging impossible.

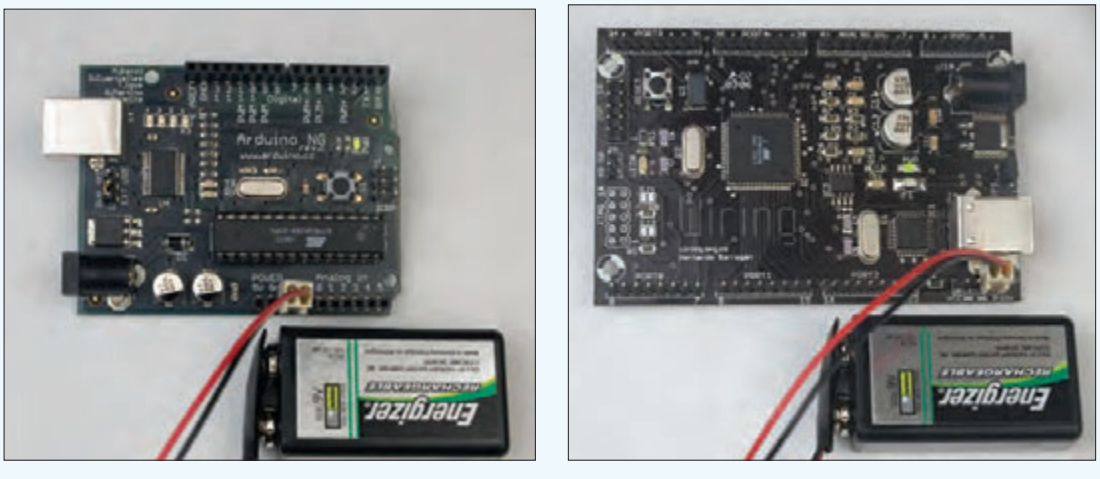


Figure 6-14
Left: Arduino module powered by a 9V battery. *Right:* Wiring module powered by 9V battery.

Project 11

Bluetooth Transceivers

In Chapter 2, you learned how to connect a microcontroller to your personal computer using a Bluetooth radio. This example shows you how to connect two microcontrollers using Bluetooth in a similar manner.

As mentioned in Chapter 2, Bluetooth was originally intended as a protocol for replacing the wire between two devices. As a result, it requires a tighter connection between devices than you saw in the preceding XBee project. In that project, a radio sent a signal out with no awareness of whether the receiver got the message, and could send to a different receiver just by changing the destination address. In contrast, Bluetooth radios must establish a connection with each other before sending data over a given channel, and must break that connection before starting a conversation with a different radio over that channel. The advantage of Bluetooth is that it's built into many commercial devices today, so it's a convenient way to connect microcontroller projects to personal computers, phones, and more. For all its complications, it offers reliable data transmission.

The modules used here, the BlueSMiRF radios from Sparkfun, use a radio from BlueRadios. The AT command set used here was defined by BlueRadios. Other Bluetooth modules from other manufacturers also use AT-style command sets, and they may execute similar functions, but their syntax is not the same. Unfortunately, Bluetooth radio manufacturers haven't set a standard AT syntax for their devices.

MATERIALS

- **2 solderless breadboards** such as Digi-Key part number 438-1045-ND, or Jameco part number 20601
- **1 USB-to-TTL serial adaptor** SparkFun's BOB-00718 from Chapter 2 will do the job. If you use a USB-to-RS-232 adaptor such as a Keyspan or Iogear dongle, refer to Chapter 2 for the schematics to convert RS-232 to 5V TTL serial.
- **2 Arduino modules** or other microcontrollers
- **2 BlueSMiRF Bluetooth modem modules** from SparkFun
- **2 potentiometers**
- **6 LEDs**

Step 1: Getting to Know the Commands

Because the Bluetooth connection process involves many steps, it's easiest to learn and understand it using a serial terminal program. Figure 6-15 shows the wiring to connect a BlueSMiRF radio to an FTDI USB-to-serial connector. If you're not using the FTDI connector, you can use the MAX3323 circuit from Chapter 2 (Figure 2-3). Build this circuit, then connect it to your computer and open a serial connection to it at 9600 bits per second using your serial terminal program.

Figure 6-16 shows the connection you're about to make. First, you'll open a serial terminal window to connect to the radio's serial interface, then you'll open a second serial terminal window to connect via your computer's Bluetooth radio to the BlueSMiRF's radio. Your computer's Bluetooth radio will show up as a second serial port on your computer, as it did after you established a pairing with it in Chapter 2 in Project #2, Wireless Monski Pong. If you didn't make that pairing, this would be a good time to go back and do it.

