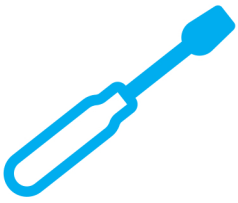
Practical methods for connecting physical objects.

2nd Edition

Making Things Talk



Learn by Discovery

Tom Igoe

USING SENSORS, NETWORKS, AND ARDUINO TO SEE, HEAR, AND FEEL YOUR WORLD



MAKER MEDIA™

Make: makezine.com

Engineering/Robotics

Make: PROJECTS

Making Things Talk

Building electronic projects that interact with the physical world is good fun. But when the devices you've built start to talk to each other, things really get interesting. With 33 easy-to-build projects, Making Things Talk shows you how to get your gadgets to communicate with you and your environment. It's perfect for people with little technical training but a lot of interest.

Make microcontrollers, PCs, servers, and smartphones talk to each other.

Maybe you're a science teacher who wants to show students how to monitor the weather in several locations at once. Or a sculptor looking to stage a room of choreographed mechanical sculptures. In this expanded edition, you'll learn how to form networks of smart devices that share data and respond to commands.

With a little electronic know-how, inexpensive microcontroller kits, and some network modules to make them communicate, you can get started on these projects right away:

Blink
Your very first program.

Monski Pong
Control a video game with a fluffy pink monkey.

Networked Air-Quality Meter
Download and display the latest report for your city.

XBee Toxic Sensor
Use ZigBee, sensors, and a cymbal monkey to warn of toxic vapors.

Bluetooth GPS
Build a battery-powered GPS that reports its location over Bluetooth.

Tweets from RFID
Read Twitter streams with a wave of an RFID tag.

You will:

- Call your home thermostat with a smartphone and change the temperature.
- Create your own game controllers that communicate over a network.
- Use ZigBee, Bluetooth, Infrared, and plain old radio to transmit sensor data wirelessly.
- Work with Arduino 1.0, Processing, and PHP—three easy-to-use, open source environments.
- Write programs to send data across the Internet based on physical activity in your home, office, or backyard.

Tom Igoe teaches courses in physical computing and networking at the Interactive Telecommunications Program at the Tisch School of the Arts at New York University. In his teaching and research, he explores ways to allow digital technologies to sense and respond to a wide range of human physical expression. He coauthored *Physical Computing: Sensing and Controlling the Physical World with Computers* with Dan O'Sullivan, which has been adopted by numerous digital art and design schools around the world. He is a contributor to MAKE magazine and a cofounder of the Arduino open source microcontroller project. He hopes to someday to visit Svalbard and Antarctica.

US $34.99 CAN $36.99
ISBN: 978-1-449-39243-7





Make:
makezine.com

MAKER MEDIA

Making Things Talk

Second Edition

Tom Igoe

MAKER MEDIA
SEBASTOPOL, CA

Making Things Talk

by Tom Igoe

Copyright © 2011 Maker Media, Inc. All rights reserved. Printed in Canada.

Published by Maker Media, Inc.
1005 Gravenstein Highway North, Sebastopol, CA 95472.

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

Print History
September 2007
First Edition

September 2011
Second Edition

Editor: Brian Jepson
Proofreader: Marlowe Shaeffer
Cover Designer: Karen Montgomery
Production Editor: Adam Zaremba
Indexer: Lucie Haskins
Cover Photograph: Tom Igoe

The Make logo and Maker Media logo are registered trademarks of Maker Media, Inc. The MAKE: Projects series designations, *Making Things Talk*, and related trade dress are trademarks of Maker Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Maker Media, Inc. was aware of the trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Please note: 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 *Making Things Talk* is at your own risk. Maker 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.

ISBN: 978-1-449-39243-7
[TI] [2013-03-22]

Contents

Preface ..... **vii**
Who This Book Is For viii
What You Need to Know ix
Contents of This Book ix
On Buying Parts x
Using Code Examples xi
Using Circuit Examples xi
Acknowledgments for the First Edition xii
Note on the Second Edition xiv

Chapter 1: The Tools **1**
It Starts with the Stuff You Touch 2
It's About Pulses 2
Computers of All Shapes and Sizes 3
Good Habits 4
Tools 5
Using the Command Line 13
Using an Oscilloscope 34
It Ends with the Stuff You Touch 35

Chapter 2: The Simplest Network **37**
Supplies for Chapter 2 38
Layers of Agreement 40
Making the Connection: The Lower Layers 42
Project 1: Type Brighter 46
Project 2: Monski Pong 50
Flow Control 62
Project 3: Wireless Monski Pong 64
Project 4: Negotiating in Bluetooth 68
Conclusion 72

Chapter 3: A More Complex Network **75**
Supplies for Chapter 3 76
Network Maps and Addresses 77
Project 5: Networked Cat 89
Conclusion 112

Chapter 4: Look, Ma, No Computer! Microcontrollers on the Internet. 115
Supplies for Chapter 4 117
Introducing Network Modules 118
Project 6: Hello Internet! 120
An Embedded Network Client Application 127
Project 7: Networked Air-Quality Meter 127
Programming and Troubleshooting Tools for Embedded Modules 140
Conclusion 147

Chapter 5: Communicating in (Near) Real Time 149
Supplies for Chapter 5 150
Interactive Systems and Feedback Loops 151
Transmission Control Protocol: Sockets & Sessions 152
Project 8: Networked Pong 153
The Clients 155
Conclusion 178

Chapter 6: Wireless Communication 181
Supplies for Chapter 6 182
Why Isn't Everything Wireless? 184
Two Flavors of Wireless: Infrared and Radio 185
Project 9: Infrared Control of a Digital Camera 188
How Radio Works 190
Project 10: Duplex Radio Transmission 193
Project 11: Bluetooth Transceivers 206
Buying Radios 216
What About WiFi? 216
Project 12: Hello WiFi! 217
Conclusion 220

Chapter 7: Sessionless Networks 223
Supplies for Chapter 7 224
Sessions vs. Messages 226
Who's Out There? Broadcast Messages 227
Project 13: Reporting Toxic Chemicals in the Shop 232
Directed Messages 246
Project 14: Relaying Solar Cell Data Wirelessly 248
Conclusion 258

Chapter 8: How to Locate (Almost) Anything 261
Supplies for Chapter 8 262
Network Location and Physical Location 264
Determining Distance 267
Project 15: Infrared Distance Ranger Example 268
Project 16: Ultrasonic Distance Ranger Example 270
Project 17: Reading Received Signal Strength Using XBee Radios 273
Project 18: Reading Received Signal Strength Using Bluetooth Radios 276
Determining Position Through Trilateration 277
Project 19: Reading the GPS Serial Protocol 278

Determining Orientation 286
Project 20: Determining Heading Using a Digital Compass 286
Project 21: Determining Attitude Using an Accelerometer 290
Conclusion 299

Chapter 9: Identification 301
Supplies for Chapter 9 302
Physical Identification 304
Project 22: Color Recognition Using a Webcam 306
Project 23: Face Detection Using a Webcam 310
Project 24: 2D Barcode Recognition Using a Webcam 313
Project 25: Reading RFID Tags in Processing 318
Project 26: RFID Meets Home Automation 321
Project 27: Tweets from RFID 329
Network Identification 353
Project 28: IP Geocoding 355
Conclusion 360

Chapter 10: Mobile Phone Networks and the Physical World. 363
Supplies for Chapter 10 364
One Big Network 366
Project 29: CatCam Redux 369
Project 30: Phoning the Thermostat 386
Text-Messaging Interfaces 393
Native Applications for Mobile Phones 396
Project 31: Personal Mobile Datalogger 401
Conclusion 415

Chapter 11: Protocols Revisited 417
Supplies for Chapter 11 418
Make the Connections. 419
Text or Binary? 422
MIDI 425
Project 32: Fun with MIDI 427
Representational State Transfer 435
Project 33: Fun with REST 437
Conclusion 440

Appendix: Where to Get Stuff 443
Supplies 444
Hardware 447
Software 452

Index 455



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. *Making Things Talk* teaches you how to make things that have computational power talk to each other, and about giving people the ability to use those things to communicate.

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 to which its creator allows you access (as well as the 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 simple modules. 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, regardless of 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. This book is a primer for people with little technical training and a lot of interest. This book is 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 their programming tools have gone from being arcane items to common, easy-to-use tools. Elementary school students are using the tools that baffled graduate students 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 respond to, sense, and interpret.

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 want to make that thing communicate with 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 quickly as possible.
X

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 programming experience begins with microcontrollers can do wonderful things with some sensors and a couple of servomotors, but they may not have done much to enable 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 your background. If so, feel free to skip past the stuff you know to get to the meatier parts.

If you've never used a microcontroller, you'll need a little background before starting this book. I recommend you read my previous book, **Physical Computing: Sensing and Controlling the Physical World with Computers** (Thomson), co-authored with Dan O'Sullivan, which introduces the fundamentals of electronics, microcontrollers, and physical interaction design.

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 to teach nonprogrammers 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.

This book includes code examples in a few different programming languages. They're all fairly simple examples, so if you don't want to work in the languages provided, you can use the comments in these examples to rewrite them in your favorite language.
X

Contents of This Book

This book explains the concepts that underlie networked objects and then provides recipes to illustrate each set of concepts. Each chapter contains instructions for building 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 discusses the basic devices that hold it together, as well as the basic relationships among those 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 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. You'll learn how to write your own server program that you can connect to anything connected to the Net. You'll connect to this server program from the command line and from a microcontroller, so that you can understand how different types of devices 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, by 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 from 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 it offers some thoughts on the relationship between physical location and network relationships.

Chapter 9 deals with identification in physical space and network space. You'll learn a few techniques for generating unique network identities based on physical characteristics. You'll also learn a bit about how to determine a networked device's characteristics.

Chapter 10 introduces mobile telephony networks, covering many of the things that you can now do with phones and phone networks.

Chapter 11 provides a look back at the different types of protocols covered in this book, and gives you a framework to fit them all into for future reference.
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 online vendors.

Jameco (http://jameco.com), Digi-Key (www.digikey.com), and Farnell (www.farnell.com) are general electronics parts retailers, and they sell many of the same things. Others, like Maker Shed (www.makershed.com), SparkFun (www.sparkfun.com), and Adafruit (http://adafruit.com) carry specialty components, kits, and bundles that make it easy to do popular projects. A full list of suppliers is included in the Appendix. Feel free to substitute parts for things with which you are familiar.

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 to someone 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. 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 Maker Media 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 2011 Maker Media, 978-1-4493-9243-7." 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 when building this book's projects. 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.

x

Acknowledgments for the First Edition

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 more than the past decade. It is a lively and warm place to work, crowded with many talented people. This book grew out of a class, 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 enjoyment in working there.

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

Dan O'Sullivan, my colleague and now chair of the program, introduced me to physical computing and then generously allowed me to share in teaching it and shaping its role at ITP. He is 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 were also close advisors on this project. Clay watched indulgently as the pile of parts mounted in our office, and he graciously interrupted his own writing to give opinions on my ideas. Daniel Rozin offered valuable critical insight as well, and his ideas are heavily influential in this book. Dan Shiffman read many drafts and offered helpful feedback. He also contributed many great code samples and libraries.

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

The rest of the faculty and staff at ITP 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 tolerated all kinds of insanity in the name of physical computing and networked objects, and made things possible for the other faculty and me, as well as the students. Research residents Carlyn Maw, Todd Holoubek, John Schimmel, Doria Fan, David Nolen, Peter Kerlin, and Michael Olson assisted faculty and students over the past few years to realize projects that influenced the ones you see in these chapters. 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 their own techniques 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 cite specifics where they come up, but in general, I'd like to thank all the students who took my Networked Objects class—they helped me understand what this is all about. Those from the 2006 and 2007 classes were particularly influential, because they had to learn the stuff from early drafts of this book. They have caught several important mistakes in the manuscript.

A few people 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 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 think about identity. Mike Kuniavsky and the folks at the "Sketching in Hardware" workshops in 2006 and 2007 helped me 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 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 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, I must give a special mention to Nathan Seidle at Spark Fun. 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 answered my countless emails and phone calls helpfully and cheerfully.

In this book's projects, I drew ideas from many colleagues from around the world 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 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 encouraged of all of my ideas, dealt patiently with my delays, and indulged me 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 than I could ever have hoped. 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:

Maker 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, Inc. is devoted entirely to the growing community of resourceful people who believe that if you can imagine it, you can make it. Maker Media encourages the Do-It-Yourself mentality by providing creative inspiration and instruction.

For more information about Maker Media, Inc., visit us online: http://makermedia.com

Note on the Second Edition

Two general changes prompted the rewriting of this book: the emergence of an open source hardware movement, and the growth of participatory culture, particularly around making interactive things. The community surrounding Arduino, and the open source hardware movement more generally, has grown quickly. The effects of this are still being realized, but one thing is clear: object-oriented hardware and physical computing are becoming an everyday reality. Many more people are making things with electronics now than I could have imagined in 2005.

Before any technology is adopted in general use, there has to be a place for it in the popular imagination. People with no knowledge of the technology must have some idea what it is and for what it can be used. Prior to 2005, I spent a lot of time explaining to people what physical computing was and what I meant by "networked objects." Nowadays, everyone knows the Wii controller or the Kinect as an example of a device that expands the range of human physical expression available to computers. These days, it's difficult to find an electronic device that isn't networked.

While it's been great to see these ideas gain a general understanding, what's even more exciting is seeing them gain in use. People aren't just using their Kinects for gaming, they're building them into assistive interfaces for physically challenged clients. They're not just playing with the Wii, they're using it as a musical instrument controller. People have become accustomed to the idea that they can modify the use of their electronics—and they're doing it.

When I joined the project, my hope for Arduino was that it might fill a need for something more customizable than consumer electronic devices were at the time, yet be less difficult to learn than microcontroller systems. I thought the open source approach was a good way to go because it meant that hopefully the ideals of the platform would spread beyond the models we made. That hope has been realized in the scores of derivative boards, shields, spinoff products, and accessories that have popped up in the last several years. It's wonderful to see so many people not just making electronics for others to build on, but doing it in a way that doesn't demand professional expertise to get started.

The growth of Arduino shields and libraries has been big enough that I almost could have written this edition so that you wouldn't have to do any programming or circuit building. There's a shield or a library to do almost every project in this book. However, you can only learn so much by fitting premade pieces together, so I've tried to show some of the principles underlying electronic communications and physical interfaces. Where there is a simple hardware solution, I've indicated it but shown the circuit it encloses as well. The best code libraries and circuit designs practice what I think of as "glass-box enclosure"—they enclose the gory details and give you a convenient interface, but they let you look inside and see what's going on if you're interested. Furthermore, they're well-constructed so that the gory details don't seem that gory when you look closely at them. Hopefully, this edition will work in much the same way.

Software Reference

There have been a number of large changes made to the Arduino platform since I started this edition. The Arduino IDE was in beta development, but by the time this book comes out, version 1.0 will be available. If you're already familiar with Arduino, please make sure you've downloaded version 1,0beta1 or later of the IDE. This book was written using Arduino 1.0 beta1, which is available online at http://code.google.com/p/arduino/wiki/Arduino1. The final 1.0 version will be available on the Download page at www.arduino.cc. Check the Arduino site for the latest updates. The code for this book can be found online on my gitHub repository at https://github.com/tigoe/MakingThingsTalk2 and I'll write about any changes on the blog, www.makingthingstalk.com.

Hardware Reference

To keep the focus on communications between physical devices, I've chosen to use the Arduino Uno as the reference hardware design for this edition. Everything in this book will work on an Arduino Uno with the appropriate accessories or shields. A few projects were made with specialty Arduino models like the Arduino Ethernet or the Arduino LilyPad because their form factor was the most appropriate, but even those projects were tested on the Uno. Anything that is compatible with the Uno should be able to run this code and interface with these circuits.

Acknowledgments for the Second Edition

The network of people who make this book possible continues to grow.

The changes in this edition are due in no small part to the work of my partners on the Arduino team. Working with Massimo Banzi, David Cuartielles, Gianluca Martino, and David Mellis continues to be enjoyable, challenging, and full of surprises. I'm lucky to have them as collaborators.

The Interactive Telecommunications Program at NYU continues to support me in everything I do professionally. None of this would be possible without the engagement of my colleagues there. Dan O'Sullivan, as always, was a valued advisor on many of the projects that follow. Daniel Shiffman and Shawn Van Every provided assistance with desktop and Android versions of Processing. Marianne Petit, Nancy Hechinger, Clay Shirky, and Marina Zurkow offered critical and moral support. Red Burns, as ever, continues to inspire me on how to empower people by teaching them to understand the technologies that shape their lives.

The cast of resident researchers and adjunct professors at ITP is ever-changing and ever-helpful. During this edition, research residents Mustafa Bağdatlı, Caroline Brown, Jeremiah Johnson, Meredith Hasson, Liesje Hodgson, Craig Kapp, Adi Marom, Ariel Nevarez, Paul Rothman, Ithai Benjamin, Christian Cerrito, John Dimatos, Xiaoyang Feng, Kacie Kinzer, Zannah Marsh, Corey Menscher, Matt Parker, and Tymm Twillman helped with examples, tried projects, out, and kept things going at ITP when I was not available.

Adjunct faculty members Thomas Gerhardt, Scott Fitzgerald, Rory Nugent, and Dustyn Roberts were valued collaborators by teaching this material in the Introduction to Physical Computing course.

Rob Faludi remains my source on all things XBee- and Digi-related.

Thanks to Antoinette LaSorsa and Lille Troelstrup at the Adaptive Design Association for permission to use their tilt board design in Chapter 5.

Many people contributed to the development of Arduino through our developers mailing list and teachers list. In particular, Mikal Hart, Michael Margolis, Adrian McEwen, and Limor Fried influenced this book through their work on key communication libraries like SoftwareSerial, Ethernet, and TextFinder, and also through their personal advice and good nature in answering my many questions off-list. Michael Margolis' *Arduino Cookbook* (O'Reilly) was a reference for some of the code in this book as well. Thanks also to Ryan Mulligan and Alexander Brevig for their libraries, which I've used and adapted in this book.

Limor Fried and Phillip Torrone, owners of Adafruit, were constant advisors, critics, and cheerleaders throughout this book. Likewise, Nathan Seidle at SparkFun continues to be one of my key critics and advisors. Adafruit and SparkFun are my major sources of parts, because they make stuff that works well.

This edition looks better graphically thanks to Fritzing, an open source circuit drawing tool available at http://fritzing.org. Reto Wettach, André Knörig, and Jonathan Cohen created a great tool to make circuits and schematics more accessible. Thanks also to Ryan Owens at SparkFun for giving me advance access to some of its parts drawings. Thanks to Giorgio Olivero and Jody Culkin for additional drawings in this edition.

Thanks to David Boyhan, Jody Culkin, Zach Eveland, and Gabriela Gutiérrez for reading and offering feedback on sections of the manuscript.

Thanks to Keith Casey at Twilio; Bonifaz Kaufmann, creator of Amarino; Andreas Göransson for his help on Android; and Casey Reas and Ben Fry for creating Processing's Android mode, and for feedback on the Android section.

New projects have inspired the new work in this edition. Thanks to Benedetta Piantella and Justin Downs of Groundlab, and to Meredith Hasson, Ariel Nevarez, and Nahana Schelling, creators of SIMbalink. Thanks to Timo Arnall, Elnar Sneve Martinussen, and Jørn Knutsen at www.nearfield.org for their RFID inspiration and collaboration.Thanks to Daniel Hirschmann for reminding me how exciting lighting is and how easy DMX-512 can be. Thanks to Mustafa Bağdatlı for his advice on Poker Face, and thanks to Frances Gilbert and Jake for their role in the CatCam 2 project. Apologies to Anton Chekhov. Thanks to Tali Padan for the comedic inspiration.

Thanks to Giana Gonzalez, Younghui Kim, Jennifer Magnolfi, Jin-Yo Mok, Matt Parker, Andrew Schneider, Gilad Lotan, Angela Pablo, James Barnett, Morgan Noel, Noodles, and Monski for modeling projects in the chapters.

Thanks, as ever, to the MAKE team, especially my editor and collaborator Brian Jepson. His patience and persistence made another edition happen. Thanks to technical editor Scott Fitzgerald, who helped pull all the parts together as well. If you can find a part on the Web from this book, thank Scott. Thanks also to my agent Neil Salkind and everyone at Studio B.

In the final weeks of writing this edition, a group of close friends came to my assistance and made possible what I could not have done on my own. Zach Eveland, Denise Hand, Jennifer Magnolfi, Clive Thompson, and Max Whitney donated days and evenings to help cut, solder, wire, and assemble many of the final projects, and they also kept me company while I wrote. Joe Hobaica, giving up several days, provided production management to finish the book. He orchestrated the photo documentation of most of the new projects, organized my workflow, kept task lists, shopped for random parts, checked for continuity, and reminded me to eat and sleep. Together, they reminded me that making things talk is best done with friends.

X


VACU-VIS


FOCUS
GATE

1

MAKE: PROJECTS 

The Tools

This book is a cookbook of sorts, and this chapter covers the key 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 it say **"Hello World!"** Chances are you've used some of the tools in this chapter before—or ones 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 how it feels 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 object 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—such as knobs, switches, keys, and other sensors—that react to your actions. The connectors that join objects are also part of the physical interface. Every network of objects begins and ends with a physical interface. Even though some objects in a network (such as software objects) have no physical interface, people construct 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 see, hear, and touch, they will never be 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; 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 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 approach a physical interface. Don't try to use all the functions at once; first, learn what each function does on its own. 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 understand a few things about this 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 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 useful instructions if you can't form a good sentence.

One thing that all communications protocols have in common—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 , which carry 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, it's got a keyboard, screen, and 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-step 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—it 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 don't think of servers 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 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, 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 how you feel 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, but since you never know when the other person is going to say something, you have to listen all the time. In networking terms, this means 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 inclined to first work out all the logic of your system—and all the steps of your messages before you start to connect things—then build it, and finally 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 when building these projects happen 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 misstep 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 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 all the parts 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 that 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 bit of compromise before you start to build, will make things flow much more smoothly.

Ask Politely for Clarification

Messages get garbled in countless ways. Perhaps you hear something that 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, rather than assume, that a message was sent. Saying nothing can be worse than saying something wrong. Minor problems can become major when no one speaks up to acknowledge that there's an issue. The same thing can occur in network communications. One device may wait forever for a message from the other side, not knowing, for example, that the remote device is unplugged. When you don't receive a response, send another message. Don't resend it too often, and give the other party time to reply. 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, you'll need physical tools, software, and (computer) hardware.

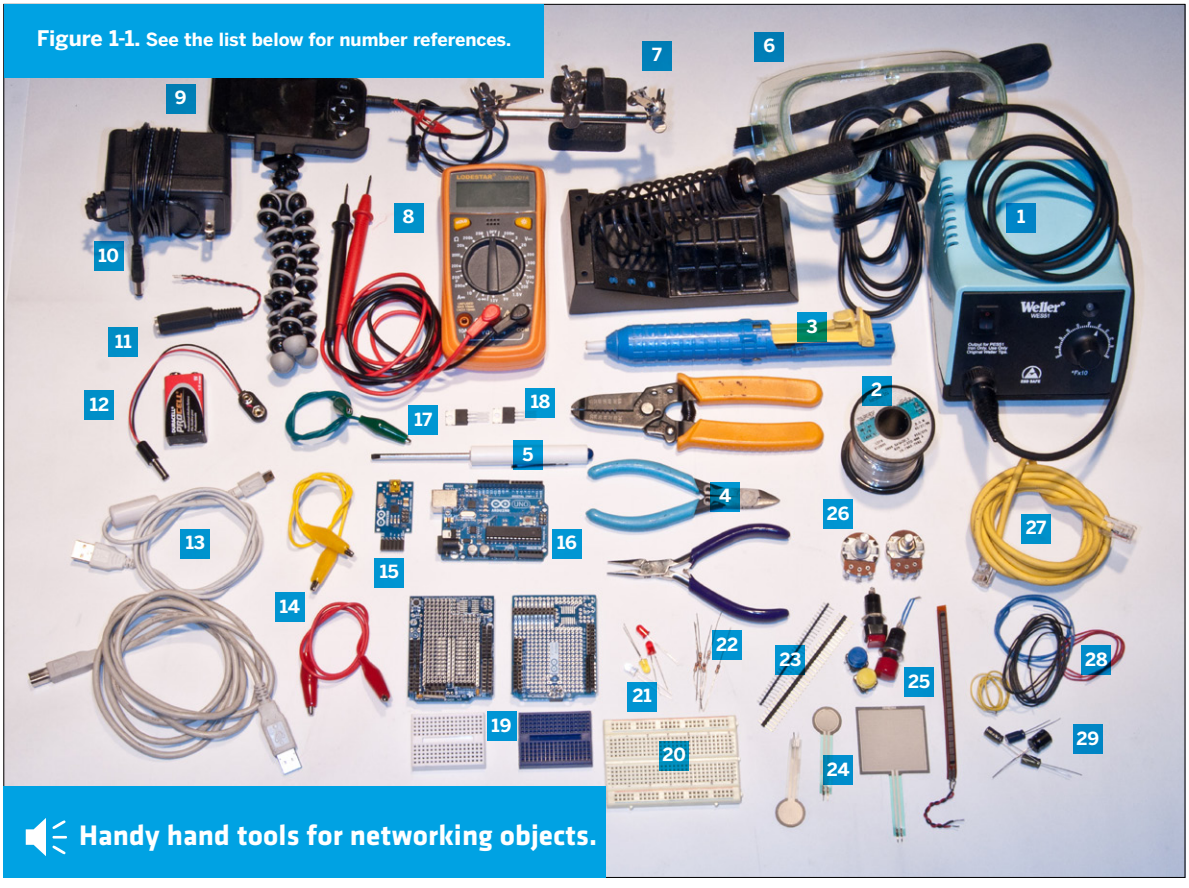
Physical Tools

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 that 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 for less. Feel free to substitute your favorite vendors. A list of vendors can be found in the Appendix.

Figure 1-1. See the list below for number references.



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 works 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 (http://jameco.com): 146595; Farnell (www.farnell.com): 1568159; RadioShack (http://radioshack.com): 640-2801 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; Farnell: 419266; RadioShack: 640-0013

3 Desoldering pump This helps when you mess up while soldering. Jameco: 305226; Spark Fun (www.SparkFun.com): TOL-00082; Farnell: 3125646

4 Wire stripper, Diagonal cutter, Needle-nose 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; Farnell: 609195; Spark Fun: TOL-00089; RadioShack: 640-2129A
Diagonal cutter: Jameco: 161411; Farnell: 3125397; Spark Fun: TOL-00070; RadioShack: 640-2043
Needlenose pliers: Jameco: 35473; Farnell: 3127199; Spark Fun: TOL-00079; RadioShack: 640-2033

5 Mini-screwdriver Get one with both Phillips and slotted heads. You'll use it all the time. Jameco: 127271; Farnell: 4431212; RadioShack: 640-1963

6 Safety goggles Always a good idea when soldering, drilling, or other tasks. Spark Fun:SWG-09791; Farnell: 1696193

7 Helping hands These make soldering much easier. Jameco: 681002; Farnell: 1367049

8 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; Farnell: 7430566; Spark Fun: TOL-00078; RadioShack: 22-182

9 Oscilloscope Professional oscilloscopes are expensive, but the DSO Nano is only about $100 and a valuable aid when working on electronics. Spark Fun: TOL-10244 (v2); Seeed Studio (www.seeed-studio.com): (TOL114C3M; Maker SHED (www.makershed.com): MKSEEED11

10 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 adapter with the same dimensions. Jameco: 170245 (12V, 1000mA); Farnell: 1176248 (12V, 1000mA); Spark Fun: TOL-00298; RadioShack: 273-355 (9V 800mA)

11 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 (www.digikey.com): CP-024A-ND; Farnell: 3648102

12 9V Battery snap adapter and 9V battery When you want to run a project off battery power, these adapters are a handy way to do it. Spark Fun: PRT-09518; Adafruit (http://adafruit.com): 80; Digi-Key: CP3-1000-ND and 84-4K-ND; Jameco: 28760 and 216452; Farnell: 1650675 and 1737256; RadioShack: 270-324 and 274-1569

13 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. Spark Fun: CAB-00512, CAB-00598; Farnell: 1838798, 1308878

14 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; RS (www.rs-online.com): 483-859; Spark Fun: CAB-00501; RadioShack: 278-016

15 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. Spark Fun: BOB-00718; Arduino Store (store.arduino.cc): A000014

16 Microcontroller module The microcontroller shown here is an Arduino Uno. Available from Spark Fun and Maker SHED (http://store.arduino.cc/ww/) in the U.S., and from multiple distributors internationally. See http://arduino.cc/en/Main/Buy for details about your region.

17 Voltage regulator 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: Digi-Key: 576-1134-ND; Jameco: 242115; Farnell: 1703357; RS: 534-3021
5V: Digi-Key: LM7805CT-ND; Jameco: 51262; Farnell: 1703357; RS: 298-8514

18 TIP120 Transistor Transistors act as digital switches, allowing you to control a circuit with high current or voltage from one with lower current and voltage. There are many types of transistors, the TIP120 is one used in a few projects in this book. Note that the TIP120 looks just like the voltage regulator next to it. Sometimes electronic components with different functions come in the same physical packages, so you need to check the part number written on the part. Digi-Key: TIP120-ND; Jameco: 32993; Farnell: 9804005

19 Prototyping shields These are add-on boards for the Arduino microcontroller module that have a bare grid of holes to which you can solder. You can build your own circuits on them by soldering, or you can use a tiny breadboard (also shown) to test circuits quickly. These are handy for projects where you need to prototype quickly, as well as a compact form to the electronics. Adafruit: 51; Arduino Store: A000024; Spark Fun: DEV-07914; Maker SHED: MSMS01
Breadboards for protoshields: Spark Fun: PRT-08802; Adafruit: included with board; Digi-Key: 923273-ND

20 Solderless breadboard Having a few around can be handy. I like the ones with two long rows on either side so that you can run power and ground on both sides. Jameco: 20723 (2 bus rows per side); Farnell: 4692810; Digi-Key: 438-1045-ND; Spark Fun: PRT-00137; RadioShack: 276-002

21 Spare LEDs for tracing signals LEDs are to the hardware developer what print statements are to the software developer. They let you see quickly whether there's voltage between two points, or whether a signal is going through. Keep spares on hand. Jameco: 3476; Farnell: 1057119; Digi-Key: 160-1144-ND; RadioShack: 278-016

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

23 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; Farnell: 1593411

24 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 (www.imagesco.com): FLX-01
Force-sensing resistors: Parallax (www.parallax.com): 30056; Images SI: FSR-400, 402, 406, 408

25 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; Spark Fun: COM-00097
Panel-mount type: Digi-Key: GH1344-ND; Jameco: 164559PS

26 Potentiometers You'll need potentiometers to let people adjust settings in your project. Jameco: 29081; Spark Fun: COM-09939; RS: 91A1A-B28-B15L; RadioShack: 271-1715; Farnell: 1760793

27 Ethernet cables A couple of these will come in handy. Jameco: 522781; RadioShack: 55010852

28 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

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



You're going to run across some hardware in the following chapters that was brand new when this edition was written, including the Arduino Ethernet board, the Arduino WiFi shield, wireless shield, RFID shield, USB-to-Serial adapter, and more. The distributors listed here didn't have part numbers for them as of this writing, so check for them by name. By the time you read this, distributors should have them in stock.

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

D Digi-Key (http://digikey.com) **R** RS (www.rs-online.com)
J Jameco (http://jameco.com) **F** Farnell (www.farnell.com)

RESISTORS

100Ω ..... **D** 100QBK-ND, **J** 690620, **F** 9337660, **R** 707-8625
220Ω ..... **D** 220QBK-ND, **J** 690700, **F** 9337792, **R** 707-8842
470Ω ..... **D** 470QBK-ND, **J** 690785, **F** 9337911, **R** 707-8659
1K ..... **D** 1.0KQBK, **J** 29663, **F** 1735061, **R** 707-8669
10K ..... **D** 10KQBK-ND, **J** 29911, **F** 9337687, **R** 707-8906
22K ..... **D** 22KQBK-ND, **J** 30453, **F** 9337814, **R** 707-8729
100K ..... **D** 100KQBK-ND, **J** 29997, **F** 9337695, **R** 707-8940
1M ..... **D** 1.0MQBK-ND, **J** 29698, **F** 9337709, **R** 131-700

CAPACITORS

0.1µF ceramic ..... **D** 399-4151-ND, **J** 15270, **F** 3322166, **R** 716-7135
1µF electrolytic ..... **D** P10312-ND, **J** 94161, **F** 8126933, **R** 475-9009
10µF electrolytic ..... **D** P11212-ND, **J** 29891, **F** 1144605, **R** 715-1638
100µF electrolytic ..... **D** P10269-ND, **J** 158394, **F** 1144642, **R** 715-1657

VOLTAGE REGULATORS

3.3V ..... **D** 576-1134-ND, **J** 242115, **F** 1703357, **R** 534-3021
5V ..... **D** LM7805CT-ND, **J** 51262, **F** 1860277, **R** 298-8514

ANALOG SENSORS

Flex sensors ..... **D** 905-1000-ND, **J** 150551, **R** 708-1277
FSRs ..... **D** 1027-1000-ND, **J** 2128260

LED

T1, Green clear ..... **D** 160-1144-ND, **J** 34761, **F** 1057119, **R** 247-1662
T1, Red, clear ..... **D** 160-1665-ND, **J** 94511, **F** 1057129, **R** 826-830

TRANSISTORS

2N2222A ..... **D** P2N2222AGOS-ND, **J** 38236, **F** 1611371, **R** 295-028
TIP120 ..... **D** TIP120-ND, **J** 32993, **F** 9804005

DIODES

1N4004-R ..... **D** 1N4004-E3, **J** 35992, **F** 9556109, **R** 628-9029
3.3V zener (1N5226) .. **D** 1N5226B-TPCT-ND, **J** 743488, **F** 1700785

PUSHBUTTONS

PCB ..... **D** SW400-ND, **J** 119011, **F** 1555981
Panel Mount ..... **D** GH1344-ND, **J** 164559PS, **F** 1634684, **R** 718-2213

SOLDERLESS BREADBOARDS

various ..... **D** 438-1045-ND, **J** 20723, 20600, **F** 4692810

HOOKUP WIRE

red ..... **D** C2117R-100-ND, **J** 36856, **F** 1662031
black ..... **D** C2117B-100-ND, **J** 36792, **F** 1662027
blue ..... **J** 36767, **F** 1662034
yellow ..... **J** 36920, **F** 1662032

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

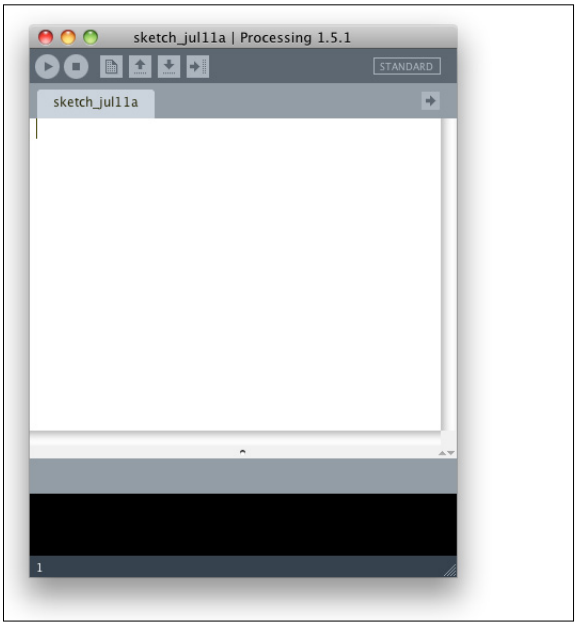


Figure 1-2
The Processing editor window.

Software Tools

Processing

The multimedia programming environment used in this book is called Processing. Based on Java, it's made for designers, artists, and others whowant to get something done without having to know all the gory details of programming. 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. It's a free, open source tool available at 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 you can run Processing on your favorite operating system. There's also Processing for Android phones and Processing for JavaScript, so you can use it in many ways. If you don't like working in Processing, you should be able to use this book's code samples and 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 then press the Run button on the top lefthand side of the toolbar.

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

“ 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 like.

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

NOTE: All code examples in this book will have comments indicating the context in which they're to be used: Processing, Processing Android mode, Arduino, PHP, and so forth.

```
/*
    Triangle drawing program
    Context: 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 braces. 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);
}
```

BASIC users: If you've never used a C-style for-next loop, it can seem forbidding. What this bit of code does is establish a variable called myCounter**. As long as a number is less than or equal to 10, it executes the instructions in the curly braces.** 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
```

"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), the creators of Processing, or their shorter book, **Getting Started with Processing** (O'Reilly). Or, read Daniel Shiffman's excellent introduction, **Learning Processing** (Morgan Kaufmann). There are dozens of other Processing books on the market, so find one whose style you like best.

Remote-Access Applications

One of the most effective debugging tools you'll use when making the projects in this book is a command-line remote-access program, which gives 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, because you'll need the command line to work with 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.

NOTE: 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.

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; 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 he's 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

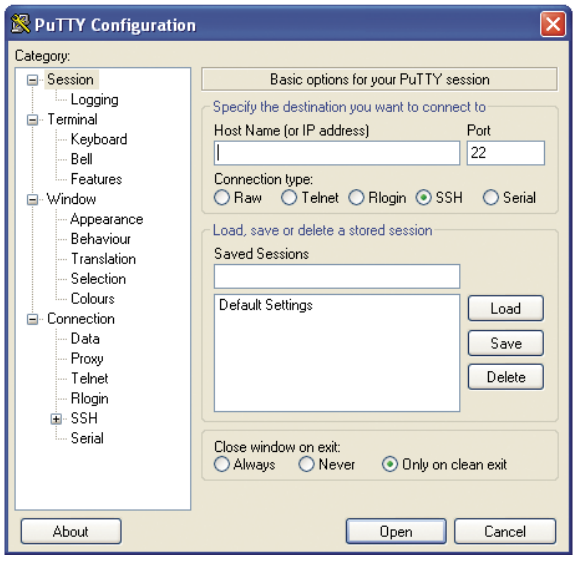


Figure 1-3
The main PuTTY window.

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, asking for your password when it connects. Type it (you won't see what you type), followed by the Enter key.

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:

```
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 of **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, as well as 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; most of the time, this is not something you'll have much need for. The third column tells you who owns it, and the fourth tells you the group (a collection of users) to which the file belongs. 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 want 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, so be careful. Don't remove any directories or files that you didn't make yourself.

Controlling Access to Files

Type ls –l to get a list of files in your current directory and to 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 who created the directory (also known as the owner of the file). Or, consider a 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 to which the file belongs (usually, the owner is a member of this group), 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, so 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 how it looks like after I typed in some text.

All the commands to work in nano are keyboard commands you type using the Ctrl key. For example, to exit the program, type Ctrl-X. The editor will then ask whether 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 colon (:) prompt at the bottom of the screen. Press the space bar 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
```

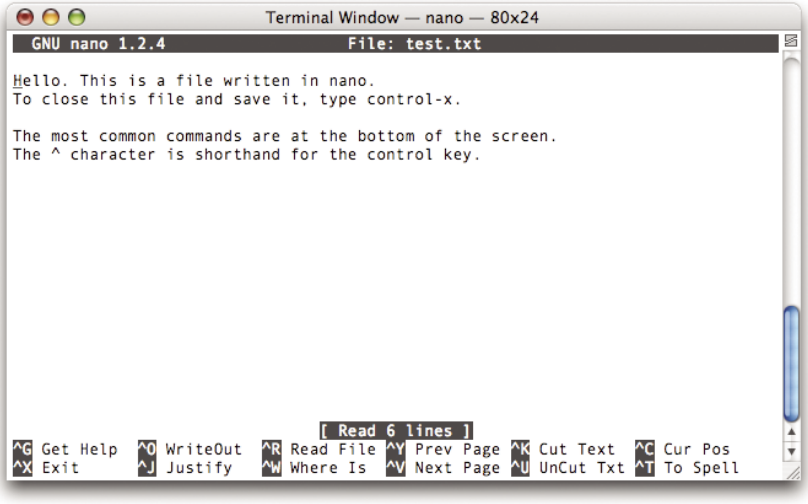


Figure 1-4
The nano text editor.

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

```
rm filename
```

Like rmdir, rm won't ask whether 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 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. When you're ready to close the connection to your server, type: logout. For more on getting around Unix and Linux systems using the command line, see **Learning the Unix Operating System** by Jerry Peek, Grace Todino-Gonguet, and John Strang (O'Reilly).

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.

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 to see whether yours does (and if not, 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 5.3.4 (cli) (built: Dec 15 2010 12:15:07)
Copyright (c) 1997-2010 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2010 Zend
Technologies
```

This tells what version of PHP is installed on your server. The code in this book was written using PHP5, 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

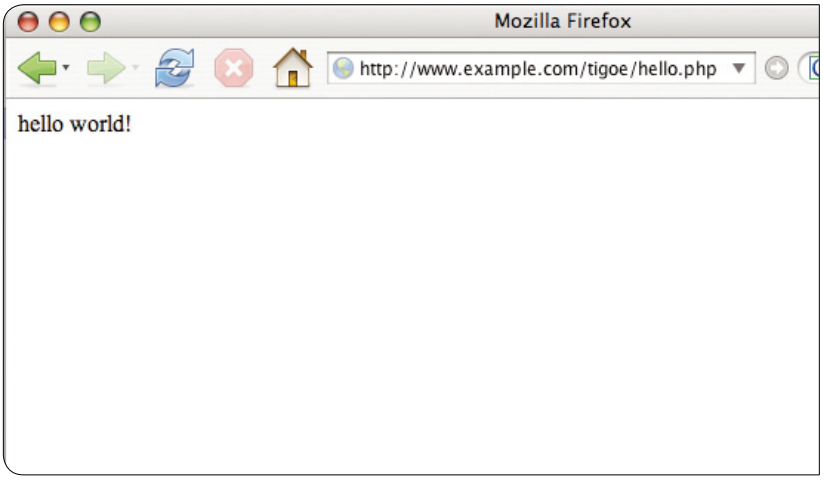


Figure 1-5
The results of your first PHP script, in a browser.

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.

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 in the following code, 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 file's address on your website. Because you saved it in **public_html**, the address is http://www.example.com/hello.php (replace example.com with your website 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 shown 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 for 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 whether 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
   Context: 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 (replacing example.com as before). 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.

Variables in PHP

PHP handles variables a little differently than Processing and Arduino. In the latter two, you give variables any name you like, as long as you don't use words that are commands in the language. You declare variables by putting the variable type before the name the first time you use it. In PHP, you don't need to declare a variable's type, but you do need to put a $ at the beginning of the name. You can see it in the PHP script above. $date is a variable, and you're putting a string into it using the date() command.

There are a number of commands for checking variables that you'll see in PHP. For example, isset() checks whether the variable's been given a value yet, or is_bool(), is_int(), and is_string() check to see whether the variable contains those particular data types (boolean, integer, and string, respectively).

In PHP, there are three important built-in variables, called environment variables, with which you should be familiar: $_REQUEST, $_GET, and $_POST. These give you the results of an HTTP request. Whether your PHP script was called by a HTML form or by a user entering a URL with a string of variables afterwards, these variables will give you the results. $_GET gives you the results if the PHP script was called using an HTTP GET request, $_POST gives the results of an HTTP POST request, and $_REQUEST gives you the results regardless of what type of request was made.

Since HTTP requests might contain a number of different pieces of information (think of all the fields you might fill out in a typical web form), these are all array variables. To get at a particular element, you can generally ask for it by name. For example, if the form you filled out had a field called Name, the name you fill in would end up in the $_REQUEST variable in an element called $_REQUEST['Name']. If the form made an HTTP POST request, you could also get the name from $_POST['Name']. There are other environment variables you'll learn about as well, but these three are the most useful for getting information from a client—whether it's a web browser or a microcontroller. You'll learn more about these, and see them in action, later in the book.

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 read **Learning PHP 5** by David Sklar (O'Reilly) 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, down-load files, and send mail to other users of the same BBS.

Nowadays, serial ports are used mainly to connect to some of your computer's peripheral devices. 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. CoolTerm is an excellent piece of freeware by Roger Meier available from http://freeware.the-meiers.org. It works on Mac OS X and Windows, and it's my personal favorite these days. If you use it, do the right thing and make a donation because it's developed in the programmer's spare time. For Windows users, PuTTY is a decent alternative because it can open both serial and ssh terminals. PuTTY is also available for Linux. Alternatively, you can keep it simple and stick with a classic: the GNU screen program running in a terminal window. OS X users can use screen as well, though it's less full-featured than CoolTerm.

Windows serial communication

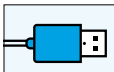
To get started, you'll need to know the serial port name. Click Start→Run (use the Search box on Windows 7), type devmgmt.msc, and press Enter to launch Device Manager. If you've got a serial device such as a Wiring or Arduino board attached, you'll see a listing for Ports (COM & LPT). Under that listing, 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 data bits, 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 serial communication

To get started, open CoolTerm and click the Options icon. In the Options tab, you'll see a pulldown menu for the port. In Mac OS X, the port names are similar to this: /dev/tty.usbmodem241241. To find your port for sure, check the list when your Arduino is unplugged, then plug it in and click Re-scan Serial Ports in the Options tab. The new port listed is your Arduino's serial connection. To open the serial port, click the Connect button in the main menu. To disconnect, click Disconnect.



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, CoolTerm, or the screen program has the serial port open to an Arduino module, for example, the Arduino IDE 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 then Ctrl-\ every time, and Windows users should disconnect the connection in PuTTY. Otherwise, you may find yourself restarting your machine a lot.

Adventurous Mac OS X users can take advantage of the fact that it's Unix-based and follow the Linux instructions.

Linux serial communication

To get started with serial communication in Linux (or Mac OS X), 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 they're more cryptic than the COM1, COM2, and so on that Windows uses. Pick your serial port and type:

```
screen portname datarate.
```

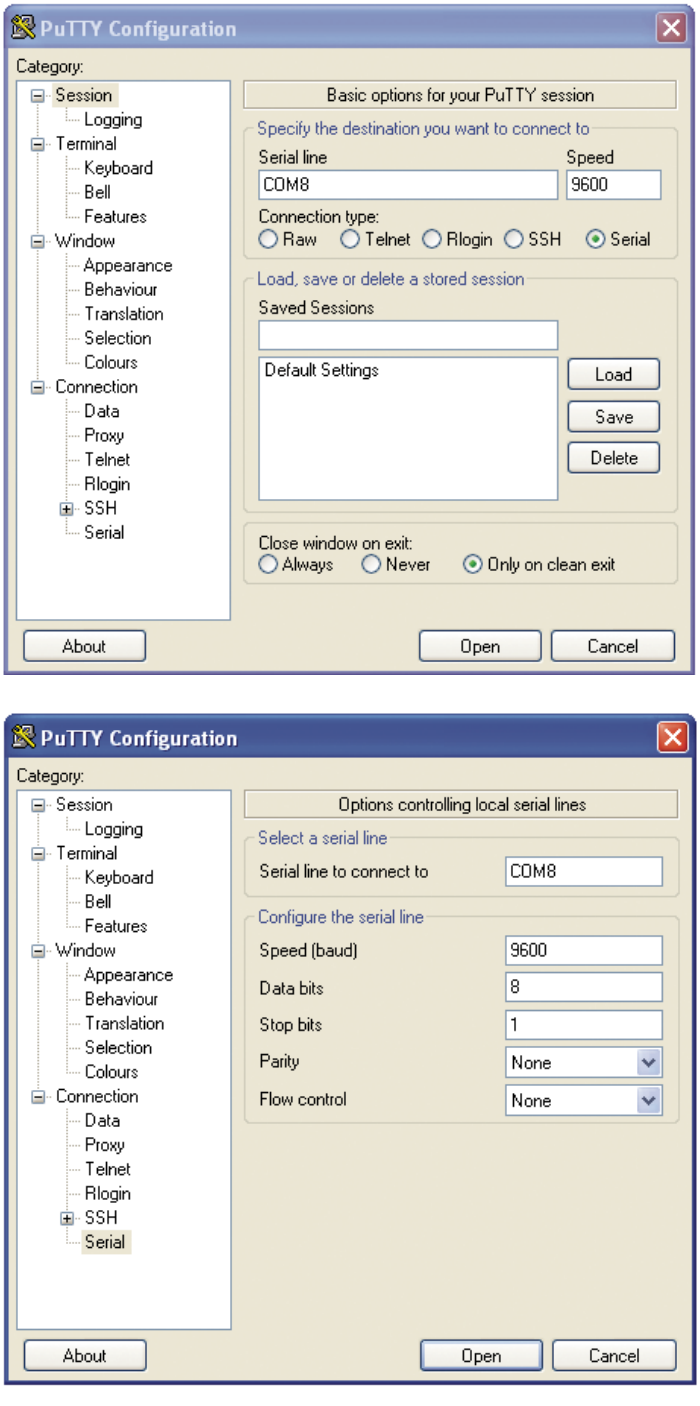


Figure 1-6
Configuring a serial connection in PuTTY.

For example, to open the serial port on an Arduino board (discussed shortly) at 9600 bits per second, you might type screen /dev/tty.usbmodem241241 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 Ctrl-A followed by Ctrl-\.

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

Hardware

Arduino, Wiring, and Derivatives

The main microcontroller used in this book is the Arduino module. Arduino and Wiring, another microcontroller module, both came out of the Institute for Interaction Design in Ivrea, Italy, in 2005. They're based on the same microcontroller family, Atmel's ATmega series (www.atmel.com), and they're both programmed in C/C++. The "dialect" they speak is based on Processing, as is the software integrated development environments (IDEs) they use. You'll see that some Processing commands have made their way into Arduino and Wiring, such as the setup() and loop() methods (Processing's draw() method was originally called loop()), the map() function, and more.

When this book was first written, there was one Wiring board, four or five variants of Arduino, and almost no derivatives. Now, there are several Arduino models, two new Wiring models coming out shortly, and scores of Arduino-compatible derivatives, most of which are compatible enough that you can program them directly from the Arduino IDE. Others have their own IDEs and will work with some (but not all) of the code in this book. Still others are compatible in their physical design but are programmed with other languages. The derivatives cover a wide range of applications.

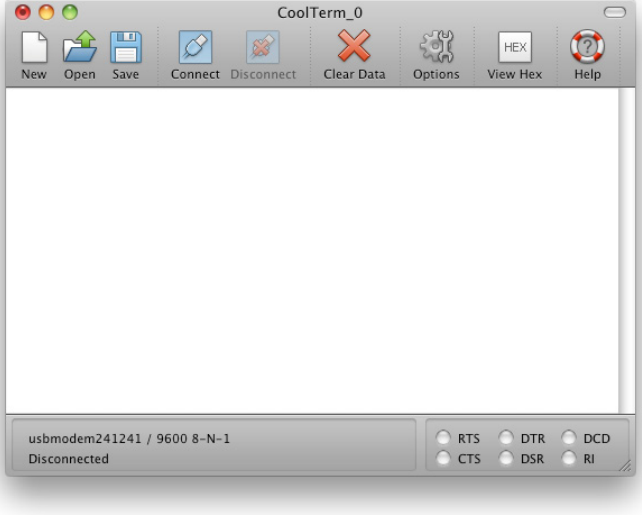


Figure 1-7
The CoolTerm serial terminal program.

The following projects have been tested extensively on Arduino boards and, when possible, on the classic Wiring board. Though you'll find some differences, code written for a Wiring board should work on an Arduino board, and vice versa. For Arduino derivatives, check with the manufacturer of your individual board. Many of them are very active in the Arduino forums and are happy to lend support.

You'll find that the editors for Arudino and Wiring look very similar. These free and open source programming environments are available through their respective websites: www.arduino.cc and www.wiring.org.co.

The hardware for both is also open source, and you can buy it from various online retailers, listed on the sites above. Or, if you're a hardcore hardware geek and like to make your own printed circuit boards, you can download the plans to do so. I recommend purchasing them online, as it's much quicker (and more reliable, for most people). Figure 1-8 shows some of your options.

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

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, visit their respective websites.

For an excellent introduction to Arduino, see Massimo Banzi's book **Getting Started with Arduino** (O'Reilly). **X**

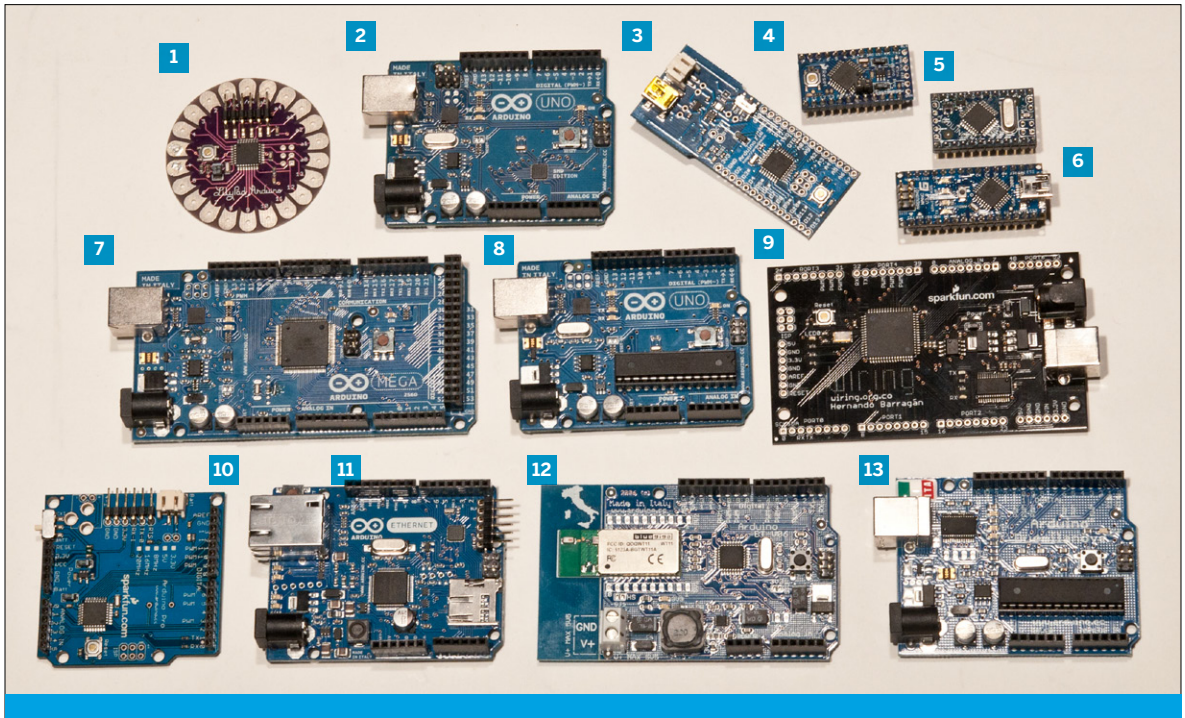


Figure 1-8. Varieties of Arduino, as well as a Wiring board: **1.** LilyPad Arduino **2.** Arduino Uno SMD **3.** Arduino Fio **4.** Arduino Pro Mini **5.** Arduino Mini **6.** Arduino Nano **7.** Arduino Mega 2560 **8.** Arduino Uno **9.** Wiring board **10.** Arduino Pro **11.** Arduino Ethernet **12.** Arduino Bluetooth **13.** Arduino Duemilanove.

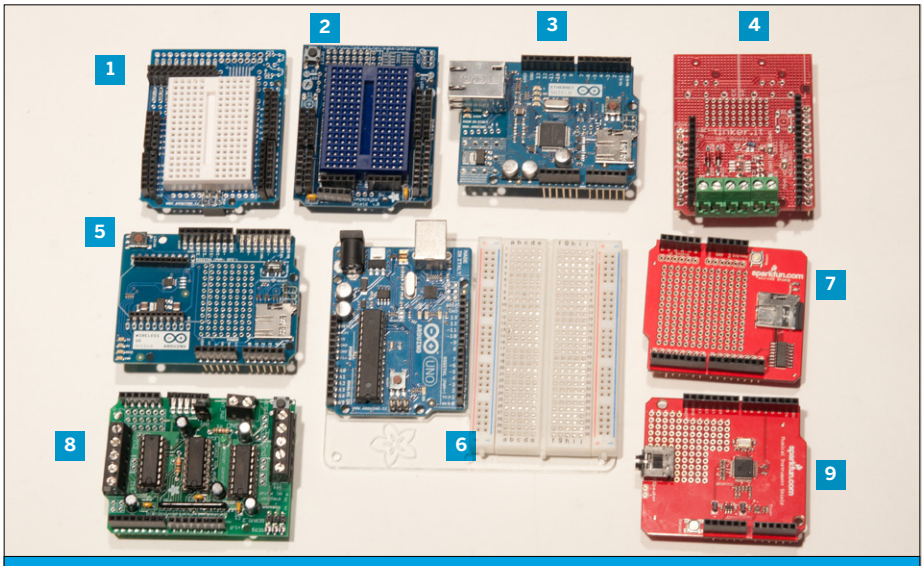


Figure 1-9. A sampling of shields for Arduino: **1.** Arduino prototyping shield **2.** Adafruit prototyping shield **3.** Arduino Ethernet shield **4.** TinkerKit DMX shield **5.** Arduino wireless shield **6.** Oomlout Arduino/breadboard mount, manufactured by Adafruit **7.** Spark Fun microSD card shield **8.** Adafruit motor driver shield **9.** Spark Fun musical instrument shield.

Arduino Shields

One of the features that makes Arduino easy to work with are the add-on modules called shields, which allow you to add preassembled circuits to the main module. For most applications you can think of, there's a third-party company or individual making a shield to do it. Need a MIDI synthesizer? There's a shield for that. Need NTSC video output? There's a shield for that. Need WiFi or Ethernet? There's a shield for that, and you'll be using them extensively in this book.

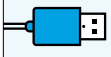
The growth of shields has been a major factor in the spread of Arduino, and the well-designed and documented ones make it possible to build many projects with no electronic experience whatsoever. You'll be using some shields in this book, and for other projects, building the actual circuit yourself.

The shields you'll see most commonly in this book are the Ethernet shield, which gives you the ability to connect your controller to the Internet; the wireless shield, which lets you interface with Digi's XBee radios and other radios with the same footprint; and some prototyping shields, which make it easy to design a custom circuit for your project.

The shield footprint, like the board designs, is available online at www.arduino.cc. If you've got experience making printed circuit boards, try your hand at making your own shield—it's fun.

Until recently, shields for Arduino weren't physically compatible with Wiring boards. However, Rogue Robotics (www.roguerobotics.com) just started selling an adapter for the Wiring board that allows it to take shields for Arduino.

Beware! Not every shield is compatible with every board. Some derivative boards do not operate on the same voltage as the Arduino boards, so they may not be compatible with shields designed to operate at 5 volts. If you're using a different microcontroller board, check with the manufacturer of your board to be sure it works with your shields.
X



Other Microcontrollers

Though the examples in this book focus on Arduino, there are many other microcontroller platforms that you can use to do the same work. Despite differences among the platforms, there are some principles that apply to them all. They're basically small computers. They communicate with the world by turning on or off the voltage on their output pins, or reading voltage changes on their input pins. Most microcontrollers can read variable voltage changes on a subset of their I/O pins. All microcontrollers can communicate with other computers using one or more forms of digital communication. Listed below are a few other microcontrollers on the market today.

8-bit controllers

The Atmel microcontrollers that are at the heart of both Arduino and Wiring are 8-bit controllers, meaning that they can process data and instructions in 8-bit chunks. 8-bit controllers are cheap and ubiquitous, and they can sense and control things in the physical world very effectively. They can sense simple physical characteristics at a resolution and speed that exceeds our senses. They show up in nearly every electronic device in your life, from your clock radio to your car to your refrigerator.

There are many other 8-bit controllers that are great for building physical devices. Parallax (www.parallax.com) Basic Stamp and Basic Stamp 2 (BS-2) are probably the most common microcontrollers in the hobbyist market. They are easy to use and include the same basic functions as Wiring and Arduino. However, the language they're programmed in, PBASIC, lacks the ability to pass parameters to functions, which makes programming many of the examples shown in this book more difficult. Revolution Education's PICAXE environment (www.rev-ed.co.uk) is very similar to the PBASIC of the Basic Stamp, but it's a less expensive way to get started than the Basic Stamp. Both the PICAXE and the Stamp are capable of doing the things shown in this book, but their limited programming language makes the doing a bit more tedious.

PIC and AVR

Microchip's PIC (www.microchip.com) and Atmel's AVR are excellent microcontrollers. You'll find the AVRs at the heart of Arduino and Wiring, and the PICs at the heart of the Basic Stamps and PICAXEs. The Basic Stamp, PICAXE, Wiring, and Arduino environments are essentially wrappers around these controllers, making them easier to work with. To use PICs or AVRs on their own, you need a hardware programmer that connects to your computer, and you need to install a programming environment and a compiler.

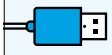
Though the microcontrollers themselves are cheap (between $1 and $10 apiece), getting all the tools set up for yourself can cost you some money. There's also a pretty significant time investment in getting set up, as the tools for programming these controllers from scratch assume a level of technical knowledge—both in software and hardware—that's higher than the other tools listed here.

32-bit controllers

Your personal computer is likely using a 64-bit processor, and your mobile phone is likely using a 32-bit processor. These processors are capable of more complex tasks, such as multitasking and media control and playback.

Initially, 32-bit processors were neither affordable nor easy to program, but that has been changing rapidly in the last couple of years, and there are now several 32-bit microcontroller platforms on the market. Texas Instruments' BeagleBoard (http://beagleboard.org) is a 32-bit processor board with almost everything you need to make a basic personal computer: HDMI video out, USB, SD card and connections for mass storage devices, and more. It can run a minimal version of the Linux operating system. Netduino (www.netduino.com) is a 32-bit processor designed to take Arduino shields, but it's programmed using an open source version of Microsoft's .NET programming framework. LeafLabs' Maple (http://leaflabs.com) is another 32-bit processor that uses the same footprint as the Arduino Uno, and is programmed in C/C++ like the Arduino and Wiring boards. In addition to these, there are several others coming on the market in the near future.

The increasing ease-of-use of 32-bit processors is bringing exciting changes for makers of physical interfaces, though not necessarily in basic input and output. 8-bit controllers can already sense simple physical



Other Microcontrollers (cont'd)

changes and control outputs at resolutions that exceed human perception. However, complex-sensing features—such as gesture recognition, multitasking, simpler memory management, and the ability to interface with devices using the same methods and libraries as personal computers—will make a big difference. 32-bit processors give physical interface makers the ability to use or convert code libraries and frameworks developed on servers and personal computers. There is where the real excitement of these processors lies.

These possibilities are just beginning to be realized and will easily fill another book, or several. However, basic sensing and networked communications are still well within the capabilities of 8-bit controllers, so I've chosen to keep the focus of this book on them.

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. 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've finished programming them. When you do, you'll power the board from the external power supply.

Figure 1-10 shows the inputs and outputs for the Arduino Uno. The other Arduino models and the Wiring module are similar. Each module has the same standard features as most microcontrollers: analog inputs, digital inputs and outputs, and power and ground connections. Some of the I/O pins can also be used for serial communication. Others can be used for pulse-width modulation (PWM), which is a way of creating a fake analog voltage by turning the pin on and off very fast. The Wiring and Arduino boards also have a USB connector that's connected to a USB-to-Serial controller, which allows the main controller to communicate with your computer serially over the USB port. They also have a programming header to allow you to reprogram the firmware (which you'll never do in this book) and a reset button. You'll see these diagrams repeated frequently, as they are the basis for all the microcontroller projects in the book.

Getting Started

Because the installation process for Wiring and Arduino is similar, I'll detail only the Arduino process here. Wiring users can follow along and do the same steps, substituting "Wiring" for "Arduino" in the instructions. Download the software from the appropriate site, then follow the instructions below. Check the sites for updates on these instructions.



Updates to the Arduino and Wiring software occur frequently. The notes in this book refer to Arduino version 1.0 and Wiring version 1.0. By the time you read this, the specifics may be slightly different, so check the Arduino and Wiring websites for the latest details.

Setup on Mac OS X

Double-click the downloaded file to unpack it, and you'll get a disk image that contains the Arduino application and an installer for FTDI USB-to-Serial drivers. Drag the application to your **Applications** directory. If you're using an Arduino Uno or newer board, you won't need the FTDI drivers, but if you're using a Duemilanove or older board, or a Wiring board, you'll need the drivers. Regardless of the board you have, there's no harm in installing them—even if you don't need them. Run the installer and follow the instructions to install the drivers.

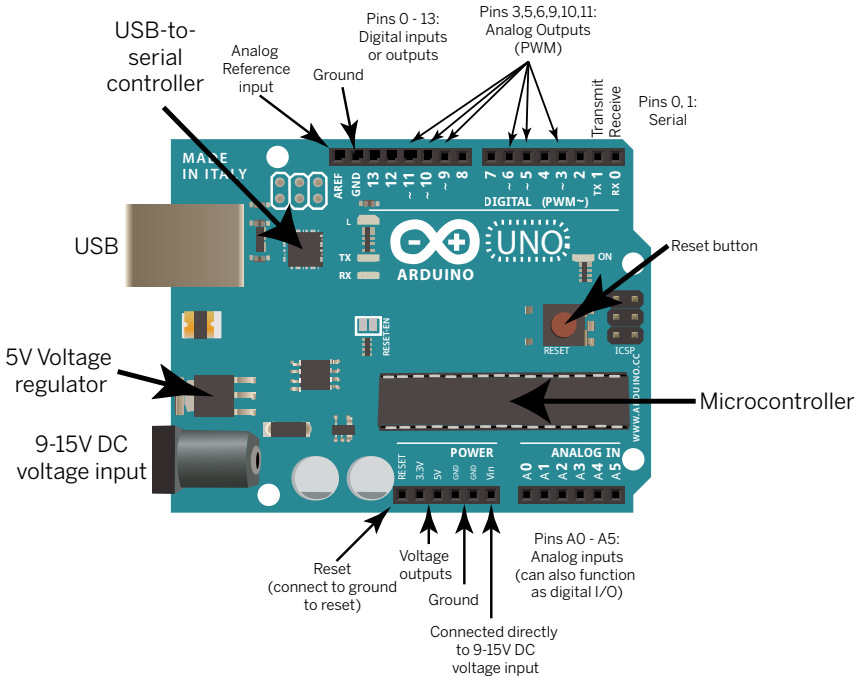


Figure 1-10
Functional parts of an Arduino. Most microcontrollers have the same or similar parts: power connections, digital and analog inputs, and serial communications.

Document What You Make

You'll see a lot of circuit diagrams in this book, as well as flowcharts of programs, system diagrams, and more. The projects you'll make with this book are systems with many parts, and you'll find it helps to keep diagrams of what's involved, which parts talk to which, and what protocols they use to communicate. I used three drawing tools heavily in this book, all of which I recommend for documenting your work:

Adobe Illustrator (www.adobe.com/products/illustrator.html). You really can't beat it for drawing things, even though it's expensive and takes time to learn well. There are many libraries of electronic schematic symbols freely available on the Web.