The BlueSMiRF radios use an AT-style command set for command and configuration, and have two modes — command mode and data mode — just like the XBee radios. When you first power up a BlueSMiRF and connect to its serial interface, it's in command mode. To see that it's alive, type: `AT\r`. It will respond:

```
\r\nOK\r\n
```

All of the radio's responses will be preceded and followed by a linefeed and carriage return as shown here. All of your input commands should be followed by a carriage return (press Enter or Return).

In order for one radio to connect to another, the second radio must be discoverable. In the BlueRadios syntax, a radio is discoverable when it's in Slave mode. The radio connecting to it is said to be in Master mode. In this step,

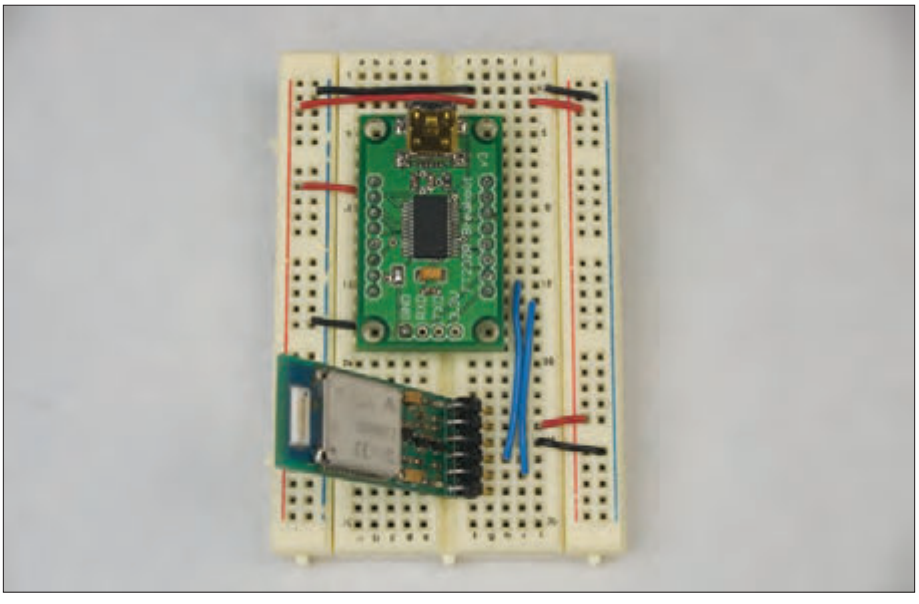


Figure 6-15
BlueSMiRF radio attached to a USB-to-serial adaptor.