Inkscape (www.inkscape.org). This is an open source tool for vector drawing. Though the GUI is not as well developed as Illustrator, it's pretty darn good. The majority of the schematics in this book were done in Illustrator and Inkscape.

Fritzing (www.fritzing.org). Fritzing is an open source tool for documenting, sharing, teaching, and designing interactive electronic projects. It's a good tool for learning how to read schematics, because you can draw circuits as they physically look, and then have Fritzing generate a schematic of what you drew. Fritzing also has a good library of vector graphic electronics parts that can be used in other vector programs. This makes it easy to move from one program to another in order to take advantage of all three.

Figure 1-10 was cobbled together from all three tools, combining the work of Jody Culkin and Giorgio Olivero, with a few details from André Knörig and Jonathan Cohen's Fritzing drawings. You'll see it frequently throughout the book.

It's a good idea to keep notes on what you do as well, and share them publicly so others can learn from them. I rely on a combination of three note-taking tools: blogs powered by Wordpress (www.wordpress.org) at www.makingthingstalk.com, http://tigoe.net/blog, and http://tigoe.net/pcomp/code; a github repository (https://github.com/tigoe); and a stack of Maker's Notebooks (www.makershed.com, part no. 9780596519414).

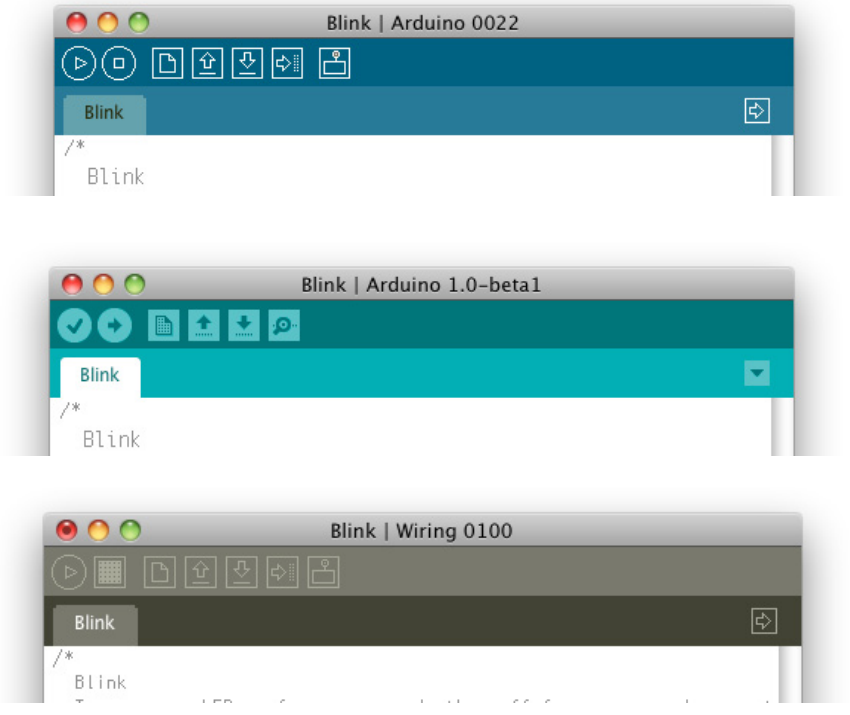


Figure 1-11
Toolbars for Arduino version 0022, Arduino 1.0, and Wiring 1.0.

Once you're installed, open the Arduino application and you're ready to go.

Setup on Windows 7

Unzip the downloaded file. It can go anywhere on your system. The **Program Files** directory is a good place. Next, you'll need to install drivers, whether you have an Arduino Uno board or an older board, or a Wiring board.

Plug in your Arduino and wait for Windows to begin its driver installation process. If it's a Duemilanove or earlier, it will need the FTDI drivers. These should install automatically over the Internet when you plug your Duemilanove in; if not, there is a copy in the **drivers** directory of the Arduino application directory. If it's an Uno or newer, click on the Start Menu and open up the Control Panel. Open the "System and Security" tab. Next, click on System, then open the Device Manager. Under Ports (COM & LPT), you should see a port named Arduino UNO (COMxx). Right-click on this port and choose the Update Driver Software option. Click the "Browse my computer for Driver software" option. Finally, navigate to and select the Uno's driver file, named **ArduinoUNO.inf**, located in the **drivers** directory. Windows will finish up the driver installation from there.

Setup on Linux

Arduino for Linux depends on the flavor of Linux you're using. See www.arduino.cc/playground/Learning/Linux for details on several Linux variants. For Ubuntu users, it's available from the Ubuntu Software Update tool.

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 New, Open, and Save buttons on the main toolbar. In Arduino and Wiring, the Run function is called Verify, and there is

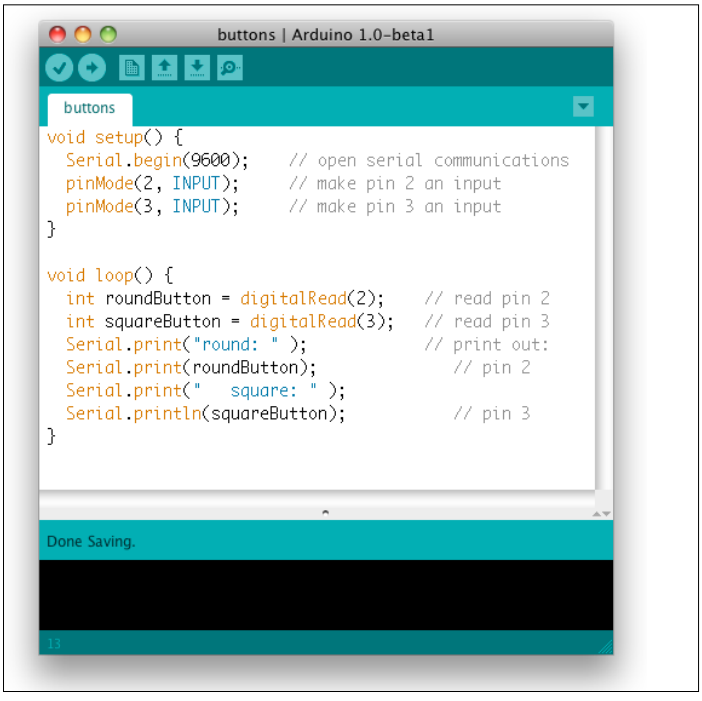


Figure 1-12
The Arduino programming environment. The Wiring environment looks similar, except the color is different.

an Upload button as well. Verify compiles your program to check for any errors, and Upload both compiles and uploads your code to the microcontroller module. There's an additional button, Serial Monitor, that you can use to receive serial data from the module while you're debugging.

Changes to version 1.0

For Arduino users familiar with previous versions, you'll see some changes in version 1.0. The toolbar has changed a bit. Figure 1-11 compares the toolbars of Arduino version 0022 (pre-1.0), Arduino 1.0, and Wiring 1.0. Arduino 1.0 now saves files with the extension .ino instead of .pde, to avoid conflict with Processing, which uses .pde. Wiring 1.0 still uses .pde. In addition, you can now upload sketches in Arduino 1.0 using an external hardware programmer. The Programmer submenu of the Tools menu lets you set your programmer.

X

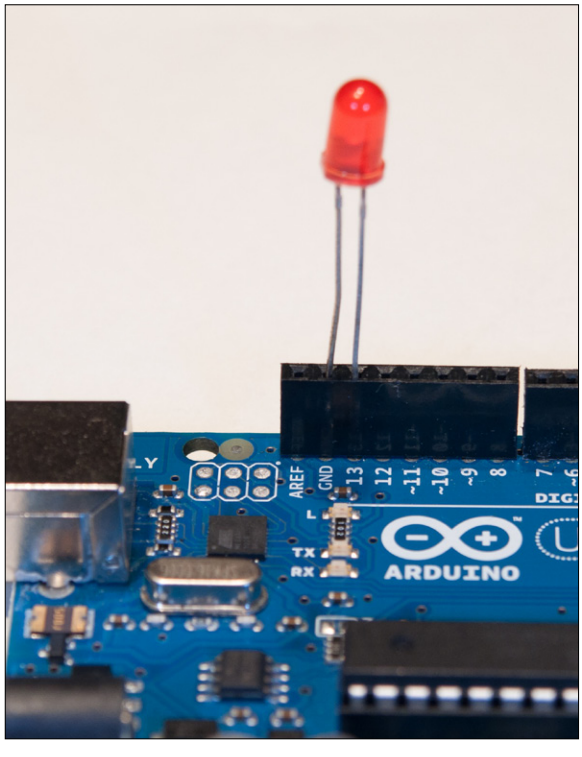


Figure 1-13
LED connected to pin 13 of an Arduino board. Add 220-ohm current-limiting resistor in series with this if you plan to run it for more than a few minutes.

Try It

Here's your first program.

```
/*  Blink
    Context: Arduino

    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→Board to choose your Arduino model, and then 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 this: /dev/tty.usbmodem241241. If it's an older board or a Wiring board, it will be more like this: /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).

NOTE: On Windows, COM1–COM4 are generally reserved for built-in serial ports, regardless of whether 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 click Upload. This will take a few seconds. Once it's done, you'll get a message saying Done uploading, and a confirmation message in the serial monitor window that says:

```
Binary sketch size: 1010 bytes (of a 32256 byte maximum)
```

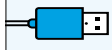
Once the sketch is uploaded, the LED you wired to the output pin will begin to blink. That's the microcontroller equivalent of "Hello World!"

NOTE: If it doesn't work, you might want to seek out some external help. The Arduino Learning section has many tutorials (www.arduino.cc/en/Tutorial). The Arduino (www.arduino.cc/forum) and Wiring (http://forum.wiring.co) 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 program listens for incoming serial data. It adds one to whatever serial value it receives, and then 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
    Context: Arduino
    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.write(inByte+1);
  }

  // Meanwhile, keep blinking the LED.
  // after a half 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.

Connecting 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.

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. 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.

Analog input

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 determines the voltage at this connection. 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 its conductive surface. 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.

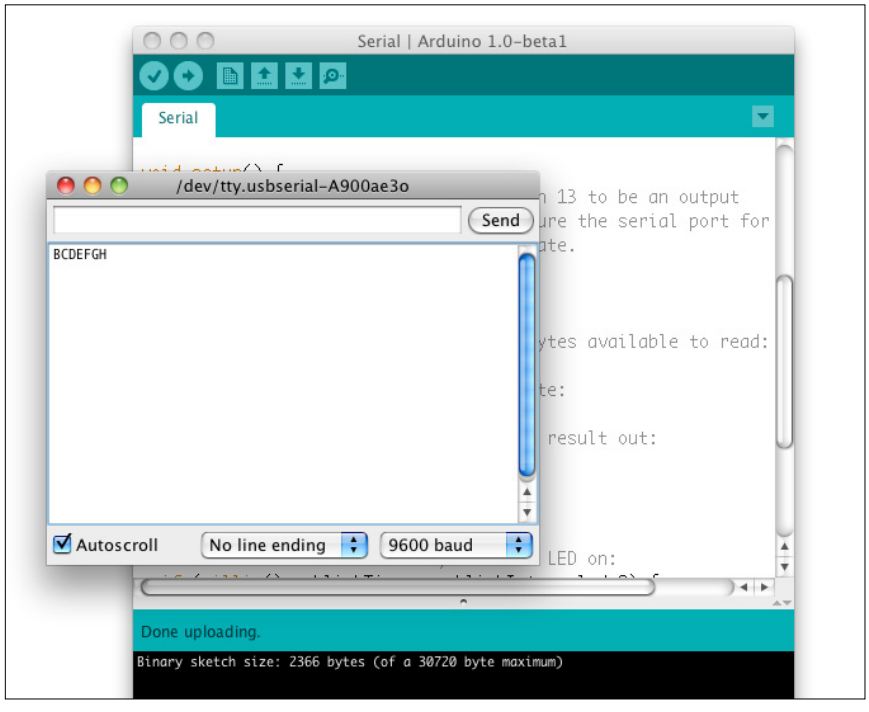


Figure 1-14
The Serial monitor in Arduino, running the previous sketch. The user typed BCDEFGH.

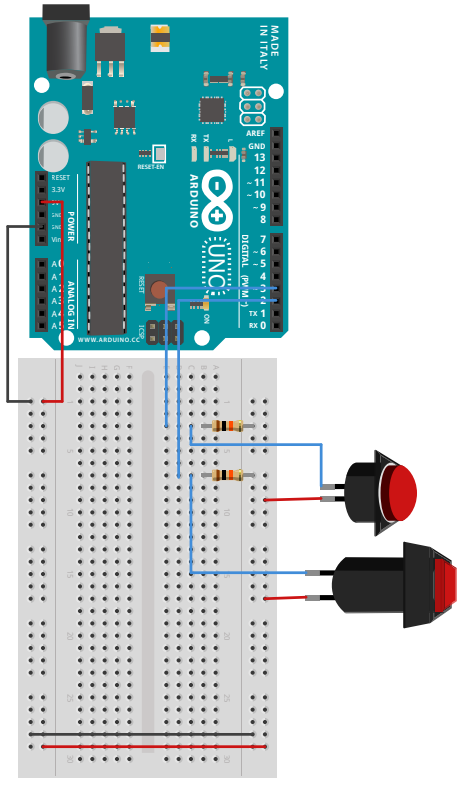


Figure 1-15
Arduino connected to a breadboard. +5V and ground run from the module to the long rows of the board. This way, all sensors and actuators can share the +5V and ground connections of the board. Control or signal connections from each sensor or actuator run to the appropriate I/O pins. In this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

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

Specialty circuits and modules

You'll see a number of specialty circuits and modules throughout this book, like the Bluetooth Mate and the XBee radios. These are devices that allow you to send serial data wirelessly. You'll also build a few of your own circuits for specific projects. All of the circuits will be shown on a breadboard like these, but you can build them any way you like. If you're familiar with working on printed circuit boards and prefer to build your circuits that way, feel free to do so.

Update to the Second Printing

As this edition's first printing went to press, the Arduino Ethernet and WiFi libraries were in transition. Since then, they have stabilized, and there may be some minor changes to the code examples you'll find for them. The TextFinder library mentioned in some examples has also been included in Arduino's core Stream class, with a slightly different interface. For up-to-date versions of the examples found here, see the GitHub repository for this book's code, at https://github.com/tigoe/MakingThingsTalk2.

x

You will encounter variations on many of the modules and components used in this book. For example, the Arduino module has several variations, as 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 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.

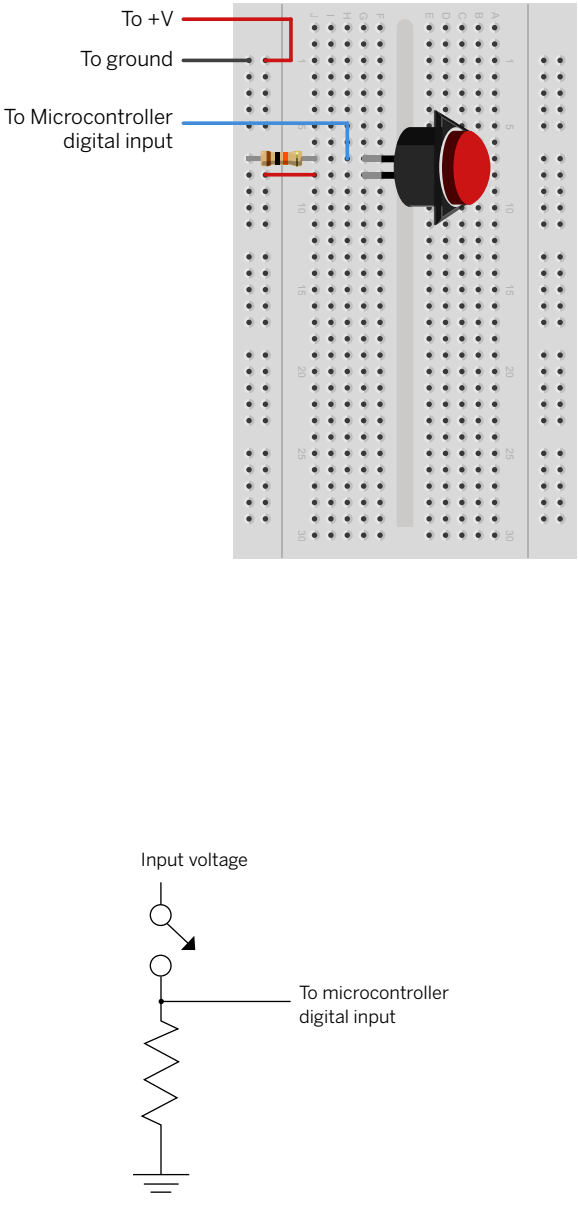
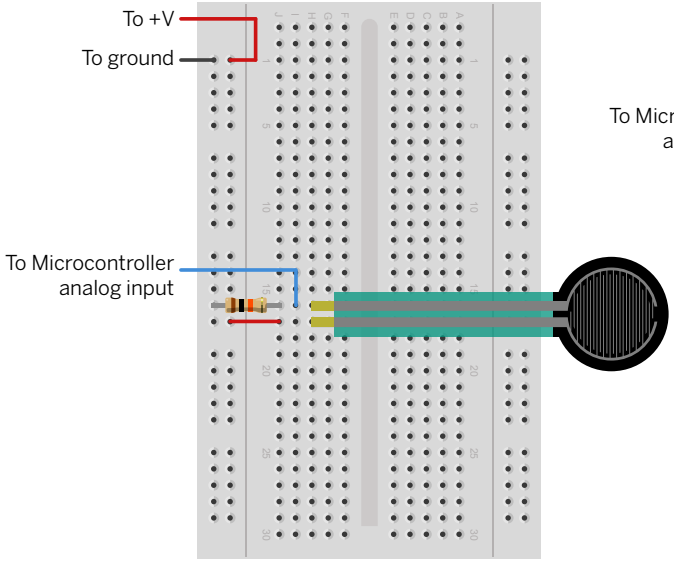


Figure 1-16
Digital input to a microcontroller.
Top: breadboard view.
Bottom: schematic view.



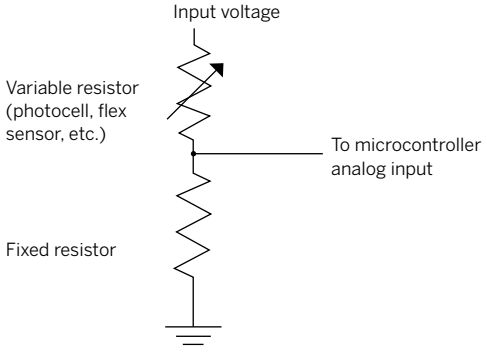
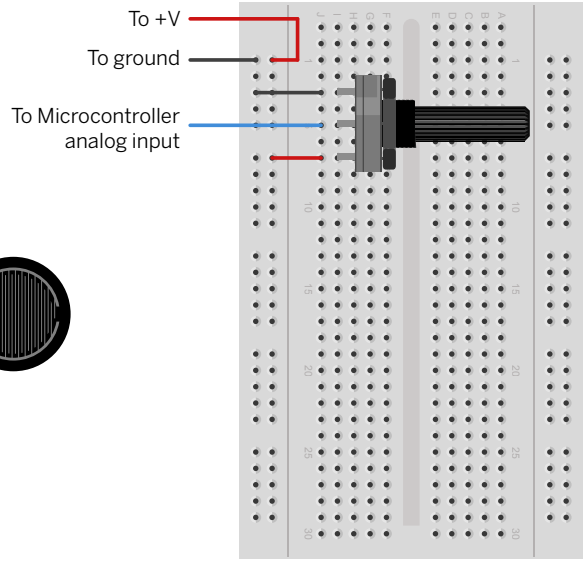


Figure 1-17
Voltage divider used as analog input to a microcontroller.
Top: breadboard view.
Bottom: schematic view.



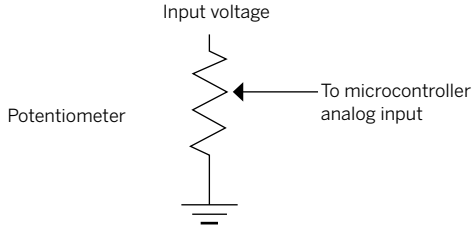


Figure 1-18
Potentiometer used as analog input to a microcontroller.
Top: breadboard view.
Bottom: schematic view.

Using an Oscilloscope

Most of what you'll be building in this book involves computer circuits that read a changing voltage over time. Whether your microcontroller is reading a digital or analog input, controlling the speed of a motor, or sending data to a personal computer, it's either reading a voltage or generating a voltage that changes over time. The time intervals it works in are much faster than yours. For example, the serial communication you just saw involved an electrical pulse changing at about 10,000 times per second. You can't see anything that fast on a multimeter. This is when an oscilloscope is useful.

An oscilloscope is a tool for viewing the changes in an electrical signal over time. You can change the sensitivity of its voltage reading (in volts per division of the screen) and of the time interval (in seconds, milliseconds, or microseconds per division) at which it reads. You can also change how it displays the signal. You can show it in real time, starting or stopping it as you need, or you can capture it when a particular voltage threshold (called a trigger) is crossed.

Oscilloscopes were once beyond the budget of most hobbyists, but lately, a number of inexpensive ones have come on the market. The DSO Nano from Seeed Studio, shown in Figure 1-19, is a good example. At about $100, it's a really good value if you're a dedicated electronic hobbyist. It doesn't have all the features that a full professional 'scope has, but it does give you the ability to change the volts per division and seconds per division, and to set a voltage trigger for taking a snapshot. It can sample up to 1 million times a second, which is more than enough to measure most serial applications. The image you see in Figure 1-19 shows the output of an Arduino sending the message "Hello World!" Each block represents one bit of data. The vertical axis is the voltage measurement, and the horizontal measurement is time. The Nano was sampling at 200 microseconds per division in this image, and 1 volt per division vertically. The 'scopes leads are attached to the ground pin of the Arduino and to digital pin 1, which is the serial transmit pin.

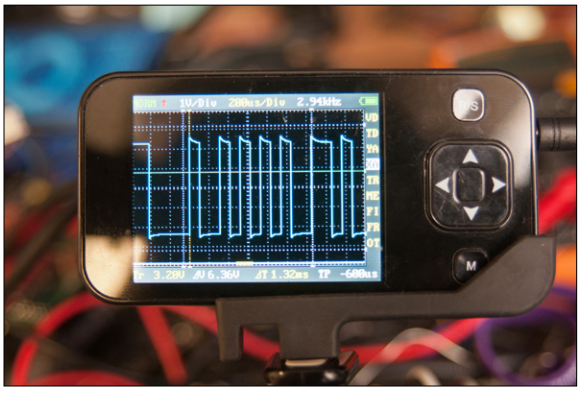


Figure 1-19
DSO Nano oscilloscope reading a serial data stream.

Besides inexpensive hardware 'scopes, there are also many software 'scopes available, both as freeware and as paid software. These typically use the audio input of your computer to sample the incoming voltage. The danger, of course, is that if you send in too much voltage you can damage your computer. For this reason, I prefer a hardware 'scope. But if you're interested in software 'scopes, a web search on software oscilloscope and your operating system will yield plenty of useful results.
X

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 him informed as to the task's progress. Physical objects don't have progress bars, but they should incorporate some indicator as to what they're doing—perhaps as simple as playing a tune or pulsing an LED gently while the network transfer's happening.

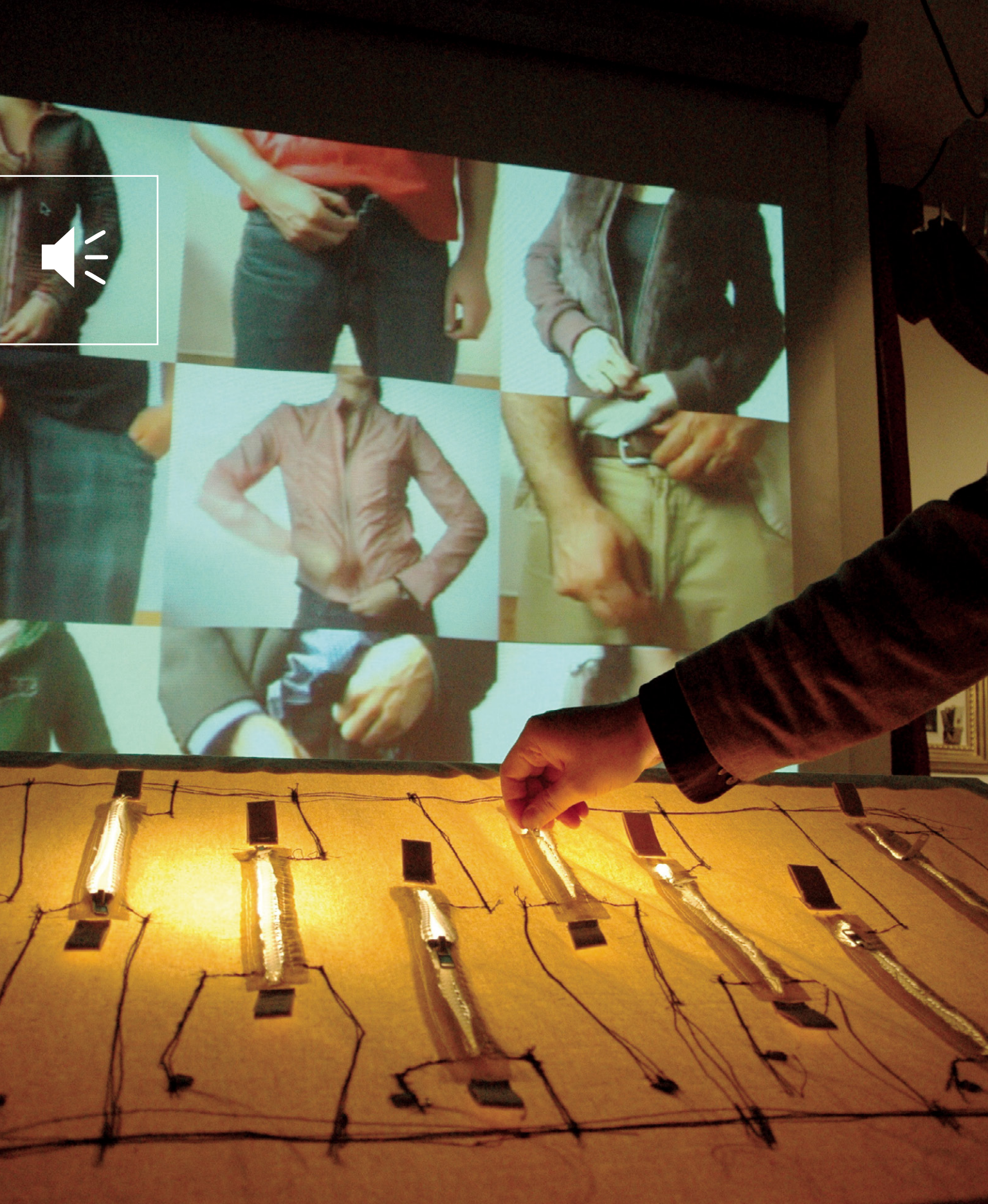
Find your own solution, but make sure you give some physical indication as to the invisible activities of your objects.

Don't forget the basic elements, either. Build in a power switch or a reset button. Include 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 sensibly. 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. 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 two examples using one-to-one serial communication between a microcontroller and a personal computer. You'll also learn about modem communications and how you can replace the cable connecting the two with Bluetooth radios.

◂◂ Joo Youn Paek's Zipper Orchestra (2006)
This 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.

Supplies for Chapter 2

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww/)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **RS** RS (www.rs-online.com)
- **SF** Spark Fun (www.SparkFun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 1: Type Brighter RGB LED Serial Control

» **1 Arduino module** Get something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards.
D 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 RGB LED, common cathode** It's like three LEDs in one! This emits red, green, and blue on separate channels.
D 754-1492-ND, **J** 2125181, **SF** COM-00105, **F** 8738661, **RS** 713-4888

» **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.

» **1 ping-pong ball** Get a white one from your local sports store.

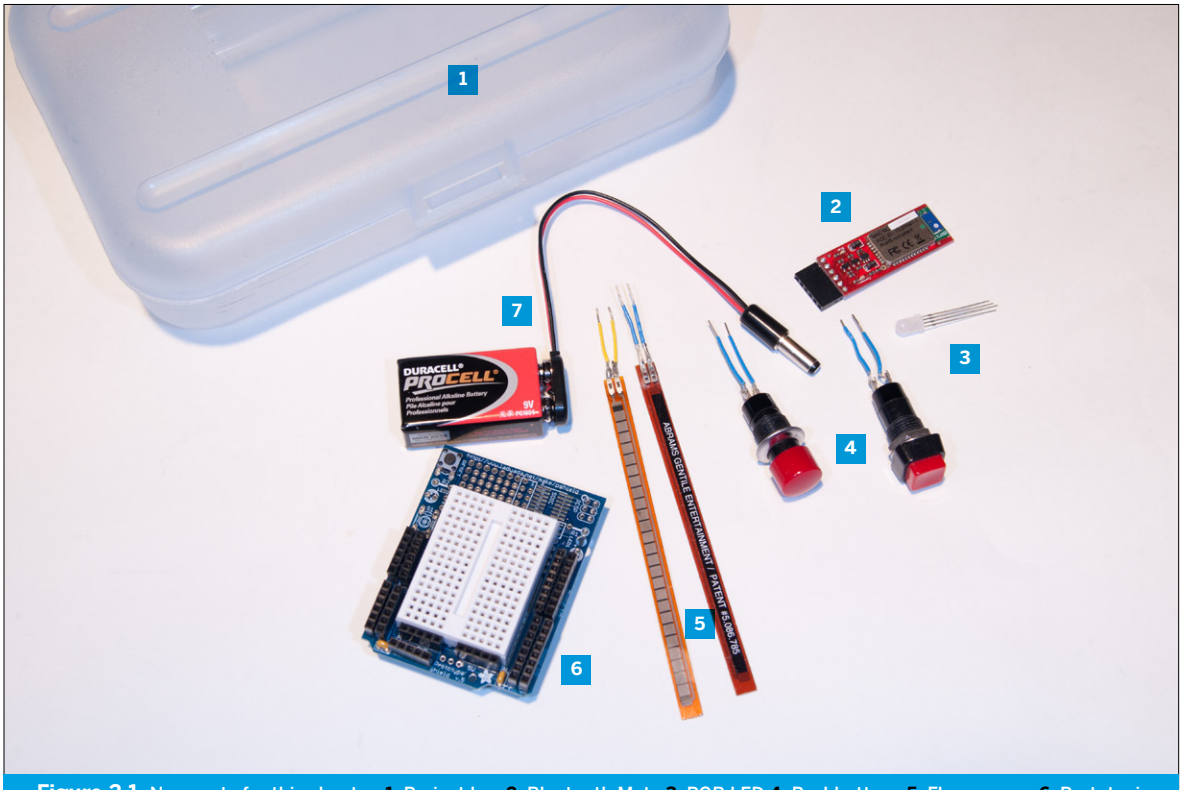


Figure 2-1. New parts for this chapter: **1.** Project box **2.** Bluetooth Mate **3.** RGB LED **4.** Pushbuttons **5.** Flex sensors **6.** Prototyping shield, 7. 9V battery snap connector and female power plug. Don't forget plenty of male header pins for the breakout boards.

PROJECT 2: Monski Pong

» **1 Arduino module** An Arduino Uno or something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards. **D** 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046,**AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **2 flex sensor resistors** **D** 905-1000-ND, **J** 150551, **SF** SEN-10264, **AF** 182, **RS** 708-1277, **MS** JM150551

» **2 momentary switches** Available from any electronics retailer. Pick the one that makes you the happiest. **D** GH1344-ND, **J** 315432, **SF** COM-09337, **F** 1634684, **RS** 718-2213, **MS** JM315432

» **4 10-kilohm resistors** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906

» **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

» **1 personal computer**

» **All necessary converters to communicate serially from microcontroller to computer** Just like the previous project.

» **1 small pink monkey** aka Monski. You may want a second one for a two-player game.

PROJECT 3: Wireless Monski Pong

» **1 completed Monski pong project** from project 2.

» **1 9V battery and snap connector** **D** 2238K-ND, **J** 101470, **SF** PRT-09518, **F** 1650675

» **Female power plug, 2.1mm ID, 5.5mm OD** If you got the Spark Fun battery connector, you don't need this part. **D** CP-024A-ND, **J** 159506, **F** 1737256

» **1 Bluetooth Mate module** **SF** WRL-09358 or WRL-10393

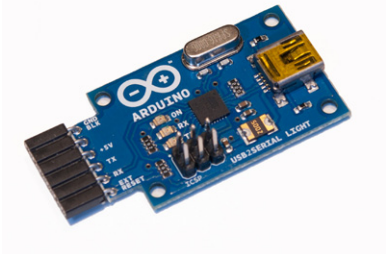
» **1 project box** to house the microcontroller, battery, and radio board.

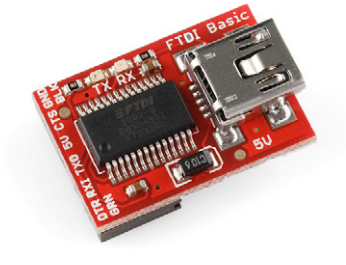
PROJECT 4: Negotiating in Bluetooth

» **1 Bluetooth Mate module** the same one used in project 3. **SF** WRL-09358 or WRL-10393

» **1 FTDI-style USB-to-Serial adapter** Both the 5V or 3.3V versions will work; these come as cables or standalone modules. **SF** DEV-09718 or DEV-09716, **AF** 70 or 284, **A** A000059, **MS** MKAD22, **D** TTL-232R-3V3 or TTL-232R-5V







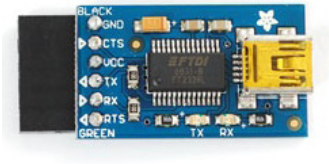


Figure 2-2
FTDI USB-to-TTL cable. This cable comes in handy for connecting to all sorts of serial devices. When plugged into a USB port, it can even provide power for the device with which it's communicating. The adapter boards shown here also act as USB-to-Serial adapters and can replace this cable. L to R: Arduino USB-to-Serial; Spark Fun FTDI Basic Breakout; Adafruit FTDI Friend. They require an additional USB-A-to-Mini-B cable (the kind that comes with most digital cameras), but have built-in LEDs attached to the transmit and receive lines. These LEDs flash when data's going through those lines, and they can be handy for troubleshooting serial problems.

The Arduino adapter, unlike the others, does not use the FTDI drivers. On Mac OS X and Linux, it needs no drivers. On Windows, it uses the same INF file as the Arduino Uno. *Photos courtesy of Spark Fun and Adafruit.*

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 others? 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 0 or a 1? When high voltage represents 1 and low voltage represents 0, it's called true logic. When it's reversed—high voltage represents 0 and low voltage represents 1—it's called inverted logic. You'll see examples of both in the pages that follow.

- **Data**
What's the timing of the bits? Are the bits read in groups of 8, 9, or 10? 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 for something to get 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 2-3) that the 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). 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.

X

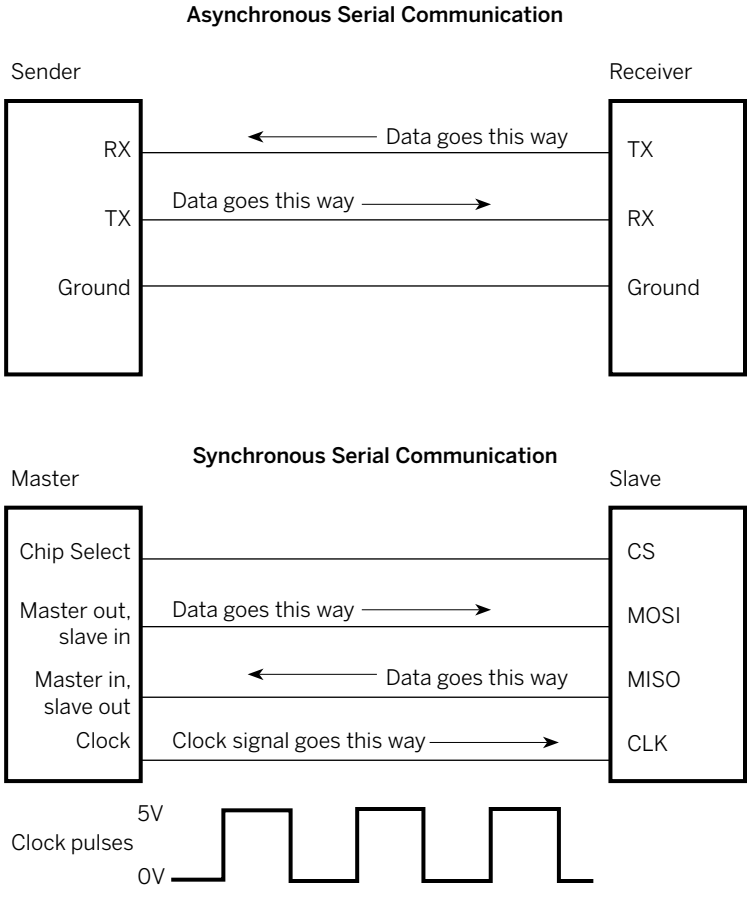


Figure 2-3
Types of serial communication.

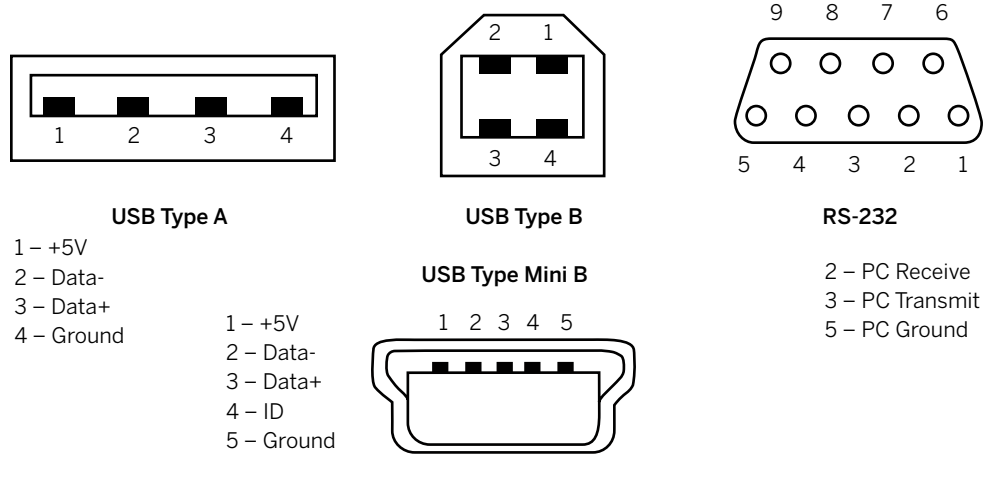


Figure 2-4
Physical connections: USB, RS-232 serial.

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've worked with a different microcontroller, such as Parallax's Basic Stamp, you probably made the connection using a serial-to-USB converter or an older PC that still had a 9-pin RS-232 serial port. That simple connection involved two serial protocols.

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

- **Physical layer**
What pins is the controller using to communicate? 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. Some microcontrollers use 3.3 volts instead of 5 volts.

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

- **Data layer**
Data is typically 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 this 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, Data+ and Data-. Every USB connector also has a 5-volt power supply line and a ground line.

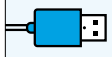
- **Electrical layer**
The signal on Data – is always the polar opposite of what's on Data+, so the sum of their voltages is always zero. Because of this, a receiver can check for electrical errors by adding the two data voltages together. If the sum isn't zero, the receiver can disregard the signal 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 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-to-Serial chip on your Arduino board presents itself to the operating system as a serial port, and it sends data through the USB connection at the rate you choose (9600 bits per second in the example in Chapter 1).



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 adapter 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.usbmodem241441
/dev/tty.usbmodem241461
/dev/tty.usbmodem241471
```