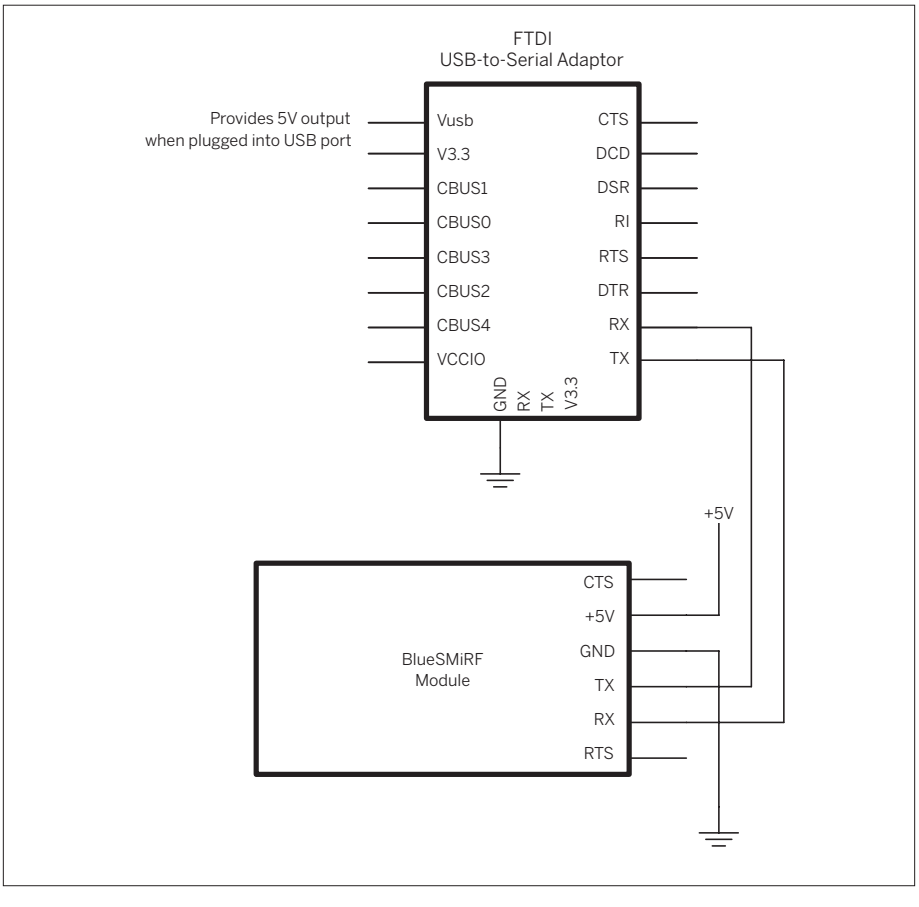
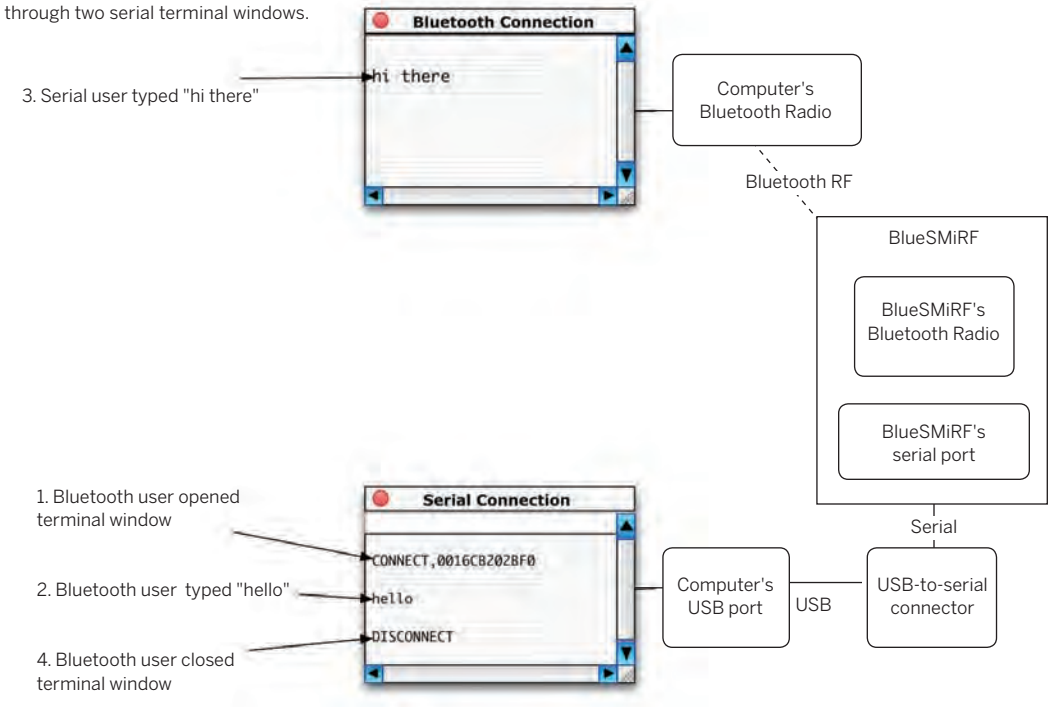


Figure 6-16
Bluetooth-to-serial connection through two serial terminal windows.



you'll learn a radio's Bluetooth address and check its connection status.

A series of status commands tell you about the radio's configuration. To learn the radio's Bluetooth address, type:

```
ATSI,1\r
```

It will respond with an address in hexadecimal notation, like this:

```
OK
1122334455AA
```

Write down this address, or copy it to a text document. You'll need it in a moment. Next, check its connection status by typing `ATSI,3\r`. It will respond like so:

```
OK
0,0
```

The first digit is telling you that the radio is in slave mode, and the second, that it's not connected. Now you can open a second serial terminal window, open the serial port that corresponds to the radio's Bluetooth connection (you established this number when you paired the radio with your computer in the Wireless Monski Pong project), and you'll be speaking via your computer's Bluetooth radio to the BlueSMiRF's radio. You'll get a message like this back in the serial terminal:

```
CONNECT, 1122334455AA
```

Following that, anything you type in the Bluetooth connection window shows up in the serial connection window, and vice versa. When you close the Bluetooth serial window, you'll get the following message in the serial window:

```
DISCONNECT
```

There are other status commands as well, but these ones are the ones that are most important to you at first.
X

Step 2: Connecting Two Bluetooth Radios

Now that you've got basics of connecting and disconnecting, it's time to connect to a microcontroller using Bluetooth. For this step, you'll connect via the same USB-to-serial connection, but instead of speaking to your computer's own radio, you'll connect to a radio attached to a microcontroller.