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

The Arduino boards come with their own USB-to-Serial adapter on board, but other microcontrollers and devices usually don't. You can buy a USB-to-Serial converter for about $15 to $40—Keyspan (www.keyspan.com) and IOGear (www.iogear.com) sell decent models. Most consumer models like these are USB-to-RS-232, because RS-232 was the standard serial connector for PCs before USB came along. USB-to-RS-232 adapters won't work directly with TTL serial devices.

FTDI makes a USB-to-TTL-Serial cable with a breadboard connector that's handy for interfacing to TTL serial devices. It's available from the Maker SHED, Spark Fun, Adafruit, and many other vendors. It comes in 5-volt and 3.3-volt versions. Arduino, Spark Fun, and Adafruit all make breakout boards with the same cable pin connections. Spark Fun's FTDI Basic Board has LEDs that flash when data is being transmitted. Adafruit's version is called the FTDI Friend. The Arduino USB-to-Serial adapter is based on a different chip but has the same pin connections. Any of these can be used for USB-to-Serial connections throughout this book. You can see them all at the beginning of this chapter. The pin connections are shown in Figure 2-6.

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 adapter. You can see it in Figure 2-4. This connector, called a DB-9 or D-sub-9, 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 seen on some older 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. This is inverted logic.

- **Data layer**
  This is the same as TTL—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 mostly obsolete, so USB-to-Serial converters are increasingly common tools for microcontroller programmers. Because the Arduino boards have an integrated USB-to-Serial converter, you can just plug them into a USB port.

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

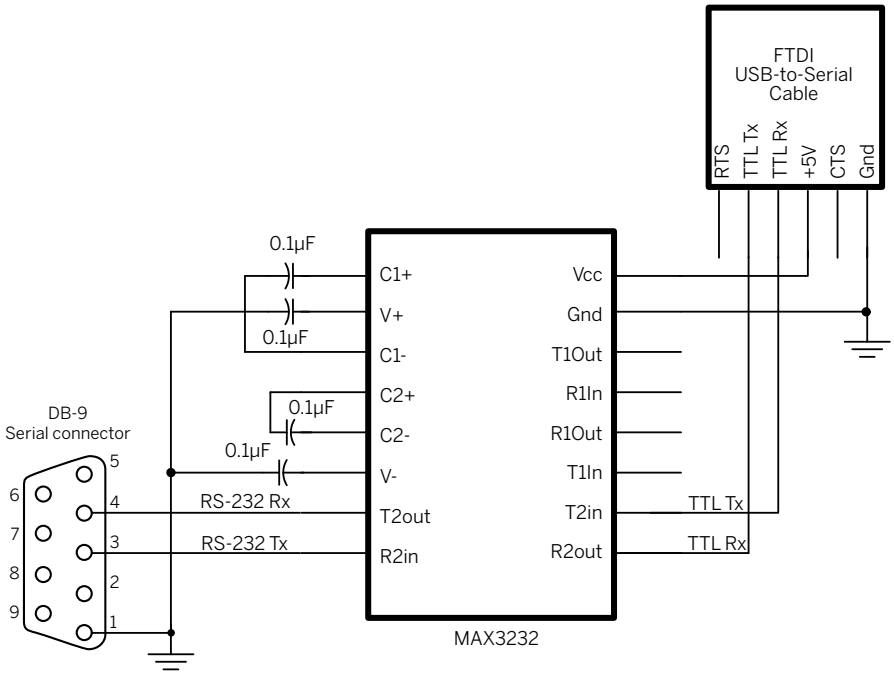
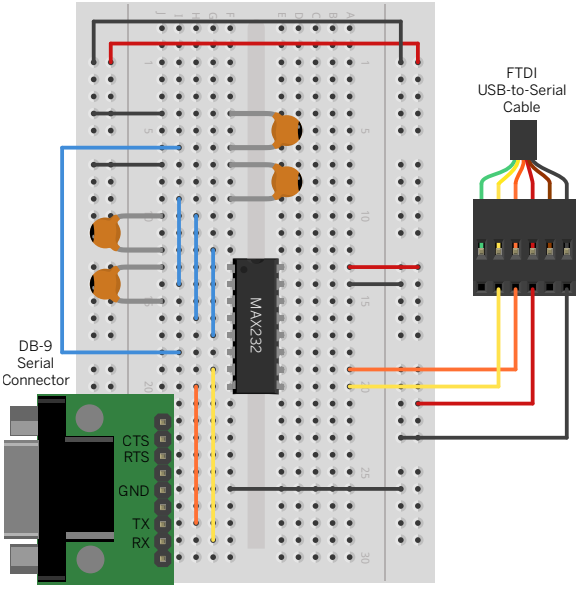


Figure 2-5
The MAX3232 chip. This circuit is really handy when you need to get any 3.3- to 5-volt TTL device to talk to a personal computer with an RS-232 serial port. This will also work for the MAX232.



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 MAX3232, available from Maxim Technologies (www.maxim-ic.com). It takes in RS-232 serial and spits out 3.3-V 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 2-5 shows the typical schematic for a MAX3232.

If you've done a lot of serial projects, you may know the MAX232, which preceded the MAX3232. In fact, the MAX232 was so common that the name became synonymous with all TTL-to-RS-232 converters, whether Maxim made them or not. The MAX232 worked only at 5 volts, but the MAX 3232 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



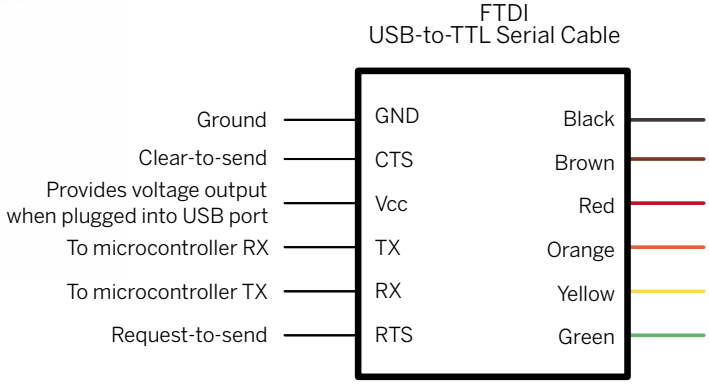
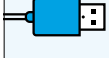


Figure 2-6
FTDI USB-to-TTL cable. In addition to the transmit, receive, voltage, and ground connections, it also has connections for hardware flow control, labeled Request to Send (RTS) and Clear to Send (CTS). Some devices use this to manage the flow of serial data.



Using an Arduino As a USB-to-Serial Adapter

Arduino boards have a built-in USB-to-serial adapter so that the microcontroller can communicate with your personal computer serially. The USB-to-Serial adapter's transmit pin (TX) is attached to the microcontroller's receive pin (RX), and vice versa. This means you can bypass the microcontroller and use the Arduino board as a USB-to-Serial adapter. There are times when this is handy, like when you want to communicate directly with the Bluetooth radios you'll see later in this chapter. To do this, either remove the microcontroller carefully, or put a sketch on it that does nothing, like so:

```
void setup() {}
void loop() {}
```

Then connect the desired external serial device as follows:

External serial device's receive pin → Pin 0 of the Arduino board

External serial device's transmit pin → Pin 1 of the Arduino board

Now your serial device will communicate directly with the USB-to-Serial adapter on the Arduino, and the microcontroller will be bypassed. When you're ready to use the microcontroller again, just disconnect the external serial device and upload a new sketch.

Saying Something: The Application Layer

Now that you've got a sense of how to make the connections between devices, it's time to build a couple of projects to understand how to organize the data you send.

Project 1

Type Brighter

In this example, you'll control the output of a microcontroller with key-strokes from your computer. It's a very simple example with minimal parts, so you can focus on the communication.

MATERIALS

- **1 RGB LED, common cathode**
- **1 Arduino microcontroller module**
- **1 personal computer**
- **All necessary converters to communicate serially from microcontroller to computer**
- **1 ping-pong ball**

Every application needs a communications protocol, no matter how simple it is. Even turning on a light requires that both the sender and receiver agree on how to say "turn on the light."

For this project, you'll make a tiny colored lamp. You'll be able to control the brightness and color of the lamp by sending it commands from your computer. The RGB LED at the heart of the lamp is actually three LEDs in one package: one red, one green, and one blue. They share a common cathode, or negative terminal. Connect the cathode, which is the longest leg, to the ground of your Arduino module, and connect the three other legs (called the anodes, one for each color) to pins 9, 10, and 11, as shown in Figure 2-7. Bend the cathode leg so it will fit, but make sure it's not touching the last leg, or it will create a short circuit.

When you've got the LED on the board, drill a small hole in the pong-pong ball, slightly larger than the LED. Fit the ball over the LED, as shown in Figure 2-10. It will act as a nice lampshade. If the LEDs form too harsh a spot on the ball, you can diffuse them slightly by sanding the top of the LED case.

The Protocol

Now that you've got the circuit wired up, you need to decide how you're going to communicate with the microcontroller to control the LEDs. You need a communications protocol. This one will be very simple:

- To choose a color of LED to control, send the first letter of the color, in lower-case (r, g, b).
- To set the brightness for that color, send a single digit, 0 through 9.

For example, to set red at 5, green at 3, and blue at (on a scale from 0 to 9), you'd send:

```
r5g3b7
```

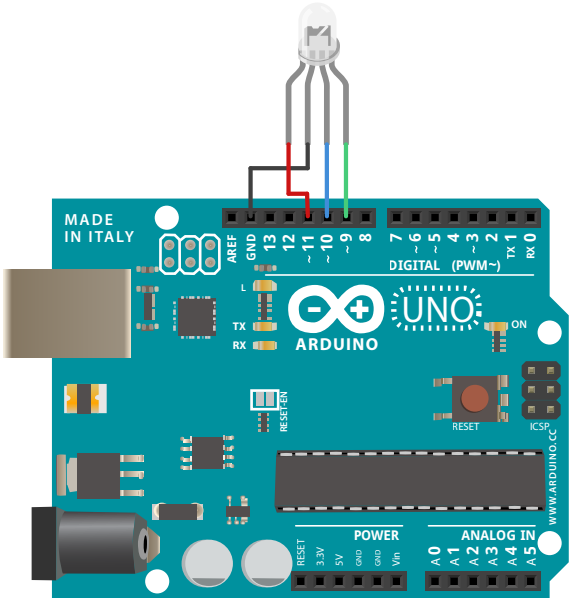
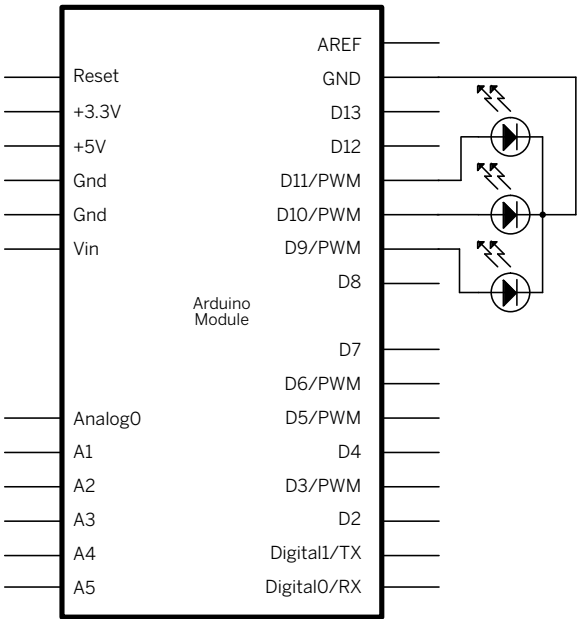


Figure 2-7
RGB LED attached to pins 9, 10, 11, and ground. The ground leg is bent back so that it doesn't touch the leg connected to pin 11, which it crosses. In theory, this LED should have a resistor on each of the anodes to limit the current going through the LEDs. In practice, however, I've run this for several hours without the resistor with no damage to the Arduino or LED. When in doubt, be safe and add a 220-ohm resistor to each anode.



That's about as simple a protocol as you could imagine. However, you still need to write a program for the microcontroller to read the data one byte at a time, and then decide what to do based on the value of each byte.

The program flow is shown in Figure 2-8.

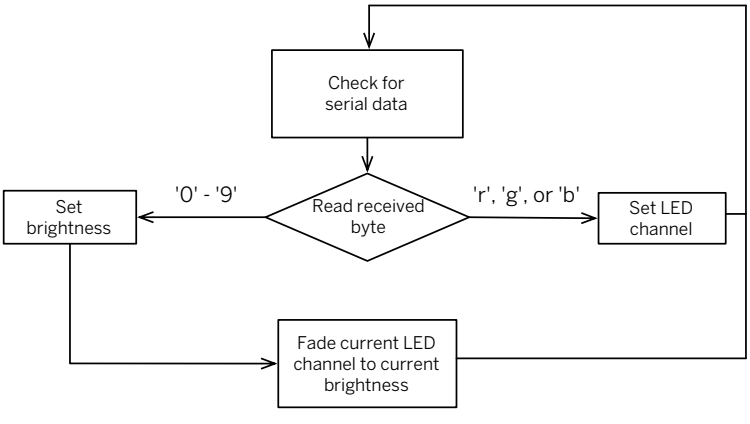


Figure 2-8
Flowchart for the program.

Try It

```
/*
  Serial RGB LED controller
  Context: Arduino

  Controls an RGB LED whose R, G and B legs are
  connected to pins 11, 9, and 10, respectively.
*/
```

» To start, you need to set up the constants to hold the pin numbers. You'll also need a variable to hold the number of the current pin to be faded, and one for the brightness.

```
// constants to hold the output pin numbers:
const int greenPin = 9;
const int bluePin = 10;
const int redPin = 11;

int currentPin = 0; // current pin to be faded
int brightness = 0; // current brightness level
```

» The setup() method opens serial communications and initializes the LED pins as outputs.

```
void setup() {
  // initiate serial communication:
  Serial.begin(9600);

  // initialize the LED pins as outputs:
  pinMode(redPin, OUTPUT);
  pinMode(greenPin, OUTPUT);
  pinMode(bluePin, OUTPUT);
}
```

» In the main loop, everything depends on whether you have incoming serial bytes to read.

```
void loop() {
  // if there's any serial data in the buffer, read a byte:
  if (Serial.available() > 0) {
    int inByte = Serial.read();

    // respond to the values 'r', 'g', 'b', or '0' through '9'.
```

» If you do have incoming data, there are only a few values you care about. When you get one of the values you want, use if statements to set the pin number and brightness value.

```
    // you don't care about any other value:
    if (inByte == 'r') {
      currentPin = redPin;
    }
    if (inByte == 'g') {
      currentPin = greenPin;
    }
    if (inByte == 'b') {
      currentPin = bluePin;
    }
```

» Finally, set the current pin to the current brightness level using the analogWrite() command.

```
   if (inByte >= '0' && inByte <= '9') {
      // map the incoming byte value to the range of
      // the analogRead() command:
      brightness = map(inByte, '0', '9', 0, 255);
      // set the current pin to the current brightness:
      analogWrite(currentPin, brightness);
    }
  }
}
```

Upload the sketch to your Arduino, then open the Serial Monitor by clicking on the icon at the top of the editor, as shown in Figure 2-9.

Once it's open, type:

```
r9
```

Then click send. You should see the ball light up red. Now try:

```
r2g7
```

The red will fade and the green will come up. Now try:

```
g0r0b8
```

The blue will come on. Voila, you've made a tiny serially controllable lamp!

If the colors don't correspond to the color you type, you probably bought an LED of a different model than the one specified above, so the pin numbers are different. You can fix this by changing the pin number constants in your sketch.

You don't have to control the lamp from the Serial Monitor. Any program that can control the serial port can send your protocol to the Arduino to control the lamp. Once you've finished the next project, try writing your own lamp controller in Processing.

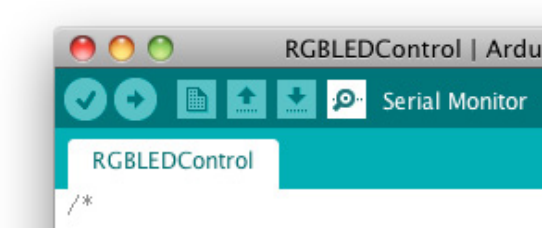
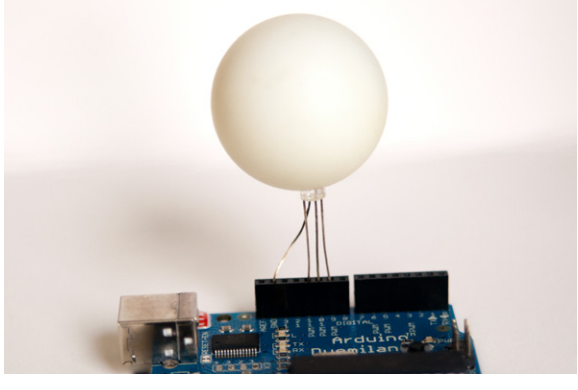


Figure 2-9
Arduino toolbar, highlighting the Serial Monitor.

Figure 2-10
LED with a ping-pong ball on top to diffuse the light.



Notice in the program how the characters you type are in single quotes? That's because you're using the ASCII values for those characters. ASCII is a protocol that assigns numeric values to letters and numbers. For example, the ASCII value for the letter 'r' is 114. The ASCII value for '0' is 48. By putting the characters in single quotes, you're programming the Arduino to use the ASCII value for that character, not the character itself. For example, this line:

```
brightness = map(inByte, '0', '9', 0, 255);
```

could also be written like this:

```
brightness = map(inByte, 48, 57, 0, 255);
```

because in ASCII, the character '0' is represented by the value 48, and the character '9' is represented by the value 57. Using the character value in single quotes instead of the actual values isn't essential to make your program run, but it makes it easier to read. In the first version of the line above, you're using the ASCII characters to represent the values to map; in the second version, you're using the raw values themselves. You'll see examples in this book that use both approaches. For more on ASCII, see "What's ASCII?" on page 54.

Complex Conversations

In the previous project, you controlled the microcontroller from the computer using a very simple protocol. This time, the microcontroller will control an animation on the computer. The communications protocol is more complex as well.

Project 2

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 2-11.

MATERIALS

- **2 flex sensor resistors**
- **2 momentary switches**
- **4 10-kilohm resistors**
- **1 solderless breadboard**
- **1 Arduino microcontroller module**
- **1 personal computer**
- **All necessary converters to communicate serially from microcontroller to computer**
- **1 small pink monkey**

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. Use flexible wire; old telephone cable works well. 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 2-12.

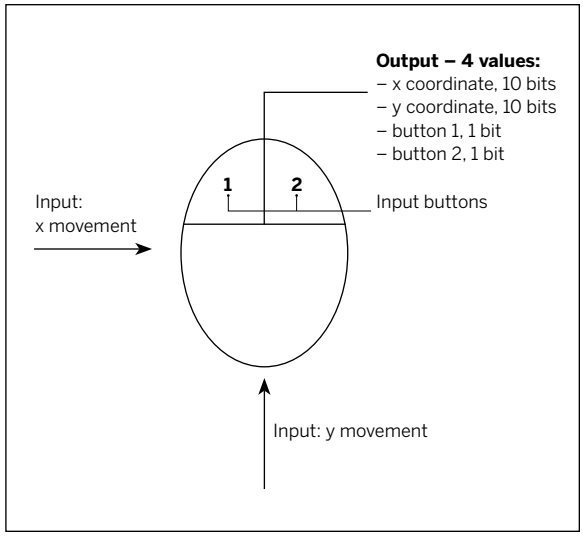


Figure 2-11
The mouse as a data object.

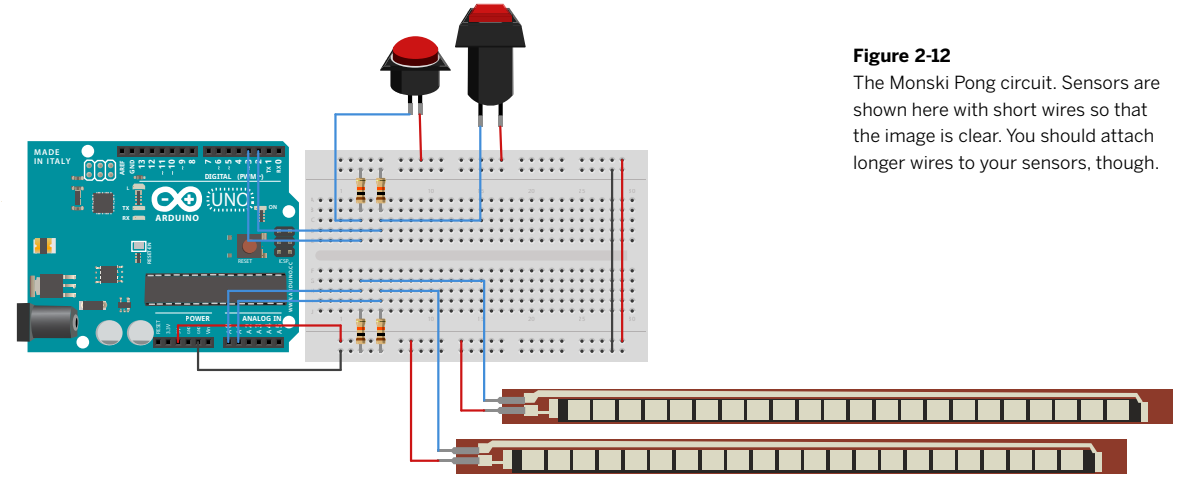
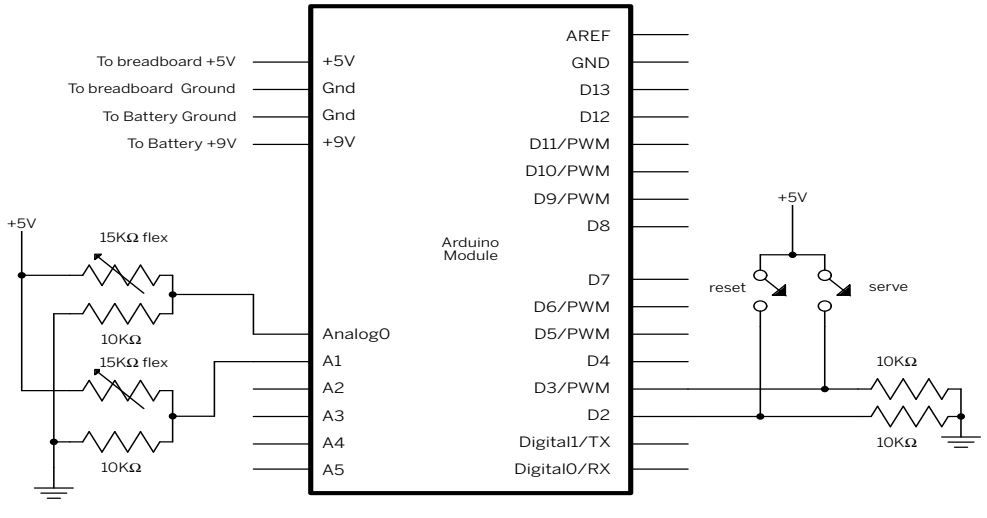


Figure 2-12
The Monski Pong circuit. Sensors are shown here with short wires so that the image is clear. You should attach longer wires to your sensors, though.



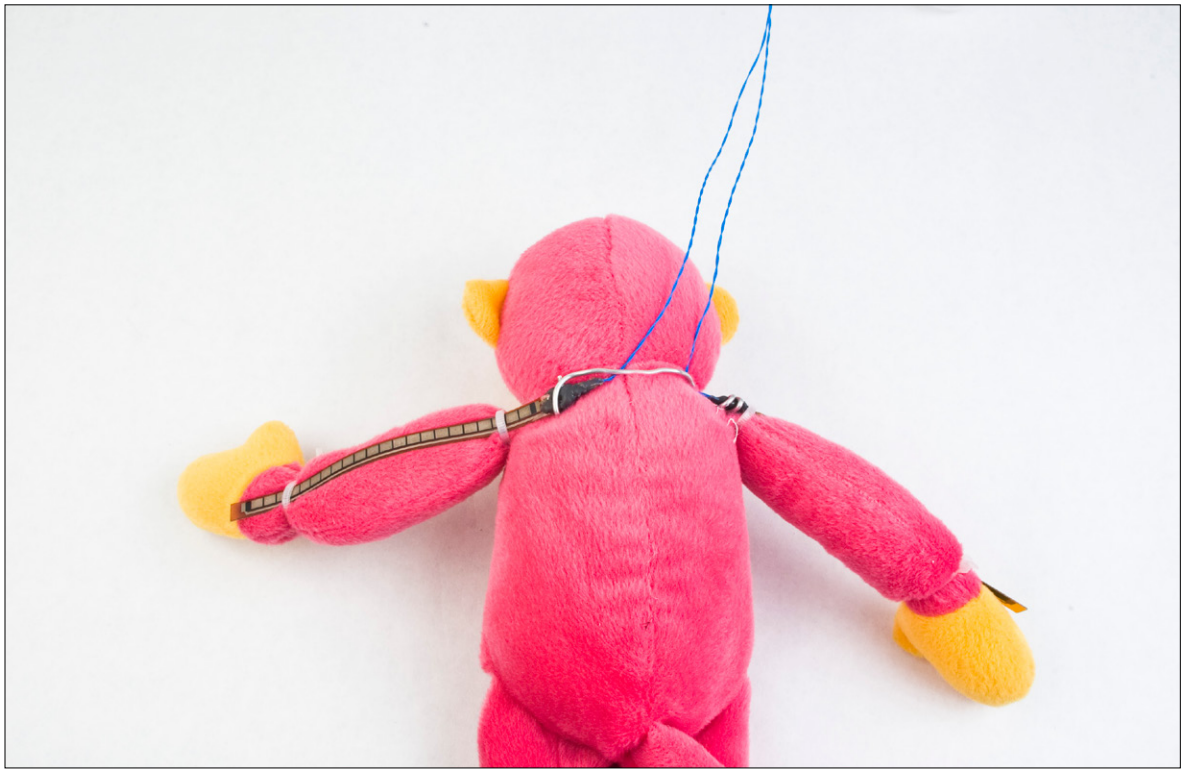


Figure 2-13
A stable support for the sensors is essential if you want good readings from them. Once you know your support works, move it inside the monkey and test it.

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, as shown in Figure 2-13. Position the sensors so that their movement is consistent. 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 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 work well.

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,
```

Just as you programmed it, each value is separated by a comma, and each set of readings is on a line by itself.

```
/*
   Sensor Reader
   Context: 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


 */

const int leftSensor = 0;    // analog input for the left arm
const int rightSensor = 1;   // analog input for tht right arm
const int resetButton = 2;   // digital input for the reset button
const 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);
  Serial.print(",");
  Serial.print(rightValue);
  Serial.print(",");
  Serial.print(reset);
  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);
}
```

Try replacing the part of your code that prints the results with the code to the right.

When you view the results in the Serial Monitor or terminal, you'll get something that looks like this:

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

```
// print the results:
Serial.write(leftValue);
Serial.write(44);
Serial.write(rightValue);
Serial.write(44);
Serial.write(reset);
Serial.write(44);
// print the last sensor value with a println() so that
// each set of four readings prints on a line by itself:
Serial.write(10);
Serial.write(13);
Serial.write(serve);
```

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

What's going on? The original example uses the Serial.print() command, which displays the values of the sensors as their ASCII code values: this modification sends out the raw binary values using Serial.write(). The Serial Monitor and serial terminal applications 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. 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

What's ASCII?

ASCII is the American Symbolic Code for Information Interchange. The scheme 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 converted to a string of numbers, transmitted, and 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 the display format of the message could be transmitted along with the text. These are referred to as control characters, and they take up the 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. However, ASCII is too limited to display non-English characters, 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 ASCII code still lingers on.

protocols you'll use in this book are ASCII-based. In addition, ASCII is readable by humans, so you may find it easier to send the data as ASCII. For Monski Pong, use the ASCII-formatted version (the first example); later in this chapter you'll see why it's the right choice.

If you haven't already done so, undo the changes you made on page 54 to the Sensor Reader program 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 Arduino board. Processing will do this well.

Try It Open the Processing application and enter this code.