First, get the Bluetooth addresses for both of your radios. You already wrote down one. Replace it with the second radio in your serial-to-USB circuit and follow the same steps to get that radio's address as well.

Next, build the microcontroller circuit shown in Figure 6-17. Just like the XBee example, it's got a potentiometer attached to the analog pin so that you can send its values. There's also a connection to the BlueSMiRF's Clear-to-Send (CTS) pin. When the BlueSMiRF reads 5V on this pin, it stops sending data until the pin goes low again. You'll use it to stop the BlueSMiRF sending serial data to the microcontroller when you don't want it to.

NOTE: You'll probably have to remove your BlueSMiRF while programming the Arduino or Wiring boards, just as you've had to for all other serial devices.

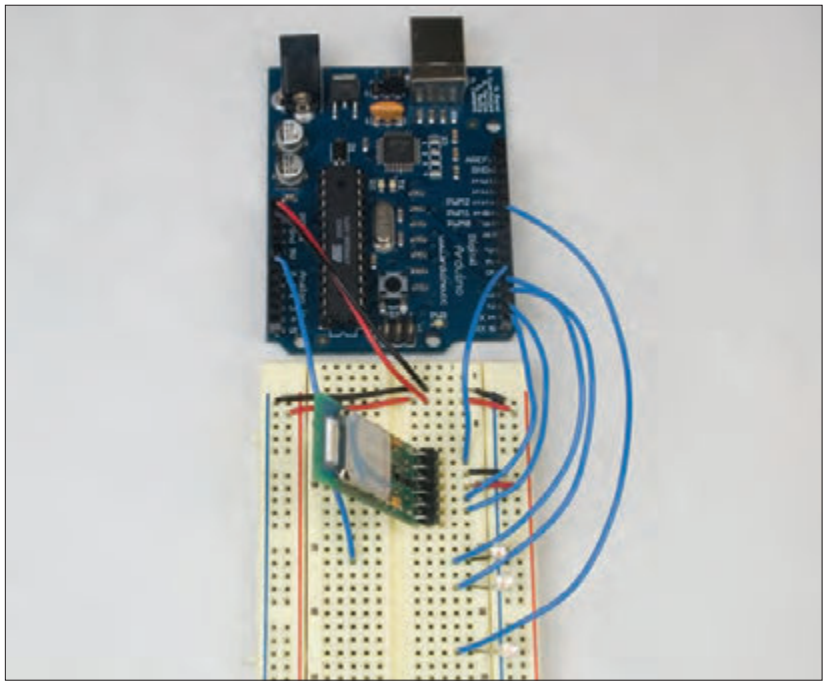
The program that follows connects to another BlueSMiRF with a set address, and when it connects, it sends its potentiometer value as an ASCII string, terminated by an asterisk, like this: `123*`

Because there are so many newline characters and carriage returns in the AT command responses, it's simplest just to use a terminator that isn't used in the command set. That's why you're using an asterisk in this case.

Just like the XBee example, this program also looks for incoming ASCII strings (terminated by an asterisk this time) and converts them to use as a PWM value to dim an LED on pin 9.

First, the constants and variables for this program are as follows:

```
/*
  BlueRadios master connection
  Language: Wiring/Arduino

*/

#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 --
// replace with the address of your remote radio:
char remoteAddress[13] = "112233445566";

byte connected = false;      // whether you're connected
```