```
/*
 Serial String Reader
 Context: Processing

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

Serial myPort;                  // The serial port
String resultString;            // string for the results

void setup() {
  size(480, 130);              // set the size of the applet window
  println(Serial.list());      // List all the available serial ports

  // get the name of your port from the serial list.
  // The first port in the serial list on my computer
  // is generally my Arduino module, so I open Serial.list()[0].
  // Change the 0 to the number of the serial port
  // to which your microcontroller is attached:
  String portName = Serial.list()[0];
  // open the serial port:
  myPort = new Serial(this, portName, 9600);

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil('\n');
}

void draw() {
  // set the background and fill color for the applet window:
  background(#044f6f);
  fill(#ffffff);
  // show a string in the window:
  if (resultString != null) {
    text(resultString, 10, height/2);
  }
}

// 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 inputString = myPort.readStringUntil('\n');
```

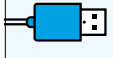


Continued from previous page.

```
  // trim the carrige return and linefeed from the input string:
  inputString = trim(inputString);
  // clear the resultString:
  resultString = "";

  // split the input string at the commas
  // and convert the sections into integers:
  int sensors[] = int(split(inputString, ','));

  // add the values to the result string:
  for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++) {
    resultString += "Sensor " + sensorNum + ": ";
    resultString += sensors[sensorNum] + "\t";
  }
  // print the results to the console:
  println(resultString);
}
```



Data Packets, Headers, Payloads, and Tails

Now that you've got data going from one device (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 (with a comma, ASCII 44, between each field):

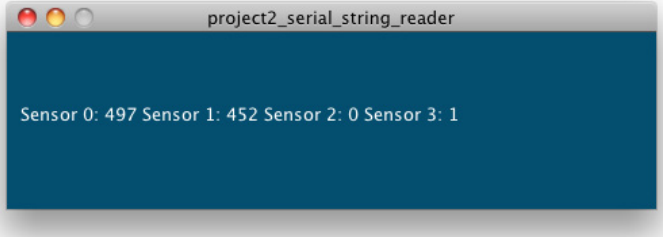
Left-arm sensor (0–1023)	Right-arm sensor (0–1023)	Reset button (0 or 1)	Serve button (0 or 1)	Return character, linefeed character
1–4 bytes	1–4 bytes	1 byte	1 byte	2 bytes

Each section of the sequence is separated by a single byte whose value is ASCII 44 (a comma). You've just made another 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, the 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, where there is no header, 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 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 reassembled.

Let's take a break from writing code and test out the sketch. Make sure you've closed the Serial Monitor or serial port in your serial terminal application so that it releases the serial port. Then run this Processing application. You should see a list of the sensor values in the console and in the applet window, as shown on the right.



» 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).

NOTE: The variables relating to the paddle range in this example are floating-point numbers (floats), because when you divide integers, you get integer results only. For example, 480/400, 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 using scaling functions like map().

```
float leftPaddle, rightPaddle;   // variables for the flex sensor values
int resetButton, serveButton;    // variables for the button values
int leftPaddleX, rightPaddleX;   // horizontal positions of the paddles
int paddleHeight = 50;           // vertical dimension of the paddles
int paddleWidth = 10;            // horizontal dimension of the paddles

float leftMinimum = 120;         // minimum value of the left flex sensor
float rightMinimum = 100;        // minimum value of the right flex sensor
float leftMaximum = 530;         // maximum value of the left flex sensor
float rightMaximum = 500;        // maximum value of the right flex sensor

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

   String portName = Serial.list()[0];
  // open the serial port:
  myPort = new Serial(this, portName, 9600);

  // read bytes into a buffer until you get a linefeed (ASCII 10):
  myPort.bufferUntil('\n');

  // initialize the sensor values:
  leftPaddle = height/2;
  rightPaddle = height/2;
  resetButton = 0;
  serveButton = 0;

  // initialize the paddle horizontal positions:
  leftPaddleX = 50;
  rightPaddleX = width - 50;

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

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 inputString = myPort.readStringUntil('\n');

  // trim the carriage return and linefeed from the input string:
  inputString = trim(inputString);
  // clear the resultString:
  resultString = "";

  // split the input string at the commas
  // and convert the sections into integers:
  int sensors[] = int(split(inputString, ','));
  // if you received all the sensor strings, use them:
  if (sensors.length == 4) {
    // scale the flex sensors' results to the paddles' range:
    leftPaddle = map(sensors[0], leftMinimum, leftMaximum, 0, height);
    rightPaddle = map(sensors[1], rightMinimum, rightMaximum, 0, height);

    // assign the switches' values to the button variables:
    resetButton = sensors[2];
    serveButton = sensors[3];

    // add the values to the result string:
    resultString += "left: "+ leftPaddle + "\tright: " + rightPaddle;
    resultString += "\treset: "+ resetButton + "\tserve: " + serveButton;
  }
}
```

Finally, change the draw() method to draw the paddles (new lines are shown in blue).

```
void draw() {
  // set the background and fill color for the applet window:
  background(#044f6f);
  fill(#ffffff);

  // draw the left paddle:
  rect(leftPaddleX, leftPaddle, paddleWidth, paddleHeight);

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

You may not see the paddles until you flex the sensors. The range of your sensors will be different, depending on how they're physically attached to the monkey, and how far you can flex his arms. The map() function maps the sensors' ranges to the range of the paddle movement, but you need to determine what the sensors' range is. 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've set 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 the maximum and minimum values on each arm. Then, fill them into the four variables, leftMinimum, leftMaximum, rightMinimum, and leftMaximum, in the setup() method. Once you've adjusted these variables, the paddles' movement should cover the screen height when you move the monkey's arms.

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
```

At the end of the setup() method, 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) &&
```

Continued from previous page.

```
        (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);
}

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. Add two more variables 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. This 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. 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();
}
```

To include the scoring display, add a new global variable before the setup() method.

```
int fontSize = 36;      // point size of the scoring font
```

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! Figure 2-14 shows the game in action. For added excitement, get a second pink monkey and put one sensor in each monkey so you can play with a friend.

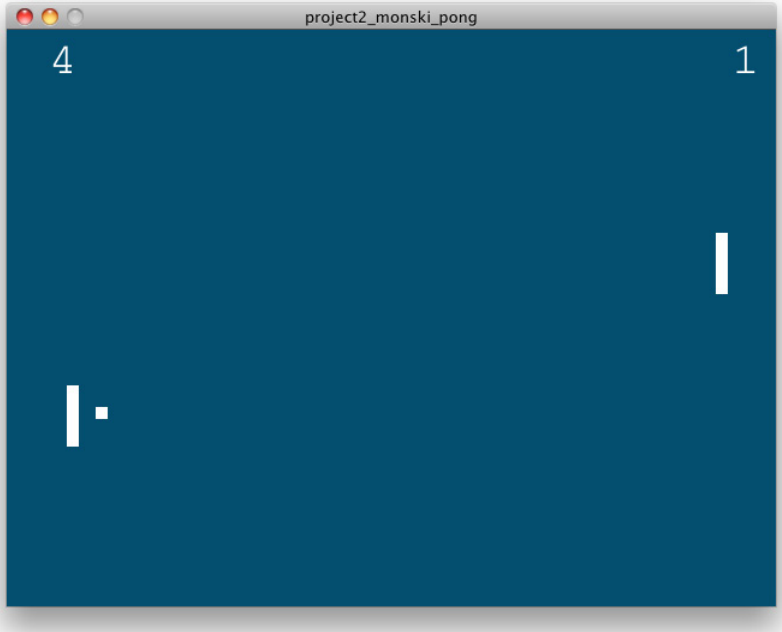


Figure 2-14
The completed Monski Pong Processing sketch.

Flow Control

You may notice that the paddles don't always move as smoothly onscreen as Monski's arms move. Sometimes the paddles don't seem to move 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 the devices agree on the rate at which data is exchanged, it doesn't mean that the receiving computer's program has to use the bits 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 allocate a buffer for each serial port that can hold a couple thousand bytes. The program using the bits (Processing, in the previous example) is juggling 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 add the following lines to the startup() of the Arduino sketch. This makes the Arduino send out a serial message until it gets a response from Processin.:

```
while (Serial.available() <= 0) {
   Serial.println("hello");   // send a starting message
}
```

» Next, 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 (new lines are shown in blue).

In the next step, you'll add some code to the Monski Pong Processing sketch.

```
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
        // ...
    }
}
```

» Add the following lines at the end of serialEvent() in the Processing sketch (new lines are shown in blue).

```
void serialEvent(Serial myPort) {
    // rest of the serialEvent goes here
    myPort.write('\r');      // send a carriage return
}
```

"Now, the paddles should move much more smoothly. Here's what's happening: the microcontroller is programmed to send out a "hello" string until it receives any serial data. When it does, it goes into the main loop. There, it reads the byte just to clear the serial buffer, then sends out its data once, then waits for more data to arrive. Whenever it gets no bytes, it sends no bytes.

Processing, meanwhile, starts its program by waiting for incoming data. When it gets any string ending in a newline, serialEvent() is called, just as before. It reads the string, and if there are commas in it, it splits the string up and extracts the sensor values, like before. If there are no commas in the string (for example, if the string is "hello"), Processing doesn't do anything with it.

The change in the Processing sketch is at the end of the serialEvent(). There, it sends a byte back to the microcontroller, which, seeing a new byte coming in, sends out another packet of data, and the whole cycle repeats itself. This way, the serial buffer on Processing's side never fills up, and it's always got the freshest sensor readings.

The value of the byte that the microcontroller receives is irrelevant. It's used only as a signal from the Processing sketch to let the microcontroller know when it's ready for new data. Likewise, the "hello" that the controller sends is irrelevant—it's only there to trigger Processing to send an intial byte—so Processing discards it. 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 3

Wireless Monski Pong

Monski Pong is fun, but it would 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**
- **1 9V battery and snap connector**
- **Female power plug, 2.1mm ID, 5.5mm OD**
- **1 Bluetooth Mate module**
- **1 project box**

NOTE: If your computer doesn't have built-in Bluetooth, you'll need a Bluetooth adapter. Most computer retailers carry USB-to-Bluetooth adapters.

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 converted data signals to audio to send them across a telephone connection. They modulated the data on the audio connection, and demodulated the audio back into data. Now increasingly rare, their descendants are everywhere, from set-top boxes that modulate and demodulate between a cable TV signal and Internet connection, to the sonar modems that convert data into ultrasonic pings 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 application 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, by 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, letting you 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, which let you focus on sending bytes in the last project.

Add the Bluetooth module to the Monski pong breadboard, as shown in Figure 2-16. Connect the module's ground and VCC to the breadboard ground and +5V, respectively. Connect Arduino's TX to the module's RX, and vice versa, Connect the battery, and the module will start up.

Pairing Your Computer with the Bluetooth Module

To make a wireless connection from your computer to the module, you have to pair them. To do this, open your computer's Bluetooth control panel to browse for new devices.

If you're using Mac OS X, choose the Apple menu→System Preferences, then click Bluetooth. Make sure Bluetooth is turned on and Discoverable, and click "Show Bluetooth status in the menu bar." At the bottom of the list of devices, click the + sign to launch the Bluetooth Setup Assistant. The computer will search for devices and find one called FireFly-XXX (Bluetooth Mate Gold) or RN42-XXX (Silver), where XXX is the Bluetooth module's serial number. If you have no other Bluetooth devices on, it will be the only one. Choose this device, and on the next screen, click Passkey Options. Choose "Use a Specific Passcode," and enter 1234. Click Continue. A connection will be established, as will a serial port. When you look for the serial port in CoolTerm, the Arduino serial port menu, or the Processing serial port list, it will be FireFly-XXX-SPP or RN42-XXX-SPP.

For Windows 7 users, there are different Bluetooth radios in different Windows-based PCs. If your PC doesn't have a built-in Bluetooth radio, any Bluetooth adapter that supports the Windows Bluetooth Stack will do. Most Bluetooth USB dongles on the market support it. Install the drivers according to your radio's instructions, and when it's done, click on the Show Hidden Icons icon in the taskbar (the small triangle in the lower-right-hand corner). When you do, you'll see the Bluetooth Devices icon, as shown in Figure 2-15. Click on this to add a new Bluetooth Device.

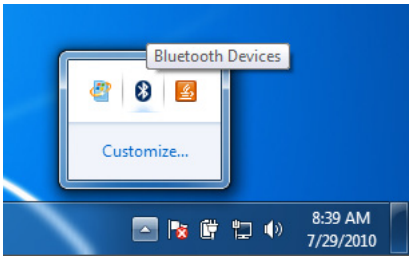


Figure 2-15
Where to find the Windows Bluetooth Devices icon (it's well hidden!). Click Customize... if you want to add it to your taskbar.

The system will search for new devices and present you with a list, which should include one called FireFly-XXX, where XXX is the serial number of your Bluetooth module. If you have no other Bluetooth devices nearby, it will be the only one. When prompted for the device's pairing code, enter 1234. This step will add a new serial port to your list of serial ports. Make note of the port name (mine is COM14) so you can use it later.

Ubuntu Linux's Bluetooth manager for version 1.0 is a bit limited, so it's easier to install BlueMan instead. Go to the Ubuntu Software center, search for BlueMan, and install it. When it's installed, open the System control panel and you'll see Bluetooth Manager, in addition to the default Bluetooth control panel. Open Bluetooth Manager, and it will scan for available devices and show them, including one called FireFly-XXX or RN42-XXX, where XXX is the serial number of your Bluetooth module. When prompted for the device's pairing code, enter 1234. Once it's added, click Setup, and you'll get a dialog asking you if you want to connect to a serial port. Click Forward, and it will tell you the name of the serial port for the Bluetooth module, **/dev/rfcomm0**.

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 these lines in the setup() method:

```
String portName = Serial.list()[0];
// open the serial port:
myPort = new Serial(this, portName, 9600);
```

For example, if the Bluetooth port is the ninth port in your list, change the first line to open Serial.list[8]. Then change the data rate in the second line as follows:

```
myPort = new Serial(this, portName, 115200);
```

> **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.**

Likewise, you'll need to change your Arduino sketch so that the Serial.begin() line reads as follows:

```
Serial.begin(115200);
```

Disconnect the Bluetooth module from the Arduino before you upload the modified sketch, as it will interfere with the upload. You can re-connect it once you've uploaded the new code. 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 on.

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 through the radio. If so, make the changes from that section first. Once you do, Monski Pong should operate as before, only now it's wireless.
X

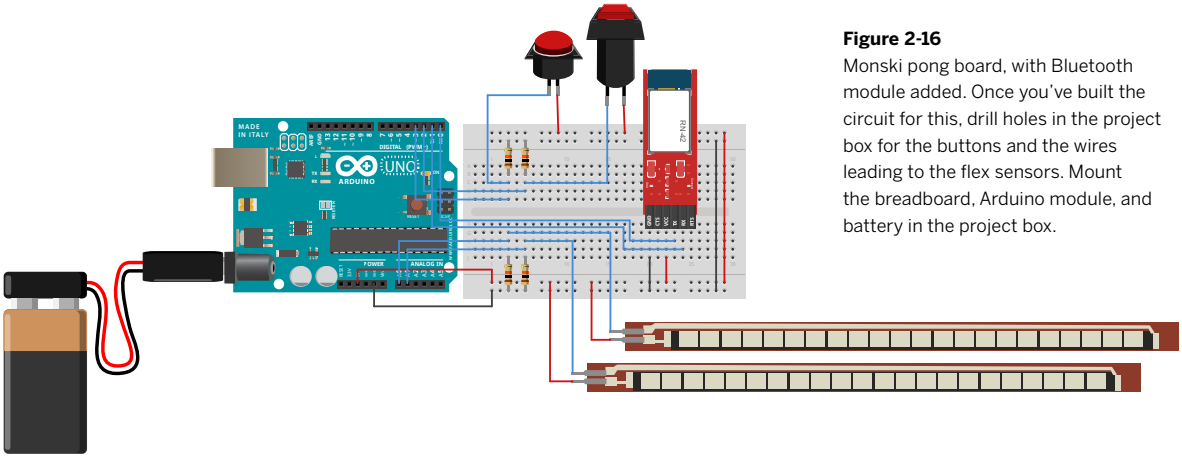


Figure 2-16
Monski pong board, with Bluetooth module added. Once you've built the circuit for this, drill holes in the project box for the buttons and the wires leading to the flex sensors. Mount the breadboard, Arduino module, and battery in the project box.

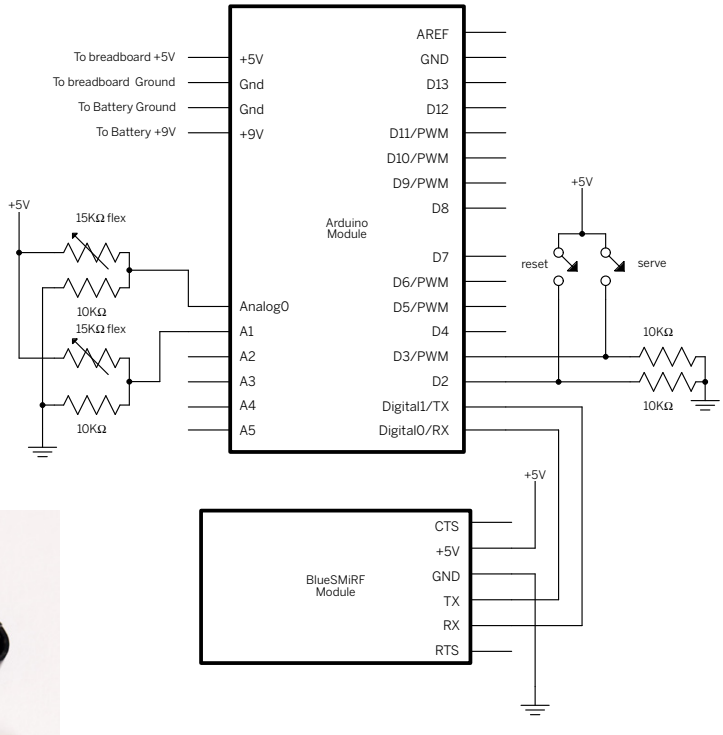
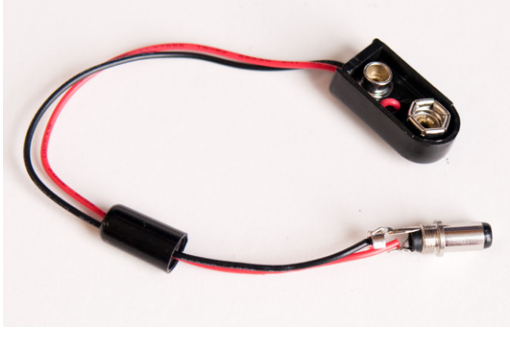


Figure 2-17
If you didn't buy a battery snap with DC power adapter like the Spark Fun one, you'll need to solder the connector onto your battery snap, as shown here.



Finishing Touches: Tidy It Up, Box It Up

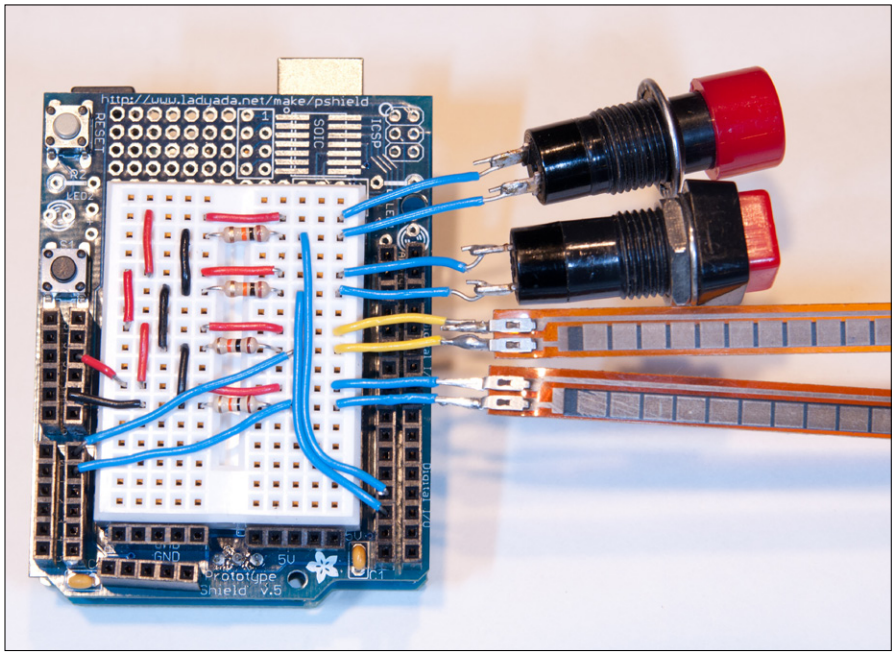


Figure 2-18.
You might want to shrink Monski Pong so it's more compact. This figure shows the Monski Pong circuit on a breadboard shield. This is the same circuit as the one in Figure 2-16, it's just on a different breadboard so it can fit in a project box.



Figure 2-19.
Kitchen storage containers make excellent project boxes. Here's the Monski Pong controller with Monski attached.

Project 4

Negotiating in Bluetooth

The steps you went through to pair your Bluetooth module with your computer negotiated a series of exchanges 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 would 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 to handle that negotiation.

MATERIALS

- **1 Bluetooth Mate module**
- **1 FTDI USB-to-Serial converter**

The Bluetooth module is essentially a modem in that it converts from one communications medium (TTL serial carried over wires) to another (Bluetooth serial carried over radio). Modems are designed to open a connection to another modem, negotiate the terms of data exchange, carry on an exchange, and then disconnect. To do this, they must 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 (and many other communications devices) use a set of commands based on the those designed originally 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. Devices

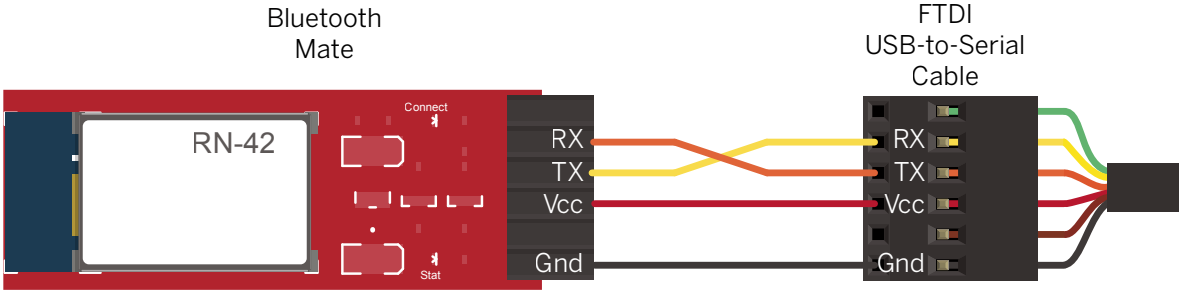


Figure 2-20
Bluetooth Mate module connected to an FT232RL USB-to-Serial converter. The Mate is designed to be a drop-in replacement for the FTDI cable, so it has the same pin configuration. As a result, you have to cross the transmit and receive connections to make the connection work.

using this protocol all have a command mode and a data mode. To switch from data mode to command mode in the Hayes protocol, send the string +++. There's a common structure to all the 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. The modem then responds to the command with the message OK, followed by any information it's expected to return.

The Bluetooth Mate doesn't use AT commands, but its protocol is similar. The commands are all ASCII-based.

There is a command mode and a data mode. In data mode, you send the string `$$$` to switch to command mode.

The commands are all short strings, and the modem responds with the response AOK. To exit command mode, send the string `---\r` (the \r is a carriage return, ASCII 13), and the Bluetooth modem switches back to data mode. In data mode, any bytes you send the modem are sent out over the radio, and any bytes received over the radio get sent out the serial connection.

Controlling the Bluetooth Module

Wire the Bluetooth module to the USB-to-Serial converter, as shown in Figure 2-20. Since the converter and the cable have identical pin configurations, you'll need to cross the TX and RX lines to make them connect. Connect the converter to a USB port on your computer.

For this project, you'll need a serial terminal program that can open two serial ports at the same time, so the Arduino Serial Monitor won't do. One serial port will be the wired connection to the Bluetooth module through the USB-to-serial adapter. The other will be the wireless connection via Bluetooth.

For Mac OS X and Windows users, CoolTerm will work well. For Ubuntu Linux users, GNU screen or PuTTY will do. Open a connection to the USB-to-Serial adapter at 115200 bits per second.

When you first power the Bluetooth Mate, it will be in data mode. To switch to command mode, send the string:

```
$$$
```

The module will respond like so:

```
CMD
```

You're now in command mode. Any time you want to check that the module is working and in command mode, type `enter` or `return`, and send it. It will respond with `?`. To see a list of all possible commands, type `H` and then enter. There's a list of all the commands available for this module at www.SparkFun.com or www.rovingnetworks.com (the Bluetooth Mate Gold module uses the Roving Networks RN-41 radio, if you're searching their site, and the Bluetooth Mate Silver uses the RN-42 radio). A few of the commands are covered here. Each Bluetooth modem manufacturer has its own set of commands; unfortunately, they're all different. But they all have the same basic structure as the one you see here.

Currently, the module is in command mode. One of the first things you'd like is to see its settings. Type `D` and hit return or enter. You'll get a list of the radio's settings, which looks like this:

```
***Settings***

BTA=000666112233
BTName=FireFly-7256
Baudrt(SW4)=115K
Parity=None
Mode=Slav
Authen=0
Encryp=0
PinCod=1234
Bonded=0
Rem=NONE SET
```

The first setting is the Bluetooth 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 settings that follow give you information about the radio's configuration, such as the serial data rate (or baudrate), whether authentication's turned on, and what the passcode or PIN code is.

You may have noticed that you can't see what you type. That's because the keystrokes in a serial terminal program aren't echoed back to the screen—they're sent straight to the serial port. However, the Bluetooth Mate will echo your characters back to you if you type `+\r`. This turns Echo Mode on or off. It's useful to have on while issuing commands.

Now that you know something about your own module, you want it to give you a list of other Bluetooth-enabled devices in the area. Type `I` for inquiry. After several seconds, it will come back with a list like this:

```
?
Inquiry, COD=0
Found 9
0010C694AFBD,,1C010C
0023125C2DBE,tigoebook,3A010C
0017F29F7A67,screen1,102104
002241CE2E79,residents,380104
002241D70127,admints Mac mini,380104
0014519266B8,ERMac,102104
002241CE7839,VideoMac05,380104
E806889B12DD,,A041C
00236CBAC2F0,Fred Mac mini,302104
Inquiry Done
```



You can't initiate a connection from the Bluetooth Mate to the computer unless you've previously paired with the Mate from the computer. This is because Bluetooth 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.

This is a list of all the other Bluetooth devices it found. The first part of every string is the device's unique address. The second, when there is one, is the device's name; the third is the device class, or what type of device it is (you can probably pick out the device class number for Mac minis from the list above). Names don't have to be unique, but addresses do, which is why you always use the address to connect.

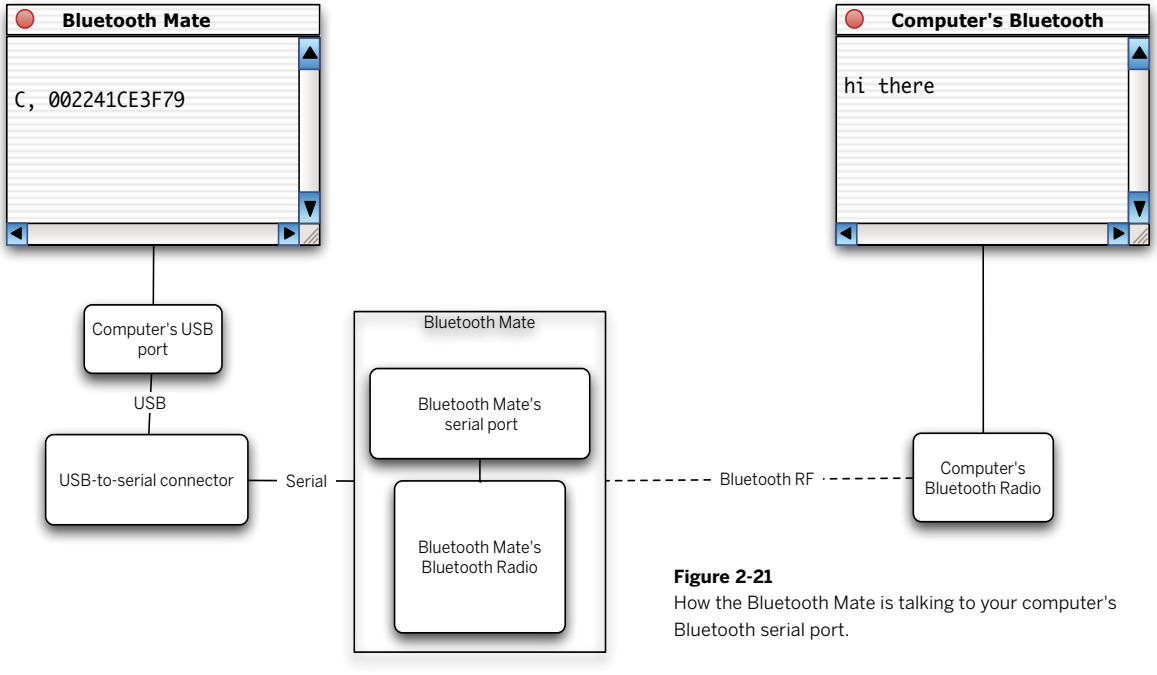


Figure 2-21
How the Bluetooth Mate is talking to your computer's Bluetooth serial port.

Now that you've got a list of connections, you're going to try to connect to the one that represents your computer. You must already be paired with the Bluetooth module for this to work, so if you aren't, go back to the previous project and do that now. Then, 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 at 115200 bps.

For Windows 7 users, click on the Show Hidden Icons icon in the taskbar to get to the Bluetooth Devices. Click the Bluetooth Devices Icon and, from the menu, choose Open Settings. In the Settings window, choose the Options pane—and make sure that you've allowed other devices to find this computer, and that you've allowed them to connect as well. Then click the COM ports pane. If you've paired with your Bluetooth module before, there will be two ports indicated for it, one outgoing and one incoming. You're not using those, though, because Windows tries to initiate contact when you do. You want the Bluetooth modem to initiate contact. Add a new incoming port. Note the port number, then open that port in your serial terminal program at 115200 bps.

For Ubuntu Linux users, the Bluetooth Manager doesn't support binding a serial port to a discoverable Bluetooth serial port protocol (SPP) connection, so you'll need to make the connection outbound from Ubuntu, as described in the earlier section, "Pairing Your Computer with the Bluetooth Module." Once you're paired, however, the procedure to send and receive data, or to switch from command mode to data mode on the Bluetooth Mate, will be the same as for other platforms.

Once you've opened the serial port on your computer, go back to the window with the serial connection to the Bluetooth module, and send the following command:

```
C, address\r
```

where address is the Bluetooth address of your computer that you discovered earlier. When you get a good connection, the LED on the Mate will turn on, and you can type back and forth between the windows. This is more exciting if you have two computers and connect them via Bluetooth, but it works on one computer nonetheless.

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

To get out of data mode (to check the modem's status, for example), type (\r indicates that you should hit Enter or Return; don't type the \ or the r):

```
$$$\r
```

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

```
---\r
```

Finally, when you're in command mode, you can type K,\r to disconnect. If you want to connect to another device, go into command mode and start over again.

Because these 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 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, remember that data communication is based on a layered series of agreements, starting with the physical layer; then the electrical, the logical, the data layers; and finally, the application layer. Keep these layers in mind as you design and troubleshoot your projects, and you'll find it's 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 all the connected devices. 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, consider the flow of data, and look for ways to 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. 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 1940s radio restored to working condition. It features a tiny video-projected dancer who shakes and shimmies to 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 1940s 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 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.

Supplies for Chapter 3

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **RS** RS (www.rs-online.com)
- **RSH** RadioShack (www.radioshack.com)
- **SF** SparkFun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 5: Networked Cat

» **Between 2 and 4 force sensing resistors, Interlink 400 series** (www.interlinkelec.com). The Interlink model 402 is shown in this project, but any of the 400 series will work well. **D** 1027-1000-ND, **J** 2128260, **SF** SEN-09673

» **One 1-Kilohm resistor** Any model will do. **D** 1.0KQBK-ND, **J** 29663, **F** 1735061, **RS** 707-8669

» **1 Arduino module** An Arduino Uno or something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards. **D** 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

» **1 personal computer**

» **1 web camera**

» **1 cat** A dog will do if you have no cat.

» **1 cat mat**

» **2 thick pieces of wood or thick cardboard, about the size of the cat mat**

» **Wire-wrapping wire** **D** K445-ND, **J** 22577, **S** PRT-08031, **F** 150080

» **Wire-wrapping tool** **J** 242801, **F** 441089, **RSH** 276-1570, **S** TOL-00068

» **Male header pins** **D** A26509-20-ND, **J** 103377, **S** PRT-0011, **F** 1593411



Figure 3-1. New parts for this chapter: **1.** Interlink Series 402 force-sensing resistors (FSRs) **2.** 30AWG wire-wrapping wire **3.** Wire-wrapping pins (or long female headers) **4.** New tool: wire-wrapping tool. The wire stripper to the left of the tool lives inside the handle. Don't forget plenty of male header pins for the breakout boards.

Network Maps and Addresses

In the previous 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. You also need an addressing scheme to know how a message gets to its destination.

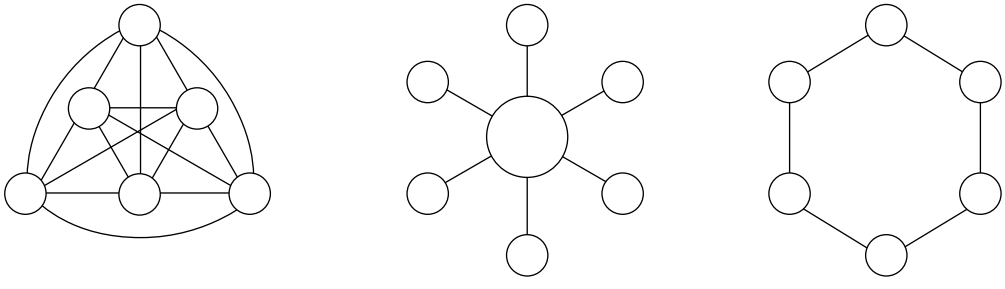
Network Maps: How Things Are Connected

The arrangement of a network's physical connections depends on how you want to route its messages. 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 3-2, 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 shown in Figure 3-2. This way works great as long as the hub continues to function, but the more objects you add, the faster the hub must 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 3-3, 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 3-3), each connector may be linked to more than one other connector, so that enabling messages to 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. As

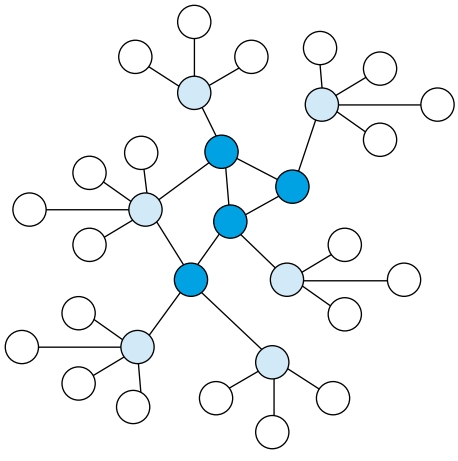
Figure 3-2
Three types of network: direct connections between all elements, a star network, and a ring network.



Directly connected network | Star network | Ring network

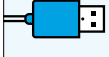
long as the number of objects connected to each of these is small, the effect on the whole network is minimal. It may not seem minimal when you're using the object whose connector fails, but the rest of the network remains stable, so it's easy to reconnect when your connector is working again.

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 these models to see which is best for you. In simpler systems, one of the three networks shown in Figure 3-2 might do the job just fine, saving you some complications. As you get further into the book, you'll see some examples of these; for the rest of this chapter, you'll work with the multitiered model by relying on the Internet as your infrastructure.
X



Multitiered network

Figure 3-3
A complex, multitiered network.



Modems, Hubs, Switches, and Routers

The connectors in Figure 3-3 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.

A switch is like a hub, but it's more sophisticated. It keeps track of the addresses of the objects attached to it, and it passes along messages addressed to those objects only. Objects attached to a switch 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 most 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 it 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.

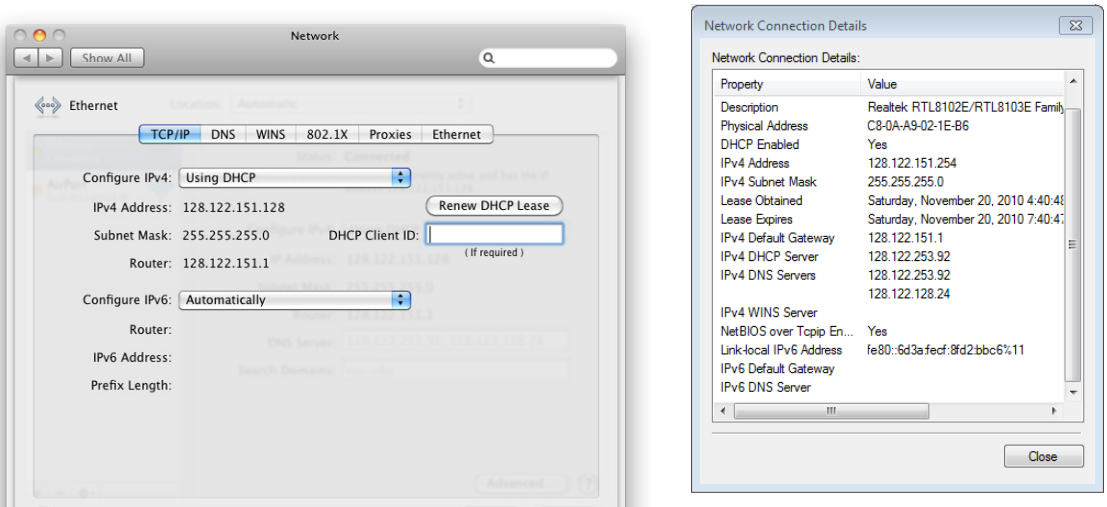


Figure 3-4
Network settings panels for Mac OS X and Windows.

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 create 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. WiFi 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. Here you'll get a list of the possible network interfaces through which your computer can connect to the Internet. 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 select either, you can find out that interface's hardware address. In either interface, click on the Advanced button to get to both the Ethernet tab (where you can see the hardware address), and the TCP/IP tab (where you can see the machine's IP address if you're connected to a network).

In Windows 7, click the Start Menu→Control Panel, then double-click "Network and Internet". 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 your WiFi connection. Under the Support tab, click Details to see the IP settings and hardware address.

For Ubuntu Linux, click the System menu, then Preferences→Network Connections. You'll see a list of network interfaces. Click Edit to see their details.

Figure 3-4 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, whenever you begin working on a new project, note both addresses for every device you're using.

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 Protocol (IP) 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: 217.123.152.20. The router that this machine is attached to most likely has this address: 217.123.152.1.

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, for example, 217.123.152.xxx. Sometimes a router manages a larger subnet or even a group of subnets, each with its own local router. The router that this router is connected to might have the address 217.123.1.1.

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 may have encountered a subnet mask when 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, there could be only one machine in the subnet: the router itself. If the last octet is 0, as it is above, 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. There are a few other reserved addresses, so the real numbers are a bit lower. Table 3-1 shows a few other representative values to give you an idea.

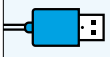
Table 3-1. The relationship between subnet mask and maximum number of machines on a network.

Subnet mask	Maximum number of machines on the subnet, including the router (accounting for reserved addresses)
255.255.255.255	1 (just the router)
255.255.255.192	62
255.255.255.0	254
255.255.252.0	1022
255.255.0.0	65,534

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

You're probably thinking 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



Private and Public IP Addresses

Not every object on the Internet can be addressed by every other object. Sometimes, in order to support more objects, a router hides the addresses of the objects attached to it, sending 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 (as well as 10.xxx.xxx.xxx, and 172.16.xxx.xxx–172.31.xxx.xxx) are to be used for private addressing only. This address range is used commonly in home routers, so 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 on my message, 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, which expands the total number of things that can be attached to the Internet.

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 Dynamic Host Control Protocol (DHCP) , which also provides their IP address, so you don't have to worry about configuring DNS. In this chapter's project, 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.

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, and 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 and receiving addresses are attached to the message in another header. Finally, you send it. Your courier might want to break up the shipment 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 main protocols used to handle transport of packets on the Internet: Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). You'll learn more about these later. The main difference is that TCP provides more error-checking from origin to destination, but is slower than UDP. On the other hand, UDP trades off error-checking in favor of speed.

Each router sends off the packets one at a time to the routers to which it's connected. 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 the packets 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. On the whole, though, the network is reliable enough that you can forget about dropped packets.

There's a command-line tool, ping, that can be useful in determining whether your messages are getting through.It sends a message to another object on the Net to say "Are you there?", and then 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 10 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 of not only how many packets got through, but also how long they took. It's a useful way to learn quickly whether a given device on the Internet is reachable or not, as well as how reliable the network is between you and that device. Later on, you'll be using devices that have no physical interface on which you can see activity, so *ping* is a handy way to check whether they're working.

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.

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. Once you know how that works, you can take it for granted and concentrate on the messages.

How Web Browsing Works

Figure 3-5 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. The browser makes a connection to the server computer, the server program accepts the connection and delivers the files representing the page, and the exchange is made.

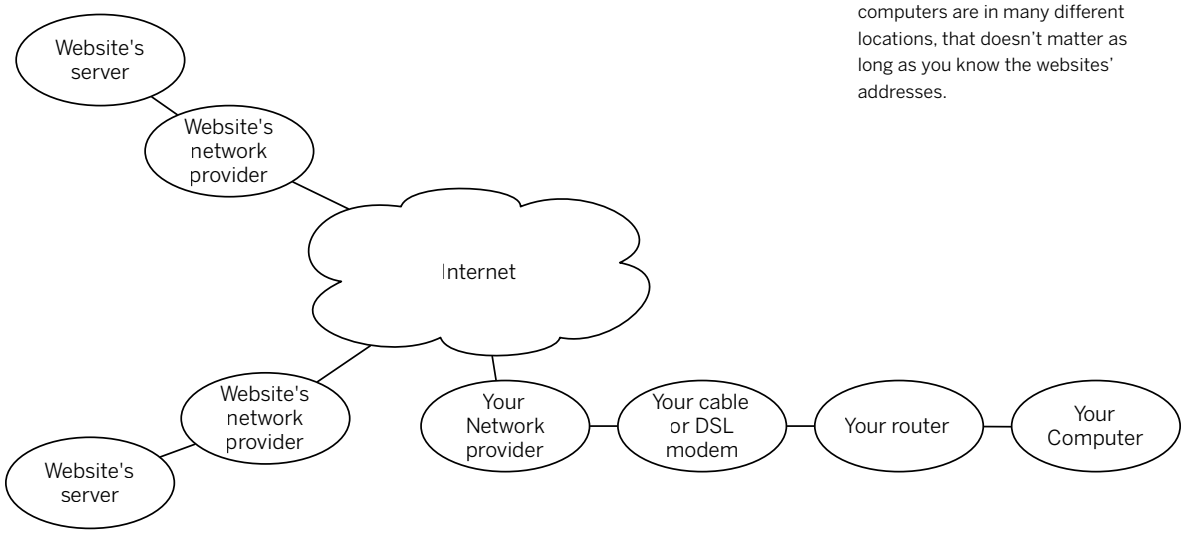


Figure 3-5
The path from a website to your browser. Although the physical computers are in many different locations, that doesn't matter as long as you know the websites' addresses.

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 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, such as 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. 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 the requests from browser to server, and all 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 terminal program again, just as you did for the *ping* example shown earlier (on Windows 7, use PuTTY).

Try It Type the code at right.

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

The server will respond as follows (on Windows,use PuTTY and open a connection to www.google.com, port 80, Connection Type: Raw):

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



The built-in Windows version of telnet is not very good. For example, you won't be able to see what you type without setting the localecho **option, and the informative "Trying . . . Connected" prompts do not appear. Use PuTTY instead (http://www.chiark.greenend.org.uk/~sgtatham/putty).**

Type the code to the right.

```
GET /index.html HTTP/1.1
Host: www.google.com
Connection: Close
```

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

```
HTTP/1.1 200 OK
Date: Mon, 18 Jul 2011 00:04:06 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws
Connection: Close
```

NOTE: If telnet doesn't close on its own, you may need to press Ctrl-] 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). Try getting this file from the command line.

Try It Modify the PHP program slightly, removing all the lines that print any HTML, like the code on the right.

```
<?php
/* Date page
   Context: 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";

?>
```

Now, telnet into your 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.

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 to the > character, and you've got it.

HTTP requests don't just request files. You can add parameters to your request. If the URL you're requesting is actually a program (like a PHP 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 (&).

There are predefined variables in PHP that give you access to these parameters and other aspects of the exchange between client and server. The variable $_REQUEST, used below, is an example that returns the parameters after the question mark in an HTTP request. Other predefined variables give you information about the client's browser and operating system, the server's operating system, any files the client's trying to upload, and much more.

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
   Context: PHP

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

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

You could also request it from telnet or PuTTY 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 to the code at right, shown here 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


name: tom<br>
age: 14<br>
```

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

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 a number greater than 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
   Context: 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.
*/

// 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, ";
    echo "but do so responsibly.</p>\n";
}
?>
```

HTTP GET and POST

The ability to respond to parameters sent in an HTTP request opens all kinds of possibilities. For example, you can write a script that lets you choose the address to which you send an email message, or what message to send. You'd just add parameters to the URL after the question mark, read them in PHP, and use them to set the various mail variables.

The method for sending variables in the URL after a question mark is called GET, and it's one of HTTP's four commands. Besides GET, you can also POST, PUT, and DELETE. Many browsers don't support PUT or DELETE, however, so stick to GET and POST for now.

POST is the method usually used to post data from a web form. Instead of adding the parameters on the end of the URL path as GET does, POST adds the parameters to the end of the whole HTTP request. There are a couple other parameters you need to add to a POST request too, like the content type and the content length. POST is a little more work to set up, but it's really useful for hiding the messy business of passing parameters, keeping your URLs tidy and easy to remember. Instead of the previous URL, all the user has to see is:

```
http://www.example.com/get-parameters.php
```

The rest can get delivered via POST.

Try It Telnet into your web server on port 80 and request the file again from the command line. This time, type the POST request as shown here.

You'll get the same response as you did using GET, but now you can put all the parameters in one place, at the end.

```
POST /age_checker.php HTTP/1.0
Host: example.com
Connection: Close
Content-Type: application/x-www-form-urlencoded
Content-length: 16

name=tom&age=14
```

» The content length is the length of this string, plus a linefeed. If your name is longer than three letters, or your age is greater or fewer than two digits, change the content length to match.

» Now make a couple changes to the script. First, wrap the section that checks name and age in another if statement, like so (new lines are shown in blue).

```
if (isset($name) && isset($age) ) {
    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";
    }
}
?>

<html>
<body>

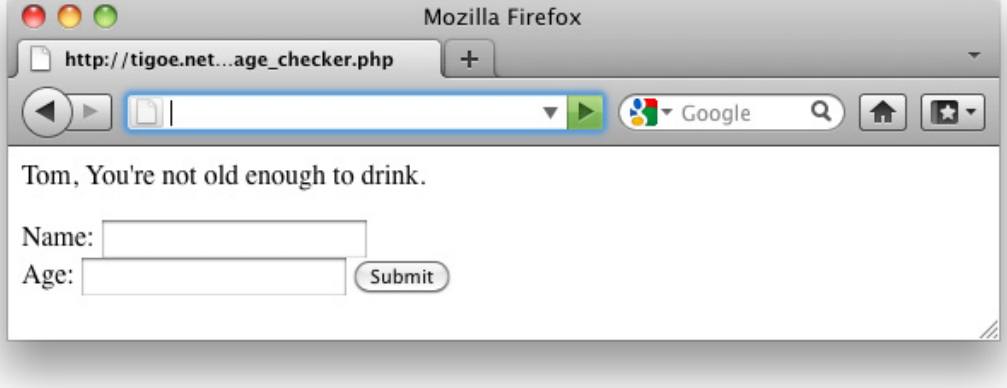
<form action="age_checker.php" method="post"
enctype="application/x-www-form-urlencoded">
Name: <input type="text" name="name" /><br>
Age: <input type="age" name="age" />
<input type="submit" value="Submit" />
</form>

</body>
</html>
```

Add the following HTML to the **age_checker.php** file after the closing PHP tag.

When you reload this script in the browser, you will see a form as shown in Figure 3-6. The if statement you added makes sure the message about age doesn't show up unless you've entered values for name and age. And the HTML form calls the same script again when you submit your values, using an HTTP POST request.

Figure 3-6
The PHP age-checker form.



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 email server. 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 servers are online all the time, waiting for new messages for all of their users.

The transport protocol for sending mail is called SMTP, the Simple Mail Transport Protocol. It's paired with two retrieval protocols: POP (Post Office Protocol) and IMAP (Internet Message Access Protocol). Just like HTTP, it's text-based. PHP has excellent functionality for sending and retrieving mail, and you can use it as an intermediary between any local application, like the Processing sketch that will follow, or some microcontroller applications you'll see later on.

Try It

Here's a PHP script that sends a mail to you. Save this script to your server as **mailer.php,** and view it in a browser as you did with the last script. You should get two results. In the browser, you'll get a page that says:

```
I mailed you@example.com
From: you@example.com
Hello world!

Hi there, how are you?
```

In your mail client, you'll get a message like this:

```
From:      You <you@example.com>
Subject:   Hello world!
Date:      May 8, 2013 2:57:42 PM EDT
To:        you <you@example.com>

Hi there, how are you?
```

```
<?php
/*
   mailer
   Context: PHP

   sends an email.
*/

// set up your variables for mailing. Change to and from
// to your mail address:
$to = "you@example.com";
$subject = "Hello world!";
$from = "From: you@example.com";
$message = "Hi there, how are you?";

// send the mail:
mail($to, $subject, $message, $from);

// give notification in the browser:
echo "I mailed " . $to . "<br>";
echo $from . "<br>";
echo $subject. "<br><br>";
echo $message;
?>
```

Make sure your server is properly configured to send mail here. If you're using a web hosting service, the settings for mail will be part of your account settings. Also, make sure your from: address is one for which the mail server will relay mail. That typically means you can only send mail from your own domain. For example, if your domain name is example.com, then you can't send mail from cat@ohaikitteh.com.

As you can see, sending mail from PHP is very simple. So in addition to using it to serve web pages, you can use it to send messages via mail. As long as you have a device or program that can make a GET or POST, you can use PHP or other server-side programming languages to start a sequence of messages in many different applications across the Net. Now that you've got the basics of HTTP requests and mail sending, it's time to put them into action in a project.

Project 5

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 while showing you how to build your first physical interface for the Internet.

MATERIALS

- **Between 2 and 4 force-sensing resistors, Interlink 400 series**
- **1 1-kilohm resistor**
- **1 solderless breadboard**
- **1 Arduino microcontroller module**
- **1 personal computer**
- **1 web camera**
- **1 cat mat**
- **1 cat**
- **2 thin pieces of wood or thick cardboard, about the size of the cat mat**
- **Wire-wrapping wire**
- **Male header pins**

If you're a cat lover, you know how cute they can be when they curl up in their favorite spot for a nap. You might find it useful during 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 would be even better if you could then check in on the cat's website to see him at his cutest. This project makes that possible.

The system works like this: force-sensing resistors are mounted under the cat mat and attached to a microcontroller. The microcontroller is attached to a personal computer, as is a camera. When the cat lies down on the mat, his weight will cause a change in the sensor readings. The microcontroller then sends a signal to a program on the personal computer, which takes a picture with the camera and uploads it to a web server via a PHP script. Then the program calls another PHP script that sends you an email, letting you know that your cat is being particularly cute. Figure 3-7 shows the whole system.

You'll do this project in several parts:

1. Write an Arduino sketch to read sensors in the cat's mat and send the results serially to Processing.
2. Write a Processing sketch to read the serial data and when appropriate, call a PHP script that sends mail.
3. Write a PHP script to send mail.
4. Make a web page for the cat cam.
5. Write a second PHP script to accept new image uploads for the web page.
6. Modify the Processing sketch to take new images and upload them via the second PHP script.

Putting Sensors in the Cat Mat

First, you need a way to sense when the cat is on the mat. The simplest way to do this is to put force sensors under the mat and sense the difference in weight when he sits on it. How you do this depends on what kinds of force-sensing resistors you use. Interlink's 400 series FSRs work well for this project. Mount the sensors on something with a firm backing, like masonite, or another type of wood or firm cardboard.

There are some FSRs that are long and thin, like Interlink's 408 series. Long sensors aren't all that common, though, so you're more likely to have small round sensors like the 400 or 402 series. If you're using those or other 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 or hardware store—and position them on the

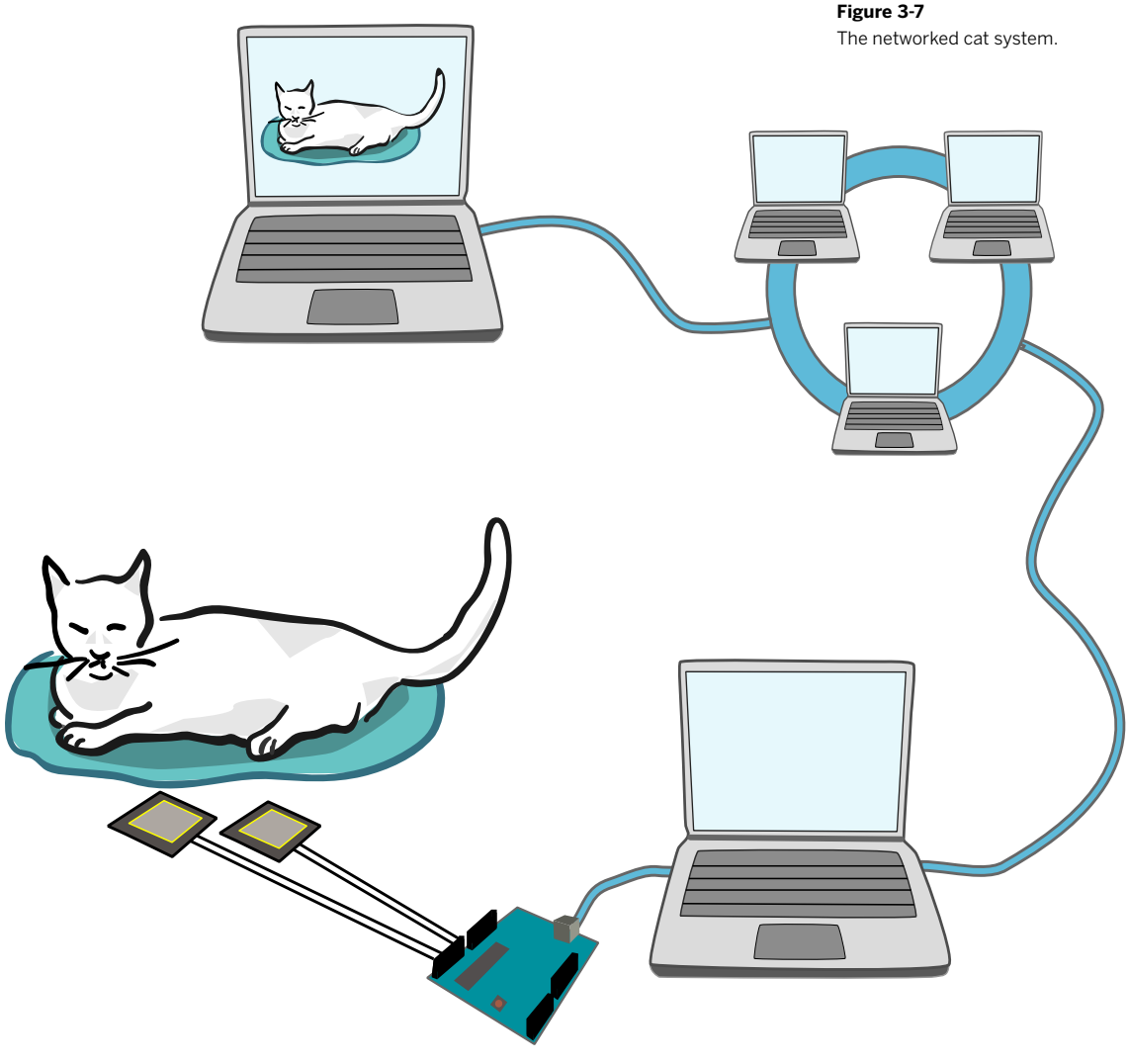


Figure 3-7
The networked cat system.

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 3-8, 3-9, and 3-10 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.

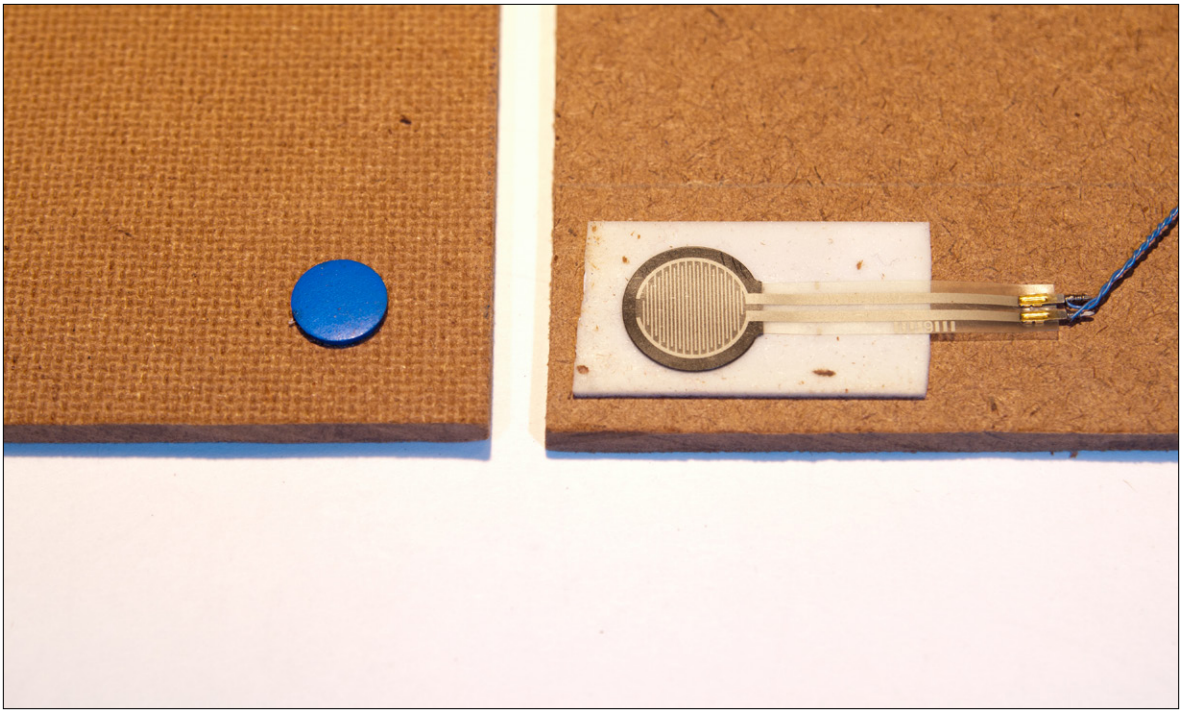
You're using multiple sensors so you can sense a large area under the mat, but it doesn't matter much which one gets triggered. Connect the sensors to an analog input of the microcontroller in parallel with each other, using the voltage divider circuit shown in Figure 3-10. This circuit combines the input from all four into one input.

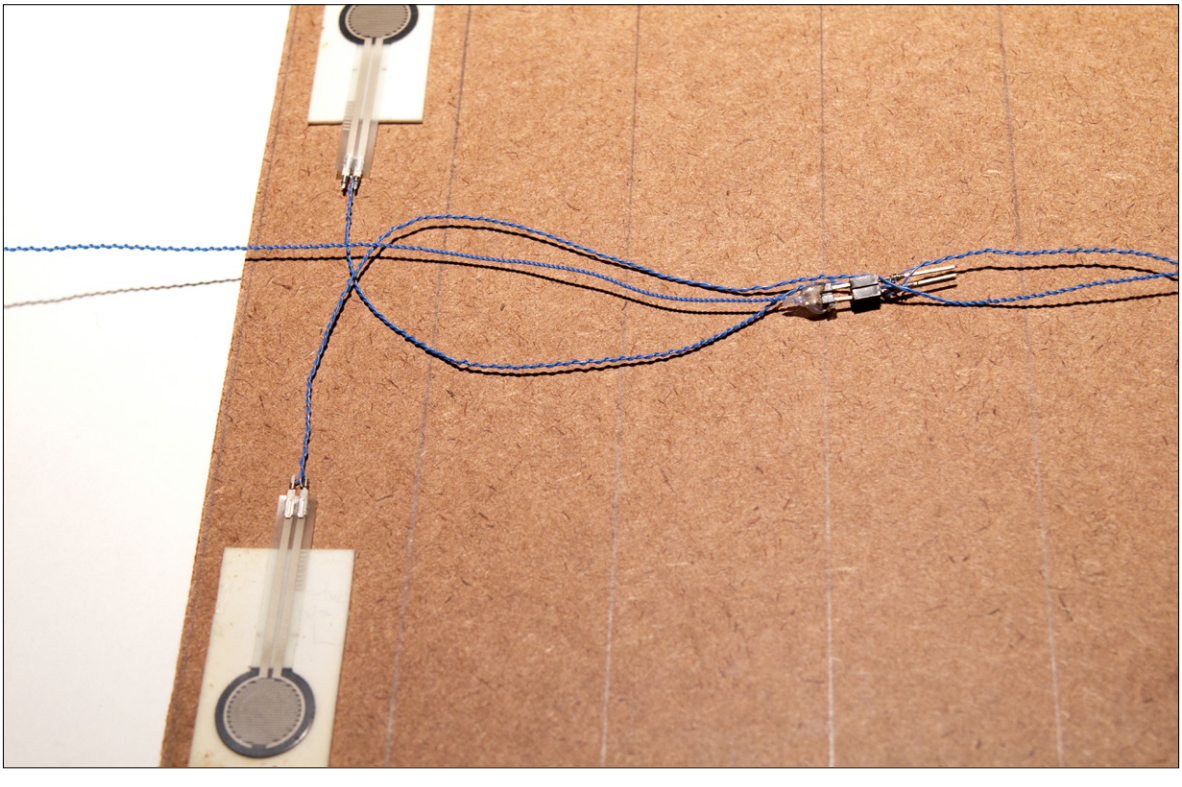
X

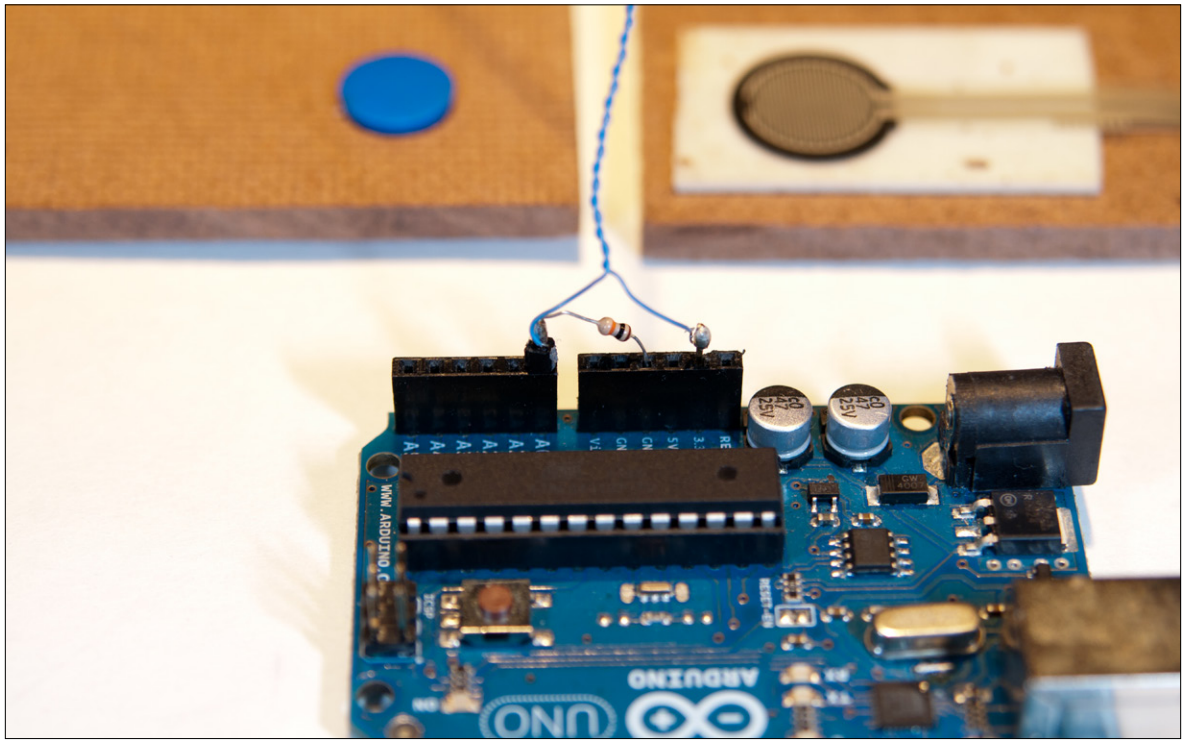


Figure 3-8
Because the force-sensing resistors melt easily, I used 30AWG wire wrap instead of solder. Wire-wrapping tools are inexpensive and easy to use, but make a secure connection. After wire wrapping, I insulated the connections with heat shrink. (Heat shrink not shown.)

Figure 3-9
The 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.