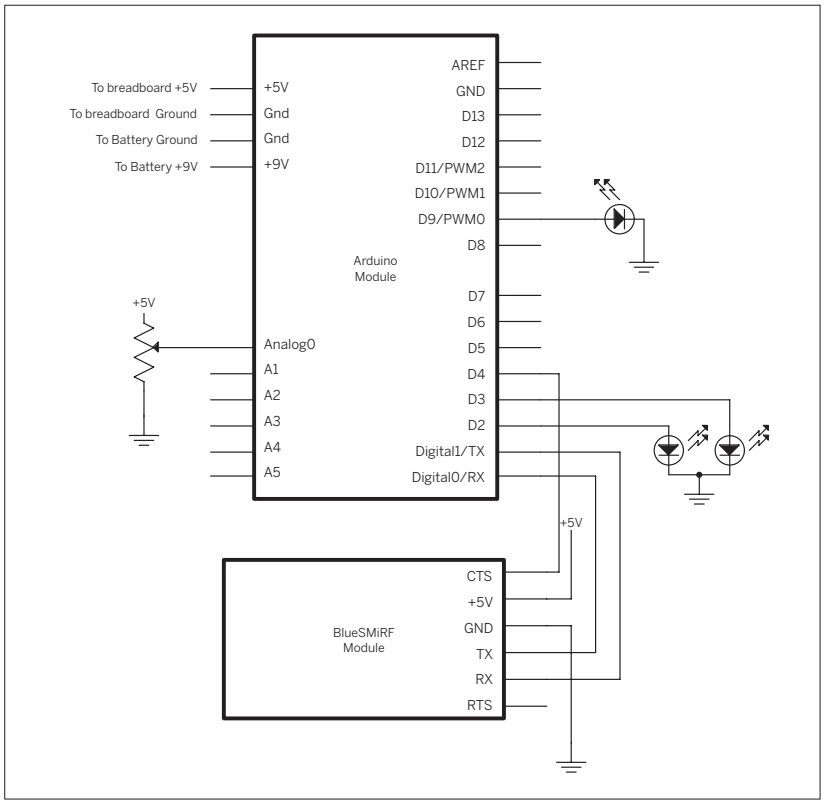


Figure 6-17
BlueSMiRF radio attached to a microcontroller. This circuit is almost identical to the XBee microcontroller circuit discussed earlier; only the radio is different.

The setup() method just sets the states of the pins, initializes serial, and blinks an LED, as usual. The clear-to-send pin is taken low here so that the BlueSMiRF can start sending serial data to the microcontroller. Then an initial attempt to connect the radios is made, using a method called BTConnect():

```
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 that you're done with setup:
  blink(3);

}
```

Here's the BTConnect() method. It sends +++ followed by ATDH to break any existing connection, then sends the ATDM command, which requests a connection to the other radio.

```
void BTConnect() {
  Serial.print("+++\r");
  delay(250);
  Serial.print("ATDH\r");
  Serial.print("ATDM");
  Serial.print(remoteAddress);
  Serial.print(",1101\r");
}
```

The readSensor() method checks the value of the potentiometer:

```
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;
}
```

» The blink() method is the same as it was in the earlier XBee example:

```
void blink(int howManyTimes) {
  for (int i=0; i< howManyTimes; i++) {
    digitalWrite(txLed, HIGH);
    delay(200);
    digitalWrite(txLed, LOW);
    delay(200);
  }
}
```

» The main loop listens for incoming serial data and handles it. If more than five seconds have passed and the radio's still not connected to the other radio, the microcontroller attempts to connect again:

```
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();
  }
}
```

The handleSerial() method is similar to the one in the XBee project, but there are some important differences. First, because there's a dedicated connection between the two radios, you need to keep track of the connection status. When a new connection is made, the BlueSMiRF will send a serial message like this before dropping into data mode:

```
CONNECT,0016CB202BF3
```

When the connection's broken, it will send this message, and stay in command mode:

```
DISCONNECT
```

In addition, when it's searching, there are a couple of other messages it might send:

```
NO CARRIER
NO ANSWER
```

Now that you know all the messages that you might get, you can establish what messages to look for. If you want to be thorough, you'd need to wait for the whole message each time and confirm that it's the right message. Parsing strings in most microcontroller languages is tricky, because of their limited memory, so it's simpler to look for unique characters in the various strings, because then you have to check for just one byte each time. It's not as thorough, but in this case, it works very consistently.

The only time a comma shows up is in the CONNECT message, so you can use that as a sign of connection. The only time a S shows up is in the DISCONNECT message, so you can use that as a sign of disconnection. You might be tempted to use D, but remember that D is a hexadecimal digit, so it might show up in the CONNECT message. Likewise C, which also shows up in three of the four messages. Finally, R shows up only in the other two messages, and they both only show up when you're disconnected, so you can use them in case your microcontroller misses a DISCONNECT message.