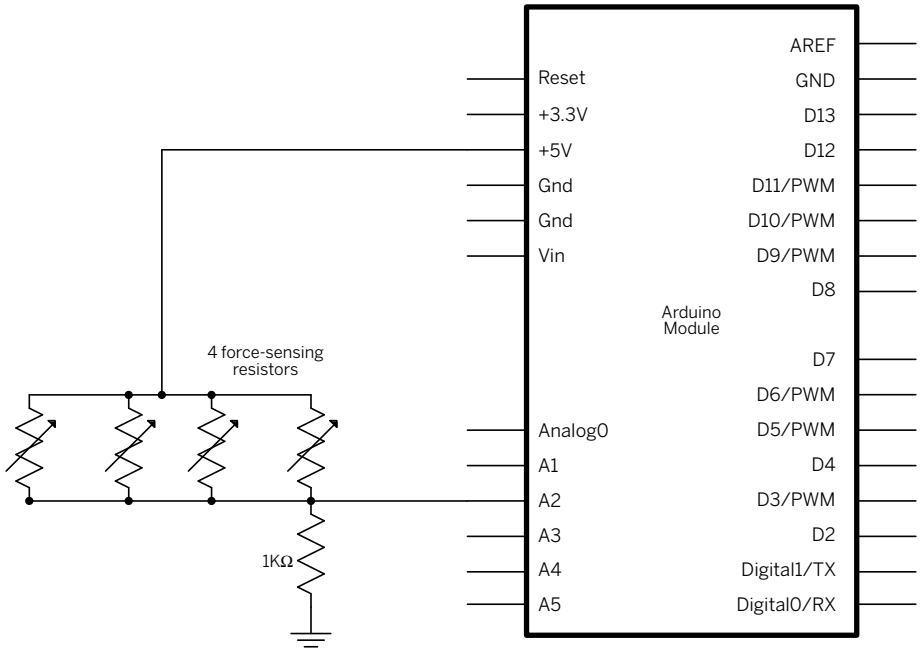


Figure 3-10
The cat-sensing circuit. Because all of the force-sensing resistors are wired in parallel, there are only two connections for all of them. This circuit is simple enough that you can just solder a resistor to a header pin to make the connection to ground.

You may need a higher value depending on the resistance range of your force-sensing resistors and the weight of your cat. If a 1K resistor doesn't give you good values, try a 4.7K or a 10K resistor. The photo here shows a 10K resistor.

Test It

Once you've got the sensor panel together and connected to the microcontroller, run this code on the Arduino 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.

```
/*
   Analog sensor reader
   Context: Arduino

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

void loop()
{
  // read analog input:
  int sensorValue = analogRead(A0);

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

Connect It

Next, send the sensor readings serially to Processing, which will send an email and trigger the camera to take a picture when the cat is on the mat. This sketch will look familiar to you because it's similar to the one you used to read the sensor values for Monski Pong in Chapter 2.

```
/*
 Serial String Reader
 Context: Processing

 Reads in a string of characters until it gets a linefeed (ASCII 10).
 Then converts the string into a number.
 */
import processing.serial.*;

Serial myPort;              // the serial port
float sensorValue = 0;      // the value from the sensor
float xPos = 0;             // horizontal position of the graph

void setup() {
  size(400,300);
  // list all the available serial ports
  println(Serial.list());

  // I know that the first port in the serial list on my Mac is always my
  // Arduino, so I open Serial.list()[0]. Open whatever port you're using
  // (the output of Serial.list() can help; they 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 newline (ASCII 10):
```

»

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 to the mat. If he jumps on and off and on again in a minute or less, you don't want another email.

What does that look like in sensor terms? To find out, you need to do one of two things: 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.

```
  myPort.bufferUntil('\n');

  // set initial background and smooth drawing:
  background(#543174);
  smooth();
}

void draw () {
  // nothing happens here
}

void serialEvent (Serial myPort) {
  // get the ASCII string:
  String inString = myPort.readStringUntil('\n');

  if (inString != null) {
    // trim off any whitespace:
    inString = trim(inString);
    // convert to an int and map to the screen height:
    sensorValue = float(inString);
    sensorValue = map(sensorValue, 0, 1023, 0, height);
    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 a few extra variables to the variable list at the beginning of your Processing program.

```
float prevSensorValue = 0;  // previous value from the sensor
float lastXPos = 0;         // previous horizontal position
```

» Next, add a new method called drawGraph().

```
void drawGraph(float prevValue, float currentValue) {
  // subtract the values from the window height
  // so that higher numbers get drawn higher
  // on the screen:
  float yPos = height - currentValue;
  float lastYPos = height - prevValue;

  // draw the line in a pretty color:
  stroke(#C7AFDE);
  line(lastXPos, lastYPos, xPos, yPos);

  // at the edge of the screen, go back to the beginning:
  if (xPos >= width) {
```



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 easier if your computer doesn't have to be tethered to the cat mat in order to program.

```
    xPos = 0;
    lastXPos = 0;
    background(#543174);
  }
  else {
    // increment the horizontal position:
    xPos++;
    // save the current graph position
    // for next time:
    lastXPos = xPos;
  }
}
```

Finally, add the following to the serialEvent() method, right after you print the sensor value (new lines are shown in blue).

```
      println(sensorValue);
      sensorValue = map(sensorValue, 0, 1023, 0, height);
      drawGraph(prevSensorValue, sensorValue);
      // save the current value for the next time:
      prevSensorValue = sensorValue;
    }
  }
```

When you run the program, you'll see a graph of the sensor values, as shown in Figure 3-11. 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. Using the sensor values, you have enough knowledge to start defining the cat's presence on the mat as an event. The event you care about is when the sensor reading increases significantly, because that's when the cat sat on the mat. To do this, pick a threshold number in between the two states. When the sensor reading changes from being less than threshold to greater than or equal to it, that's your first event. When the sensor value goes below the threshold, the cat has left the mat. That's your second event.

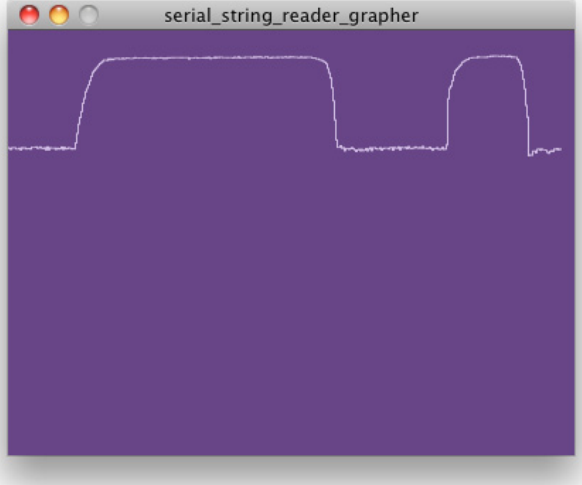


Figure 3-11
Output of the sensor-graphing program.

You don't just want to respond to events, though. You might have a fickle cat who jumps on and off the mat a lot. 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 don't respond to any input during that time. Once the interval's over, start looking for input again.

» Add the following new variable to the beginning of your program. Determine the threshold value by watching the sensor value without the cat on the mat, and picking a number that's higher.

```
int threshold = 250;        // above this number, the cat is on the mat.
```

» Then put this code in your serialEvent() method, right after drawGraph() (new lines are shown in blue). Remove or comment out the line in the serialEvent() method that prints the sensor value.

```
  drawGraph(prevSensorValue, sensorValue);

  if (sensorValue > threshold ) {
    // if the last reading was less than the threshold,
    // then the cat just got on the mat.
    if (prevSensorValue <= threshold) {
      println("cat on mat");
      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) {
      println("cat not on mat");
    }
  }
```

» 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, you should see messages in the console area saying when the cat jumps on or off the mat, and when a mail would be sent. Your cat may be fickle or may take his time settling on the mat, which can result in several mail messages as he gets comfortable. To avoid this, you want to change the sketch so that once it sends a message, it doesn't send any others for an acceptable period. You can do this by modifying the sendMail() method to keep track of when the last message was sent. Here's how to do it.

Tame It Every time the program sends a mail message, it should take note of the time. Add a few new variables at the beginning of the program.

```
int currentTime = 0;      // the current time as a single number
int lastMailTime = 0;     // last time you sent a mail
int mailInterval = 60;    // minimum seconds between mails
String mailUrl = "http://www.example.com/cat-script.php";
```

» Add one line at the beginning of the draw() method to update currentTime continuously.

```
void draw() {
  currentTime = hour() * 3600 + minute() * 60 + second();
```

» Now, modify the sendMail() method as shown in the code at right.

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

```
void sendMail() {
  // how long has passed since the last mail:
  int timeDifference = currentTime - lastMailTime;

  if ( timeDifference > mailInterval) {
    String[] mailScript = loadStrings(mailUrl);
    println("results from mail script:");
    println(mailScript);

    // save the current minute for next time:
    lastMailTime = currentTime;
  }
}
```

When you run the sketch this time, you'll see an error message like this when you cross the threshold:

```
cat on mat

results from mail script:The file "http://www.example.com/cat-script.php" is missing or inaccessible, make sure the URL is valid or that the file has been added to your sketch and is readable.
```

The sketch is now making an HTTP GET request to call a PHP script that's not there. In the next section, you'll write that script.

Sending Mail from the Cat

Processing doesn't have any libraries for sending and receiving email, so you can call a PHP script to do the job. Processing has a simple command to make HTTP GET requests: loadStrings(). The same technique can be used to call other PHP scripts or web URLs from Processing. POST requests are a little more complicated; you'll see how to do that later.

Save the following PHP script to your server with the name **cat-script.php**. Test it from a browser. When you call it, you should get an email in your inbox. Some mail servers may require that you send mail only from your proper account name. If that's the case, replace cat@example.com with the email address for your account name on the server on which the script is running.

Once you've confirmed that this script is running, go back to the Processing sketch. Double-check that the mailUrl variable is the URL of the script, and run the sketch. When the cat sits on the mat now, the sketch will call the mailer script and you should get an email from the cat. The console output will look like this:

```
cat on mat
results from mail script:
[0] "TO: you@example.com"
[1] "FROM: cat@example.com"
[2] "SUBJECT:the cat"
[3] ""
[4] "The cat is on the mat at http://www.example.com/catcam."
[5] ""
```

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

   Sends an email if sensorValue is above a threshold value.
*/
   // form the message:
   $to = "you@example.com";
   $subject = "the cat";
   $message = "The cat is on the mat at http://www.example.com/catcam.";
   $from = "cat@example.com";

   // send the mail:
   mail($to, $subject, $message, "From: $from");
   // reply to processing:
   echo "TO: " .$to;
   echo "\nFROM: " .$from;
   echo "\nSUBJECT:" .$subject;
   echo "\n\n" .$message . "\n\n";
?>
```

» 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.

Now you're getting somewhere! You're able to send mail triggered by a physical event—the cat sitting on the mat. This is a good moment to stop and celebrate your achievement and pet the cat. In the next section, you're going to use a webcam to make your cat Internet-famous.

X

Making a Web Page for the Cat Cam

» Next, you need a web page for the cat cam. Take a picture of your cat and upload the image to the directory containing your script with the filename **catcam.jpg**. Once you know the image is there and visible, frame it with a web page in the same directory, called **index.html**. Here is a bare-bones page that will automatically refresh itself in the user's browser every five seconds, as indicated by the meta tag in the head of the document. Feel free to make the page as detailed as you want, but keep the meta tag in place.

```
<html>
<head>
   <title>Cat Cam</title>
   <meta http-equiv="refresh" content="5">
</head>
<body>
   <center>
      <h2>Cat Cam</h2>
         <img src="catcam.jpg">
   </center>
</body>
</html>
```

Refine It The previous page is really simple, but as you can see, it reloads the entire page each time. However, if you want only the image to reload, you can do something fancier with some CSS DIV tags and a little JavaScript.

This page is a bit more complex. Only the image reloads in this page. The other elements stay stable.

Once you've made the Cat Cam page to your liking, it's time to automate the process of taking the picture and uploading it. First, you'll make an uploader PHP script. Then you'll modify the Processing sketch to take a picture and upload it via the PHP script. Figure 3-12 shows the image uploaded to the Cat Cam page.

X

```
<html>
<head>
   <title>Cat Cam</title>
   <script type="text/javascript">
      function refresh() {
         var refreshTime = 5 * 1000;     // 5000 ms
         var thisImage = "catcam.jpg";   // the image location
         var today = new Date();         // the current time
         // add the time to the end of the image string
         //to make a unique URL:
         document.images["pic"].src=thisImage+"?"+today;
         // reload if the images are loaded:
         if(document.images) {
            window.onload=refresh;
         }
         // if the time is up reload the image
         t=setTimeout('refresh()',refreshTime);
      }
   </script>
</head>
<body onload="refresh()">
   <center>
      <h2>Cat Cam</h2>
<img src="catcam.jpg" id="pic">
   </center>
</body>
</html>
```

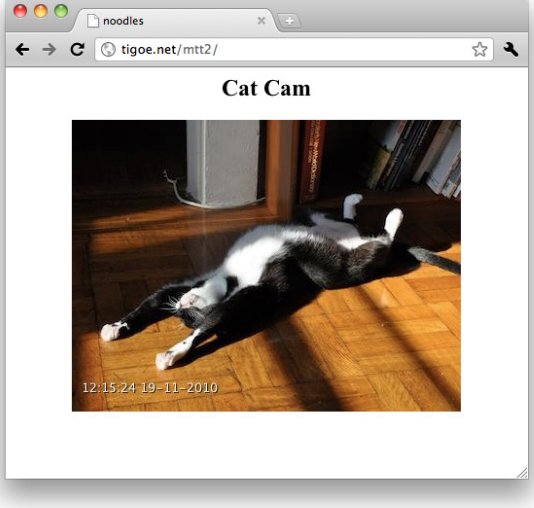


Figure 3-12
Regardless of which option you use, the Cat Cam page will look like this.

Uploading Files to a Server Using PHP

Next, you're going to write a script that uploads images to the server. PHP has a predefined variable, $_FILES, that allows you to get information about any files the user attempts to upload via HTTP. Like $_REQUEST, which you saw earlier, $_FILES is an array, and each of the array elements is a property of the file. You can get the name of the file, the size of the file, and the file type, which will be useful when you write a script that takes only JPEG files below a certain size.

Try It Here's a PHP script with an HTML form that uploads a file. The PHP script doesn't do anything much so far—it just shows you the results of the HTTP request. Save this to your server as **save2web.php**.

```
<?php
if (isset($_FILES)) {
    print_r($_FILES);
}
?>

<html>
<body>

<form action="save2web.php" method="post"
enctype="multipart/form-data">
<label for="file">Filename:</label>
<input type="file" name="file" id="file" />
<br />
<input type="submit" name="submit" value="Upload" />
</form>

</body>
</html>
```

View this in a browser, and you'll get a simple form with a file chooser button and an upload button. Choose a JPEG image file and upload, and you'll get a response that looks like this:

```
Array ( [file] => Array ( [name] =>
catcam.jpg [type] => image/jpeg [tmp_
name] => /tmp/phpoZT3BS [error] => 0
[size] => 45745 ) )
```

This is the $_FILES variable printed out for you.

» Replace the PHP part of this script (the part between <?php and ?>) with the following. This script checks to see that the uploaded file is a JPEG file with fewer than 100 kilobytes, and it saves it in the same directory as the script.

```
<?php
if (isset($_FILES)) {
    // put the file parameters in variables:
    $fileName =  $_FILES['file']['name'];
    $fileTempName = $_FILES['file']['tmp_name'];
    $fileType = $_FILES['file']['type'];
    $fileSize = $_FILES['file']['size'];
    $fileError = $_FILES['file']['error'];

    // if the file is a JPEG and under 100K, proceed:
    if (($fileType == "image/jpeg") && ($fileSize < 100000)){
        // if there's a file error, print it:
        if ( $fileError > 0){
            echo "Return Code: " . $fileError . "<br />";
        }
```



Now when you load this script in a browser and upload a file, you'll be able to see the file in the directory afterwards. Try uploading a new JPEG of the cat, then reload the browser window of the **index.html** page you made earlier. You should see the same page with the new image.

Continued from previous page.

```
    // if there's no file error, print some HTML about the file:
     else {
        echo "Upload: " . $fileName . "<br />";
        echo "Type: " . $fileType . "<br />";
        echo "Size: " . ($fileSize / 1024) . " Kb<br />";
        echo "Temp file: " . $fileTempName . "<br />";

        // if the file already exists,
        // delete the previous version:
        if (file_exists($fileName)) {
            unlink($fileName);
        }
        // move the file from the temp location to
        // this directory:
        move_uploaded_file($fileTempName, $fileName);
        echo "Uploaded file stored as: ".$fileName;
    }
  }
  // if the file's not a JPEG or too big, say so:
  else {
      echo "File is not a JPEG or too big.";
  }
}
?>
```

Capturing an Image and Uploading It Using Processing

Now that you've got an uploader script on the server, you need a program on your local computer to capture a picture of the cat and to call the script. There are several automated webcam applications on the market, but it's fun to do it on your own. This section will describe how you can do it in Processing using two external libraries: the Processing video library and the Processing net library.

Before you get going, there are a few things you'll need to do. First, make sure your webcam can be read by both your computer and by Processing.

If you're using Mac OS X, you're all set. The Processing video library works with QuickTime, which comes with your computer. Open a video applicatio,n like Photo Booth, just to make sure it's working, then you're ready to program.

For Windows 7 users, you'll need to install QuickTime, if you don't have it already. Download it from http://www.apple.com/quicktime and follow the installer directions. You'll also need a VDIG, which is a software library that allows Processing to connect to the webcam through QuickTime. You can download WinVDIG from http://www.eden.net.nz/7/20071008/. Use version 1.0.1, as it's the most stable as of this writing. Finally, you'll need to change the Compatibility settings for Processing. Open the Processing application directory, right-click on the Processing application icon, and choose Properties. In the application's Properties window, choose the Compatibility tab. In that tab, click the checkbox marked "Run this program in compatibility mode for:", and from the associated drop-down menu, choose "Windows XP (service pack 3)". Then click Apply, and you're ready to move on. The Processing team hopes to change this soon, so it may be much simpler by the time you read this.

For Ubuntu Linux users, you're out of luck this time. The Processing video library is not currently supported under Linux, though the developers welcome patches for this.

Try It Here's a sketch that uses the video library to paint a live image from the webcam to the screen. Run it and you'll see yourself (or your cat).

```
/**
 Image capture and upload
 Context: Processing
 */
import processing.video.*;    // import the video library
Capture myCam;                // the camera

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

  // For a list of cameras on your computer, use this line:
  println(Capture.list());

  // use the default camera for capture at 30 fps:
  myCam = new Capture(this, width, height, 30);
}

void draw() {
  // if there's data from the camera:
  if (myCam.available()) {
    myCam.read();      // read the camera image
    set(0, 0, myCam);  // draw the camera image to the screen
  }
}
```

» Add these lines after the set() command in the draw() method to draw a timestamp on the screen (new lines are shown in blue).

```
void draw() {
  // if there's data from the camera:
  if (myCam.available()) {
    myCam.read();      // read the camera image
    set(0, 0, myCam);  // draw the camera image to the screen

    // get the time as a string:
    String timeStamp = nf(hour(), 2) + ":" + nf(minute(), 2)
      + ":" + nf(second(), 2) + " " + nf(day(), 2) + "-"
        + nf(month(), 2) + "-" +  nf(year(), 4);

    // draw a dropshadow for the time text:
    fill(15);
    text(timeStamp, 11, height - 19);
    // draw the main time text:
    fill(255);
    text(timeStamp, 10, height - 20);
  }
}
```

Now add a global variable at the top of your sketch for a filename, and add a method at the end of the sketch to save the window image to a JPEG file when you hit any key (new lines are shown in blue).

```
import processing.video.*;    // import the video library
Capture myCam;                // the camera
String fileName = "catcam.jpg";

// setup and draw methods go here

void keyReleased() {
  PImage img = get();
  img.save(fileName);
}
```

Save this sketch before you run it. When you run it, hit any key. Then check the sketch folder, and you should see a JPEG image generated by the sketch, like Figure 3-13.



Figure 3-13
The output of the Cat Cam Processing sketch.

Next, it's time to add a method that can make an HTTP POST request. For that, you'll need the Processing network library. Like the serial library, it adds some functions to the core of Processing. The serial library allowed you to access the serial ports; the network library allows you to make network connections and a few more global variables. Add these before the setup() method.

```
import processing.net.*;

String pictureScriptUrl = "/save2web.php";
String boundary = "----H4rkNrF";
Client thisClient;
```

Now you need a method to formulate and send the actual POST request. To do this, open a new Client connection to the server, then send the request. In this method, the content of the request is broken into three parts: everything that comes before the file itself; the actual file, which is loaded into a byte array; and the end of the request that comes after the file. Add this method at the end of your sketch.

```
void postPicture() {
  // load the saved image into an array of bytes:
  byte[] thisFile =loadBytes(fileName);

  // open a new connection to the server:
  thisClient = new Client(this, "www.example.com", 80);
  // make an HTTP POST request:
  thisClient.write("POST " + pictureScriptUrl + " HTTP/1.1\n");
  thisClient.write("Host: www.example.com\n");
  // tell the server you're sending the POST in multiple parts,
  // and send a unique string that will delineate the parts:
  thisClient.write("Content-Type: multipart/form-data; boundary=");
  thisClient.write(boundary + "\n");

  // form the beginning of the request:
  String requestHead ="\n--" + boundary + "\n";
  requestHead +="Content-Disposition: form-data; name=\"file\"; ";
  requestHead += "filename=\"" + fileName + "\"\n";
  requestHead +="Content-Type: image/jpeg\n\n";

  // form the end of the request:
  String tail ="\n--" + boundary + "--\n\n";
```

Once you've got the three pieces of the request, calculate the total number of bytes by adding the length of the two strings and the byte array. Next, send the content length, and then the content itself. Finally, you close the connection by stopping the client.

```
  // calculate and send the length of the total request,
  // including the head of the request, the file, and the tail:
  int contentLength = requestHead.length() + thisFile.length + tail.length();
  thisClient.write("Content-Length: " + contentLength + "\n\n");

  // send the header of the request, the file, and the tail:
  thisClient.write(requestHead);
  thisClient.write(thisFile);
  thisClient.write(tail);

  // close the client:
  thisClient.stop();
}
```

Finally, add a call to this new method in the keyReleased() method (new lines are shown in blue).

```
void keyReleased() {
  PImage img = get();
  img.save(fileName);
  postPicture();
}
```

Save the sketch and run it. Now when you type any key, the sketch will not only save the image locally, but also upload it to the **catcam** directory on your sever, via the **save2web.php** script.

Anatomy of a Multipart POST Request

HTML forms that use HTTP to enable file uploading are done with the POST request, as well as a special content type, multipart/form-data, so the server knows how to interpret what comes through. Each element from the form is sent in a separate part by a unique string. Here's what the server might see from the uploader script you wrote earlier:

```
POST /catcam/save2web.php HTTP/1.1
Host: www.example.com
Content-Type: multipart/form-data; boundary=----H4rkNrF
Content-Length: 40679

------H4rkNrF
Content-Disposition: form-data; name="submit"

Upload
------H4rkNrF
Content-Disposition: form-data; name="file"; filename="catcam.jpg"
Content-Type: image/jpeg

[actual bytes of the file go here]

------H4rkNrF--
```

» Boundary: has to be a unique string, but you can use whatever you wish. Comes before each part.

» Value for this part.

» Content-Disposition: says what this part is. For files, also includes the local path to the file.

» Final boundary ends with an extra two dashes.

Putting It All Together

Finally, it's time to combine the Processing sketch, which reads the serial port and calls the mailer script, with the one that takes the image and calls the image uploader script. The final sketch will work as shown in Figure 3-14. All the action is driven by the serial input of sensor data. If the received value is over the threshold, there's a cat there, so the sketch takes a picture. If it's been an appropriate time since the last picture, the picture is uploaded. If the sensor value has just changed from below the threshold to above, and enough time has passed since the last mail, the sketch sends you mail that the cat just jumped on the mat.

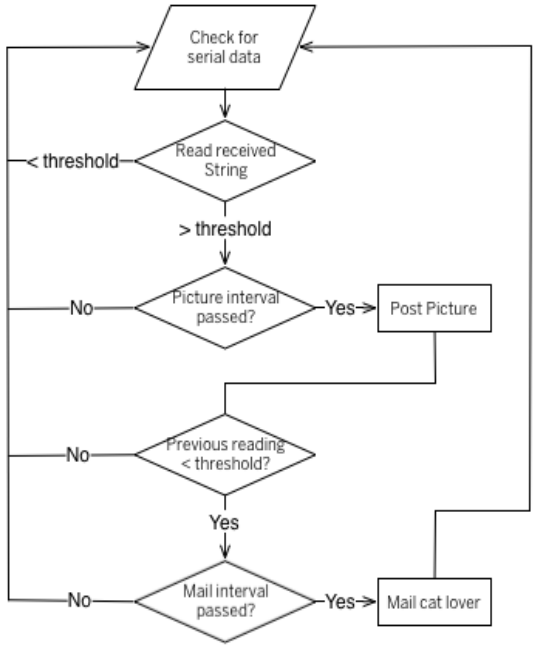


Figure 3-14
The flowchart of the Cat Cam sketch.

Try It Start by combining the list of global variables from both sketches. Here's what it should look like.

```
/*
 Cat webcam uploader/emailer
 Context: Processing

 takes a webcam image continually, uploads it, and
 mails you when it receives a serial string above a given value
 */

// import the libraries you need:  Network, serial, video:
import processing.serial.*;
import processing.video.*;
import processing.net.*;

Serial myPort;              // the serial port
float sensorValue = 0;      // the value from the sensor
float prevSensorValue = 0;  // previous value from the sensor
int threshold = 250;        // above this number, the cat is on the mat.

int currentTime = 0;        // the current time as a single number
int lastMailTime = 0;       // last minute you sent a mail
int mailInterval = 60;      // minimum seconds between mails
String mailUrl = "http://www.example.com/cat-script.php";
int lastPictureTime = 0;    // last minute you sent a picture
int pictureInterval = 10;   // minimum seconds between pictures

Capture myCam;                  // camera capture library instance
String fileName = "catcam.jpg"; // file name for the picture

// location on your server for the picture script:
String pictureScriptUrl = "/save2web.php";
String boundary = "----H4rkNrF"; // string boundary for the POST request

Client thisClient;          // instance of the net library
```

Now combine the setup() methods like so.

```
void setup() {
  size(400,300);
  // list all the available serial ports
  println(Serial.list());

  // I know that the first port in the serial list on my Mac is always my
  // Arduino, so I open Serial.list()[0]. Open whatever port you're using
  // (the output of Serial.list() can help; they 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 newline (ASCII 10):
  myPort.bufferUntil('\n');
```

Continued from previous page.

```
  // set initial background and smooth drawing:
  background(#543174);
  smooth();

  // For a list of cameras on your computer, use this line:
  println(Capture.list());

  // use the default camera for capture at 30 fps:
  myCam = new Capture(this, width, height, 30);
}
```

The draw() method paints the camera image to the screen and adds a timestamp.

```
  void draw () {
  // make a single number from the current hour, minute, and second:
  currentTime = hour() * 3600 + minute() * 60 + second();

  if (myCam.available() == true) {
    // draw the camera image to the screen:
    myCam.read();
    set(0, 0, myCam);

    // get the time as a string:
    String timeStamp = nf(hour(), 2) + ":" + nf(minute(), 2)
      + ":" + nf(second(), 2) + " " + nf(day(), 2) + "-"
        + nf(month(), 2) + "-" +  nf(year(), 4);

    // draw a dropshadow for the time text:
    fill(15);
    text(timeStamp, 11, height - 19);
    // draw the main time text:
    fill(255);
    text(timeStamp, 10, height - 20);
  }
}
```

The main action happens in the serialEvent(), just as shown in the flowchart in Figure 3-14. If the sensor reading is greater than the threshold, the sketch takes a picture. Every five seconds, it uploads the picture. If the sensor reading just changed, it calls the sendMail() method.

```
void serialEvent (Serial myPort) {
  // get the ASCII string:
  String inString = myPort.readStringUntil('\n');

  if (inString != null) {
    // trim off any whitespace:
    inString = trim(inString);
    // convert to an int and map to the screen height:
    sensorValue = float(inString);
    sensorValue = map(sensorValue, 0, 1023, 0, height);

    if (sensorValue > threshold ) {
      if (currentTime - lastPictureTime > pictureInterval) {
```

Continued from previous page.

```
      PImage thisFrame = get();
      thisFrame.save(fileName);
      postPicture();
      lastPictureTime = currentTime;
    }

    // if the last reading was less than the threshold,
    // then the cat just got on the mat.
    if (prevSensorValue <= threshold) {
      println("cat on mat");
      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) {
      println("cat not on mat");
    }
  }
  // save the current value for the next time:
  prevSensorValue = sensorValue;
 }
}
```

The sendMail() method checks to see whether enough time has passed since the last mail before it sends. Then it calls the **cat-mail.php** script using the loadStrings() function to make an HTTP GET request.

```
void sendMail() {
  // how long has passed since the last mail:
  int timeDifference = currentTime - lastMailTime;

  if ( timeDifference > mailInterval) {
    String[] mailScript = loadStrings(mailUrl);
    println("results from mail script:");
    println(mailScript);

    // save the current minute for next time:
    lastMailTime = currentTime;
  }
}
```

Finally, the postPicture() method takes the last image saved. The method uploads it by calling the **save2web.php** script using an HTTP POST request.

```
void postPicture() {
  // load the saved image into an array of bytes:
  byte[] thisFile =loadBytes(fileName);

  // open a new connection to the server:
  thisClient = new Client(this, "www.example.com", 80);
  // make an HTTP POST request:
  thisClient.write("POST " + pictureScriptUrl + " HTTP/1.1\n");
  thisClient.write("Host: www.example.com\n");
  // tell the server you're sending the POST in multiple parts,
  // and send a unique string that will delineate the parts:
  thisClient.write("Content-Type: multipart/form-data; boundary=");
  thisClient.write(boundary + "\n");

 // form the beginning of the request:
  String requestHead ="\n--" + boundary + "\n";
  requestHead +="Content-Disposition: form-data; name=\"file\"; ";
  requestHead += "filename=\"" + fileName + "\"\n";
  requestHead +="Content-Type: image/jpeg\n\n";

  // form the end of the request:
  String tail ="\n\n--" + boundary + "--\n\n";

  // calculate and send the length of the total request,
  // including the head of the request, the file, and the tail:
  int contentLength = requestHead.length() + thisFile.length
      + tail.length();
  thisClient.write("Content-Length: " + contentLength + "\n\n");

  // send the header of the request, the file, and the tail:
  thisClient.write(requestHead);
  thisClient.write(thisFile);
  thisClient.write(tail);
}
```

One Final Test

To test this code, open the browser to the Cat Cam page, and then run the sketch. On startup, it should take a picture and upload it. A few seconds later, the Cat Cam page will refresh itself, and you'll see that image. Then get the cat to jump on the mat. The sketch will take another image and upload it, and will then send you an email. Check your mail to see whether there's a new message from the cat.

As long as the cat stays on the mat, the sketch will upload a new image every five seconds. Watch for a minute or so to see the image change a few times, and then remove the cat from the mat. You should get another message in the console from the call to the mailer script. See whether the image changes once the cat is off the mat. It shouldn't; the image that remains should be the last one of the cat—that is, until the cat jumps on the mat again. The timestamp in that image will give you an idea when the cat was last on the mat.

When all that works, take a breath and admire your work. Figure 3-15 shows the finished cat bed. Congratulations! You've just made your first Internet-connected project. You and your cat can now be in constant contact.
X



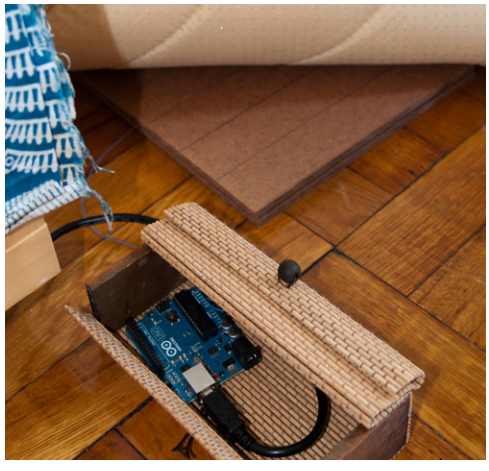




Figure 3-15
The finished cat bed (at right) and a detail of the sensor pad, which sits under the cat bed itself. A bamboo jewelry box houses the electronics—and matches the furniture. The USB cable runs to the computer. Make sure to secure the wires thoroughly, or the cat may try to chew on them.

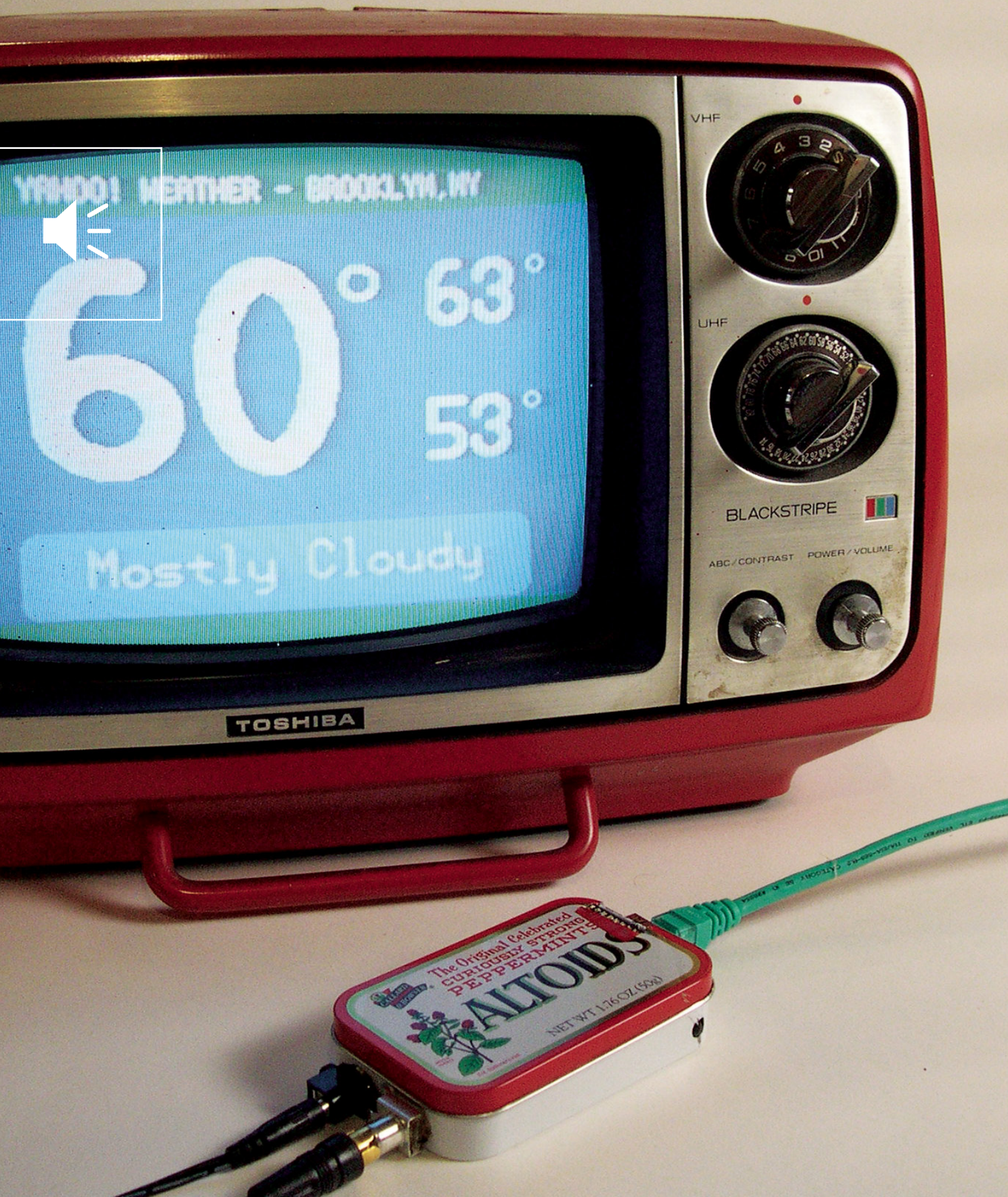
Conclusion

Now you have 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 dependable route through the Internet from client to server. Client and server applications swap 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 soon get the chance to do it without a personal computer. In the next chapter, you'll connect a microcontroller to the Internet directly using an Ethernet interface for 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
NET WT 1.76 OZ (50g)

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://uncommonprojects.com/site/play/ybox-2) 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 WiFi. Ceiva, eStarling, and others make picture frames with WiFi 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, such as 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 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

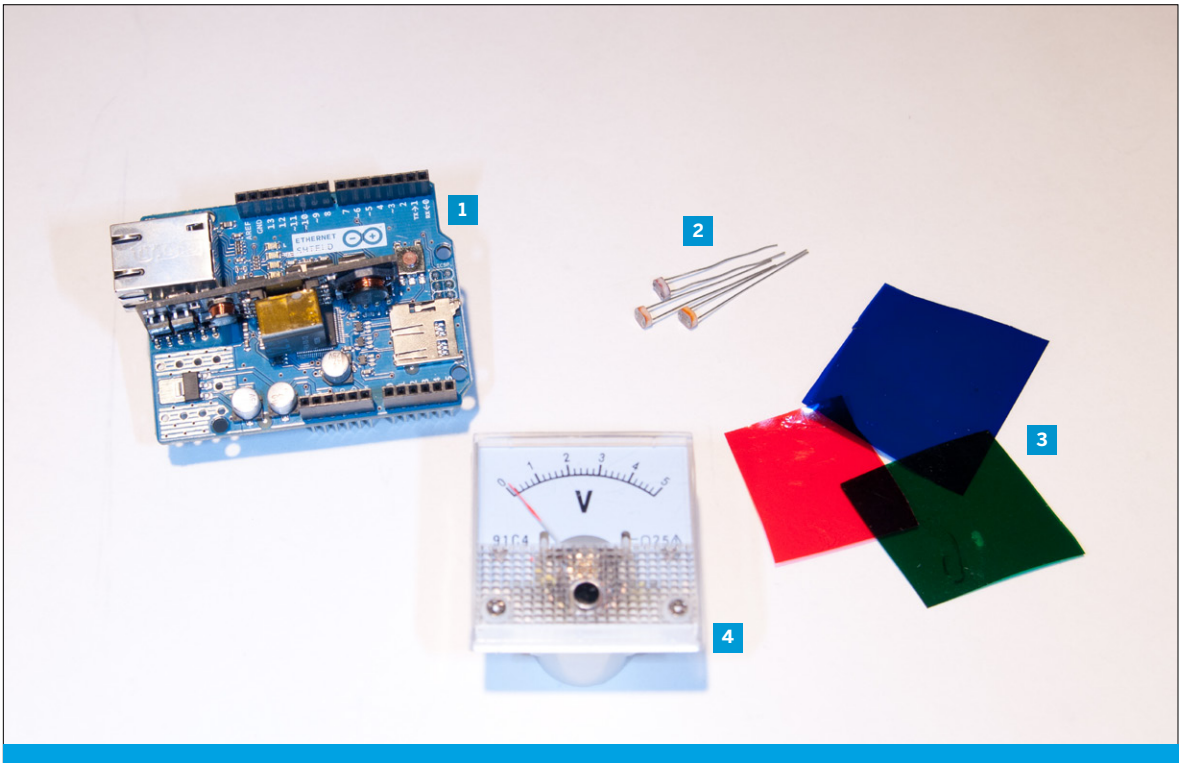


Figure 4-1. New parts for this chapter: **1.** Arduino Ethernet shield—this one has a Power over Ethernet module on board **2.** Photocells **3.** Red, green, and blue lighting filter gels **4.** Voltmeter. You can use an off-the-shelf one, but it's better if you can find one that's antique. Don't forget plenty of male header pins for the breakout boards.

Supplies for Chapter 4

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **RS** RS (www.rs-online.com)
- **SF** SparkFun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 6: Hello Internet! Daylight Color Web Server

» **1 Arduino Ethernet board** **A** A000050
Alternatively, an Uno-compatible board (see Chapter 2) with an Ethernet shield will work.
SF DEV-09026, **J** 2124242, **A** A000056, **AF** 201, **F** 1848680

» **1 Ethernet connection to the Internet** Your home router most likely has Ethernet jacks in the back. If you've hooked up your computer to the Internet using Ethernet, you know where the ports are.

» **3 10-kilohm resistors** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906

» **3 photocells (light-dependent resistors)** **D** PDV-P9200-ND, **J** 202403, **SF** SEN-09088, **F** 7482280, **RS** 234-1050

» **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

» **3 lighting filters** One primary red, one primary green, and one primary blue. Available from your local lighting- or photo-equipment supplier.

PROJECT 7: Networked Air-Quality Meter

» **1 Arduino Ethernet board** **A** A000050
Alternatively, an Uno-compatible board (see Chapter 2) with an Ethernet shield will work.
SF DEV-09026, **J** 2124242, **A** A000056, **AF** 201, **F** 1848680

» **1 Ethernet connection to the Internet** Your home router most likely has Ethernet jacks in the back. If you've hooked up your computer to the Internet using Ethernet, you know where the ports are.

» **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

» **1 voltmeter** Get a nice-looking antique one if you can. Ideally, you want a meter that reads a range from 0–5V, or 0–10V at most.
SF TOL-10285, **F** 4692810, **RS** 244-890

» **4 LEDs** **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590

» **4 220-ohm resistors** **D** 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842

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 select 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. Others, like the Beagle Board, are a full network computer on a single circuit board. These are very powerful, and if you're an experienced network programmer, they present a very comfortable programming environment. However, they don't make it easy to add sensors and actuators, and for people not experienced with network programming, they present a steep learning curve. Others—like Lantronix' modules, the XPort, XPort Direct, MatchPort, and WiPort—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. These are serial-to-Ethernet modems. They work much like the Bluetooth modems you used in Chapter 2, but they have a different serial protocol. These are a comfortable beginning place, and I used them extensively in the first edition of this book. For this edition, however, you'll learn about Ethernet modules that have a synchronous serial interface. These offer the ease-of-use of the Serial-to-Ethernet modules, but they don't take up

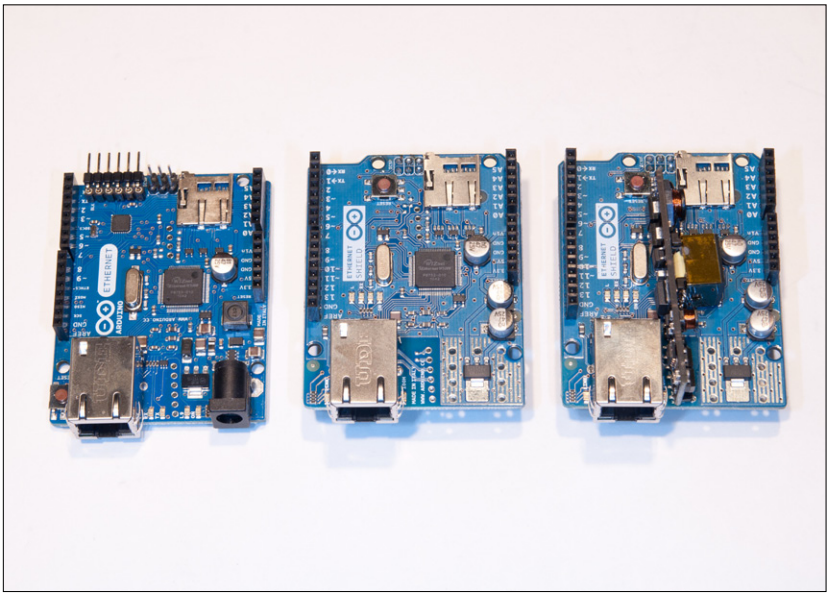
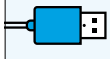


Figure 4-2
The Arduino Ethernet (left), an Arduino Ethernet shield (center), and an Ethernet shield with power-over-Ethernet module attached (right). You can use any of these for the Ethernet projects in this book.

an asynchronous serial port on your microcontroller. It means you can still use the serial port for debugging or communicating with other devices.

There are two options for Ethernet connections to an Arduino. One is the Arduino Ethernet board, which is an Arduino board that has an Ethernet module on the board itself. The Ethernet connector replaces the USB connector found on the standard Arduinos. To program this board, you also need an FTDI-style USB-to-Serial adapter. The other option is an add-on board for regular Arduino modules, called an Ethernet shield. There are a few versions of it on the market. The Arduino Ethernet shield uses an Ethernet chip from WizNet, the W5100. This shield has a built-in SD memory card slot as well, like the Arduino Ethernet board. The Adafruit Ethernet shield also uses the W5100 chip, or it can use the Lantronix serial-to-Ethernet modules mentioned earlier. The projects in this chapter will work with the Arduino Ethernet, with the Arduino Ethernet shield, or with the Adafruit Ethernet shield with the W5100 module, which are shown in Figure 4-2. They will not work with the Lantronix module, however.
X



Introducing Serial Peripheral Interface (SPI)

The Ethernet shield and Arduino Ethernet communicate with their Ethernet controllers using a form of synchronous serial communication called Serial Peripheral Interface, or SPI. SPI, along with another synchronous serial protocol, Inter-Integrated Circuit or I2C (sometimes called Two-Wire Interface, or TWI), are two of the most common synchronous serial protocols you'll encounter.

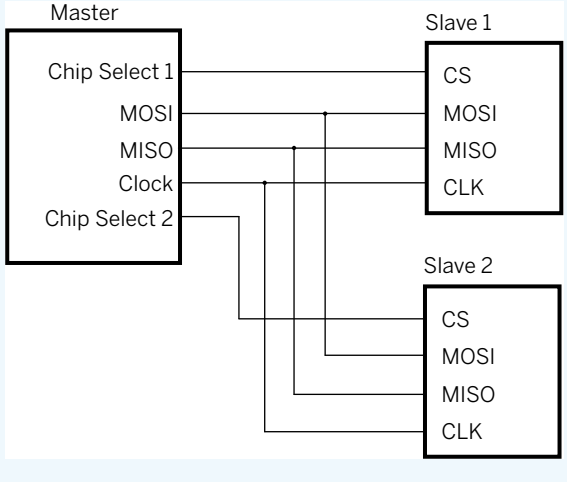
Synchronous serial protocols all feature a controlling device that generates a regular pulse, or clock signal, on one pin while exchanging data on every clock pulse (see Chapter 2). The advantage of a synchronous serial protocol is that it's a bus: you can have several devices sharing the same physical connections to one master controller. Each protocol implements the bus in a different way.

SPI connections have three or four connections between the controlling device (or master device) and the peripheral device (or slave), as follows:

Clock: The pin that the master pulses regularly.
Master Out, Slave In (MOSI): The master device sends a bit of data to the slave on this line every clock pulse.
Master In, Slave Out (MISO): The slave device sends a bit of data to the master on this line every clock pulse.
Slave Select (SS) or Chip Select (CS): Because several slave devices can share the same bus, each has a unique connection to the master. The master sets this pin low to address this particular slave device. If the master's not talking to a given slave, it will set the slave's select pin high.

If the slave doesn't need to send any data to the master, there will be no MISO pin.

The Arduino SPI library uses pin 11 for MOSI, pin 12 for MISO, and pin 13 for Clock. Pin 10 is the default Chip Select pin, but you can use others, as you'll see. For example, the Arduino Ethernet and Ethernet shield have two slave devices on their SPI bus: the WizNet chip and the SD card. They are both connected to pins 11, 12, and 13 for MISO, MOSI, and Clock. The WizNet module uses pin 10 for its Chip Select, while the SD card uses pin 4 for its Chip Select.



Project 6

Hello Internet!

To use any network module, you first need to connect it to the network. That's the goal here. In this project, you'll make a very simple web server on your Arduino that serves a web page whose background color changes with the color of the light where the Arduino is located.

MATERIALS

- **1 Arduino Ethernet or**
- **1 Arduino Ethernet shield and 1 Arduino microcontroller module**
- **1 Ethernet connection to the Internet**
- **3 10-kilohm resistors**
- **3 photocells (light-dependent resistors)**
- **1 solderless breadboard**
- **3 lighting filters**

Making the Connections

If you're using the Arduino Ethernet board, just plug it into a USB-to-Serial adapter and it will show up in the list of serial ports like a regular Arduino. The USB-to-Serial adapter will supply it with power like a normal USB cable. Programming it is the same as with any other Arduino, but in the Tools→Board menu, choose Arduino Uno or Arduino Ethernet.

The Ethernet shield, like many shields for the Arduino, is very easy to connect to your controller. Just plug it into the board. Both the Ethernet module and the shield use pins 10, 11, 12, and 13 for communication to the Ethernet controller, and pins 4, 11, 12, and 13 for communication to the SD card, so you can't use those pins for other inputs or outputs. Connect the Ethernet module to your router using an Ethernet cable. Figure 4-3 shows the Arduino Ethernet shield's connections to the microcontroller.

You can use the Arduino Ethernet or Ethernet ahield with a standard Arduino interchangeably, so from here on out, I'll just use "Ethernet module" to refer to either.

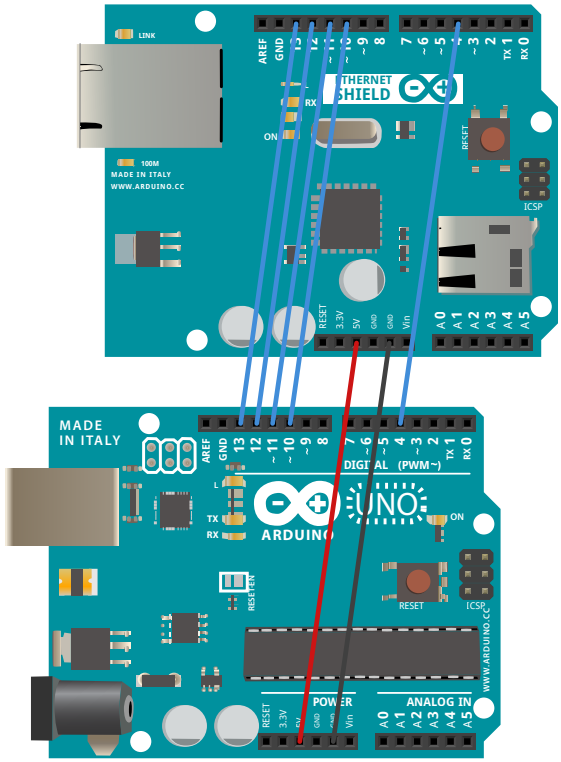


Figure 4-3
The connections between the Ethernet shield and the Arduino controller when they're stacked together.

To read the color of the ambient light, use three photocells and cover each one with a different color of lighting filter: red, green, and blue. You can get these at many lighting-supply stores, photo supply-stores, or art stores, or you can use any translucent colored plastic you have around the house. You don't have to be scientific here. Figure 4-4 shows the connection between the Arduino Ethernet module and the three photocells.

How the Ethernet Library Works

To begin, you're going to try a couple simple programs using the Ethernet library for Arduino. This library lets you control the Ethernet module using methods similar to those you use for printing to a serial port. There are two types of software objects you'll make: servers and

clients. A server waits for connections from remote devices on the Internet and allows them to connect to the Ethernet module, just like regular servers. A client initiates a connection to remote devices, makes requests, and delivers the replies, just like a regular client. Both servers and clients can be read from and written to, using the read(), write(), print(), and println() commands that you've already seen with the Serial library. There's also an available() command, as in the Serial library, to see whether there's any new data available from the server or client in question. The client object also has a connected() command that tells you whether it's connected to a remote server. This will be useful later, when you're trying to connect to a remote server to get data. Here, you'll get started with a simple server sketch.

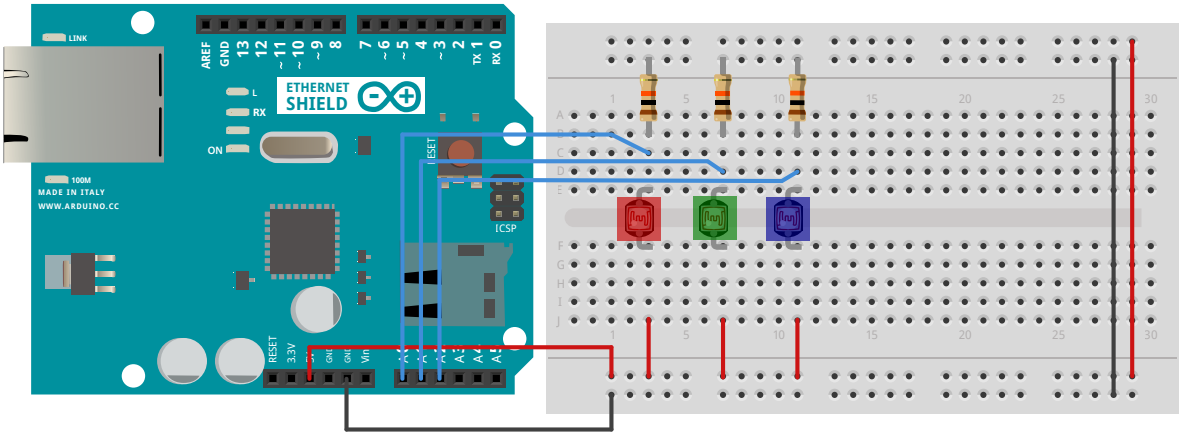
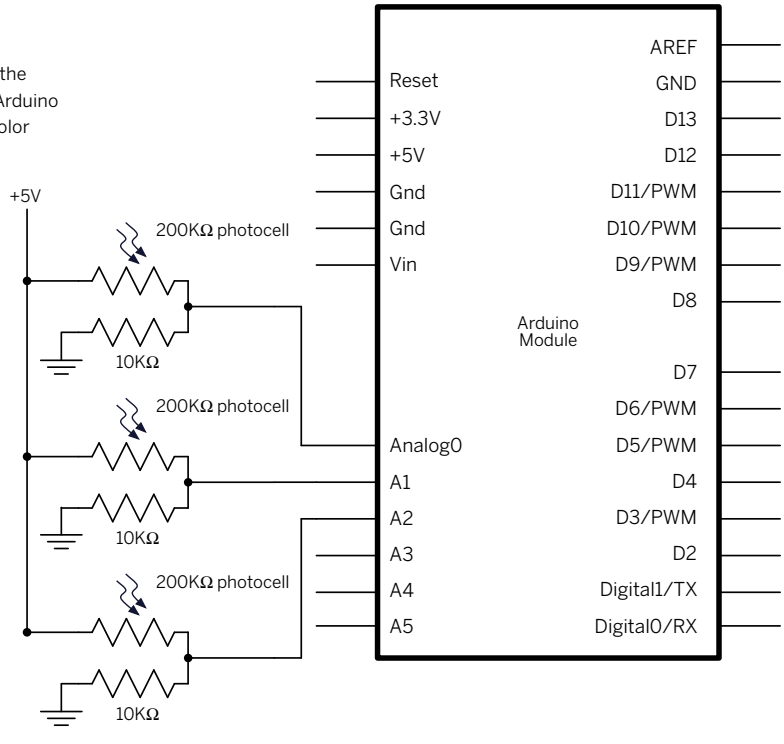


Figure 4-4
The RGB server circuit. The connections are the same for either the Arduino Ethernet or the Arduino and Ethernet shield combo. Note the three color filters over the photocells.



Before you can write your program, you need to establish some basic information about how your Ethernet module will connect to the Internet. Just as you did in Chapter 3, you'll need the device's Media Access Control (MAC) address. That's the hardware address of your Ethernet controller. The Arduino Ethernet modules have a six-byte address on the back, written in hexadecimal notation, that you can use. If the sticker's missing for any reason, you can make up your own MAC address, or use the generic one you find in the examples below. You'll also need to know the router's address (aka the gateway address because your router is the gateway to the rest of the Internet), as well as the address that your device will use on the router's subnet.

Your device's IP address will be similar to your router's address, probably using the same three numbers for the start of the address, but a different number for the last. For example, if your router's local address is 192.168.1.1, you can use an address like 192.168.1.20 for your Ethernet module—as long as no other device connected to the router is using the same address.

When a router assigns addresses to its connected devices, 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. With the Ethernet module, you'll assign it similarly.

Once you know your MAC address, your router's address, your IP address, and your subnet mask, you're ready to go. X

Try It

To get started, you need to include the SPI library and the Ethernet library to control the module. You also need to initialize a variable to hold a server instance. The server will run on port 80, just as most web servers do.

You also need four variables for the MAC address, IP address, gateway address, and subnet mask. These last four will be arrays of bytes, one byte for each byte in the respective addresses.

```
/*
  Web  Server
  Context: Arduino

 */

#include <SPI.h>
#include <Ethernet.h>

EthernetServer server(80);

byte mac[] = { 0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress gateway(192,168,1,1);
IPAddress subnet(255,255,255,0);
IPAddress ip(192,168,1,20);
```

» Change these to match your own device and network.

» In the setup() method, you'll start the Ethernet module and the server. Open the serial connection as well, for debugging purposes.

```
void setup()
{
  // start the Ethernet connection and the server:
  Ethernet.begin(mac, ip, gateway, subnet);
  server.begin();
  Serial.begin(9600);
}
```

In the main loop, you'll spend all your time listening for a connection from a remote client. When you get a connection, you'll wait for the client to make an HTTP request, like you saw in Chapter 3. This loop won't reply to the client, but it will show you what the client requests.

```
void loop()
{
  // listen for incoming clients
  EthernetClient client = server.available();
  if (client) {
      while (client.connected()) {
        if (client.available()) {
          char thisChar = client.read();
          Serial.write(thisChar);
        }
      }
    // close the connection:
    client.stop();
  }
}
```

Run this sketch with the Ethernet module attached to your router and the Serial Monitor open. Then open a browser window and go to the Arduino's address. Using the example as shown, you'd go to http://192.168.1.20. You won't see anything in the browser, but you will see the HTTP request come through in the Serial Monitor. Now you're seeing what the server saw when you made HTTP requests in Chapter 3. A typical request will look like this:

```
GET / HTTP/1.1
Host: 192.168.1.1
Connection: keep-alive
Accept: application/xml,application/xhtml+xml,text/
html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X
10_6_5; en-US) AppleWebKit/534.10 (KHTML, like Gecko)
Chrome/8.0.552.215 Safari/534.10
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

You can ignore most of the HTTP parameters, but two pieces are useful: the first line, where you see what the client is requesting; and the end, where it sends a blank line to finish the request. In this case, the first line shows that the client is asking for the main index page at the root of the server, signified by the / in GET / . You're only going to have one result to return, but in the future, you will write a more complex server that looks at this part of the request and responds with different results depending on what's requested.

For now, you can keep it simple and just look for the end of the request, which will be a linefeed (\n or ASCII 10), followed by a carriage return (\r or ASCII 13), followed by another linefeed. Figure 4-5 shows what happens in the code that follows.

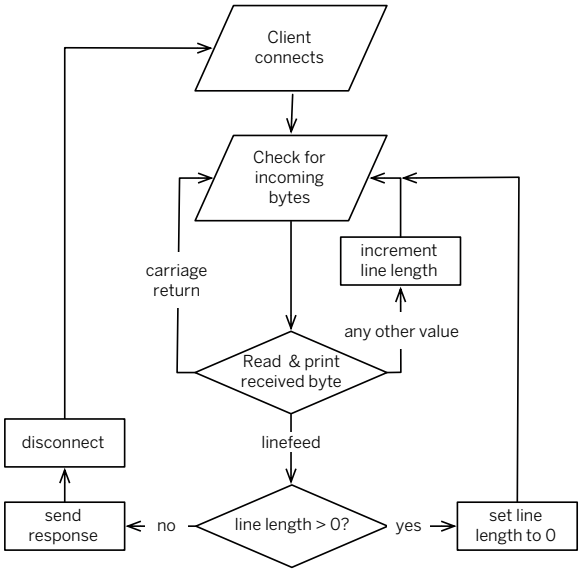


Figure 4-5
Logic flow for a simple server.

» Change your main loop as follows, adding the lines shown in blue after Serial.write(thisChar).

```
void loop()
{
  // listen for incoming clients
  EthernetClient client = server.available();
  if (client) {
    Serial.println("Got a client");
    int lineLength = 0; // length of the incoming text line

    while (client.connected()) {
      if (client.available()) {
        char thisChar = client.read();
        // if you get a linefeed and the request line is blank,
        // then the request is over:
        if (thisChar == '\n' && lineLength < 1) {
          // send a standard http response header
          makeResponse(client);
          break;
        }
        //if you get a newline or carriage return,
        // you're at the end of a line:
        if (thisChar == '\n' || thisChar == '\r') {
          lineLength = 0;
        }
        else {
          // for any other character, increment the line length:
          lineLength++;
        }
      }
    }
    // give the web browser time to receive the data
    delay(1);
    // close the connection:
    client.stop();
  }
}
```

» Before you can run this, you'll need to add a method , makeResponse(), at the end of your sketch to make a String to send the client. Here's a start.

```
void makeResponse(EthernetClient thisClient) {
  thisClient.print("HTTP/1.1 200 OK\n");
  thisClient.print("Content-Type: text/html\n\n");
  thisClient.print("Hello from Arduino</head><body>\n");
  thisClient.println("</body></html>\n");
}
```

When you enter the Arduino's address in your browser now, you'll get a web page. There's not much there, but it's legitimate enough that your browser doesn't know the difference between your Arduino and any other web server. You can add anything you want in the HTML in the makeResponse() method—even links to images and content on other servers. Think of the Arduino as a portal to any other content you want the user to see, whether it's physical data from sensors or other web-based data. The complexity is really up to you. Once you start thinking creatively with the capabilities of the print() and println() statements, you get a wide range of ways to dynamically generate a web interface to your Arduino through the Ethernet shield.

X

How about printing the states of the analog inputs? Add the following to makeResponse() (new lines are shown in blue).

```
void makeResponse(EthernetClient thisClient) {
  thisClient.print("HTTP/1.1 200 OK\n");
  thisClient.print("Content-Type: text/html\n\n");
  thisClient.print("<html><head>");
  thisClient.print("<title>Hello from Arduino</title></head><body>\n");

  // output the value of each analog input pin
  for (int analogChannel = 0; analogChannel < 6; analogChannel++) {
    thisClient.print("analog input ");
    thisClient.print(analogChannel);
    thisClient.print(" is ");
    thisClient.print(analogRead(analogChannel));
    thisClient.print("<br />\n");
  }
  thisClient.println("</body></html>\n");
}
```

Reload the page, and you've got the states of the analog inputs. But how about updating them continually? You could use the methods you used for the Cat Cam in Chapter 3. Change the line in makeResponse() that prints the HTML head, like the code shown at right.

```
  thisClient.print("Content-Type: text/html\n\n");
  thisClient.print(
    "<html><head><meta http-equiv=\"refresh\" content=\"3\">");
```

You can do all sorts of things in the response. Now it's time to take advantage of those three photocells you attached to the analog inputs. This version of the makeResponse() method prints out a page that changes its background color with the values from the three photocells.

Now that you've got a light color server, look for colorful places to put it.

```
  thisClient.print("Content-Type: text/html\n\n");
  thisClient.print(
    "<html><head><meta http-equiv=\"refresh\" content=\"3\">");
  thisClient.print("<title>Hello from Arduino</title></head>");
  // set up the body background color tag:
  thisClient.print("<body bgcolor=#");
  // read the three analog sensors:
  int red = analogRead(A0)/4;
  int green = analogRead(A1)/4;
  int blue = analogRead(A2)/4;
  // print them as one hexadecimal string:
  thisClient.print(red, HEX);
  thisClient.print(green, HEX);
  thisClient.print(blue, HEX);
  // close the tag:
  thisClient.print(">");
  // now print the color in the body of the HTML page:
  thisClient.print("The color of the light on the Arduino is #");
  thisClient.print(red, HEX);
  thisClient.print(green, HEX);
  thisClient.println(blue, HEX);
  // close the page:
  thisClient.println("</body></html>\n");
}
```

Making a Private IP Device Visible to the Internet

Up until now, the Internet-related projects in this book either worked only on a local subnet, or have only sent data outbound and waited for a reply. This is the first project in which your device needs to be visible to the Internet at large. You can view it while you're on the same local network, but if it's connected to your home router and has a private IP address, it won't be visible to anyone outside your home. To get around this, you need to arrange for one of your router's ports to forward incoming messages and connection requests to your Ethernet shield.

To do this, open your router's administrator interface and look for controls for "port forwarding" or "port mapping." The interface will vary depending on the make and model of your router, but the settings generally go by one of these names. It's easiest if the forwarded port on the router is the same as the open port on the Ethernet module, so configure it so that port 80 on your router connects to port 80 on the Ethernet shield (if your router allows it). Once you've done this, any incoming requests to connect to your router's public IP address on that port will be forwarded to the Ethernet module's private IP address on the port that you set. Note that the router reserves some ports for special purposes. For example, you may not be able to port forward port 80, because the router uses it for its own interface. That's why you might need to use a high number, like 8080.

Web browsers default to making their requests on port 80, but you can make a request on any port by adding the port number at the end of the server, address like so:

```
http://www.myserver.com:8080/
```

The new public address of your Ethernet module will follow this pattern, too. For example, if your home router's public address is 203.48.192.56, you'd access your new Arduino server at http://203.48.192.56:8080.

Figures 4-6 and 4-7 show the settings on an Apple AirPort Express router and a Linksys wireless router. On the Linksys router, you can find port forwarding under the Advanced tab.

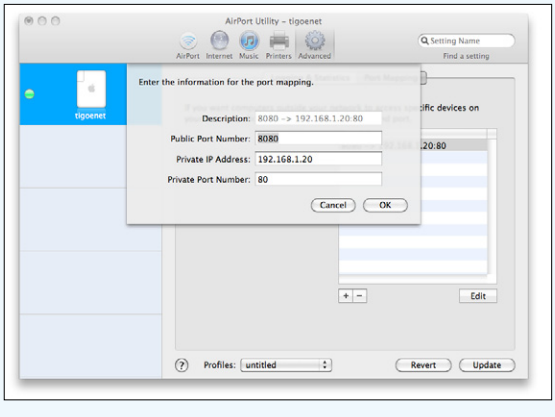


Figure 4-6
Port mapping tab on an Apple AirPort Express router. Port mapping can be found under the Advanced tab.

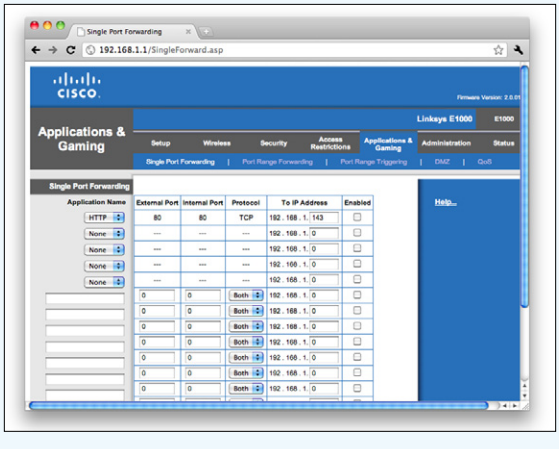


Figure 4-7
Port forwarding on a Linksys wireless router.

An Embedded Network Client Application

Now that you've made your first server, it's time to make a client. 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—but it's all yours.

Project 7

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 is less picturesque than mine, but it will do for a placeholder until you find one you love.

Figure 4-8 shows how it works: the microcontroller makes a network connection to a PHP script through the Ethernet shield. 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 can set a meter from your home or office to see the current air quality in your city (assuming you live in the U.S.).

MATERIALS

- **1 Arduino Ethernet or**
- **1 Arduino Ethernet shield and 1 Arduino microcontroller module**
- **1 Ethernet connection to the Internet**
- **1 solderless breadboard**
- **1 voltmeter**
- **4 LEDs**
- **4 220-ohm resistors**

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 per 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 4-9 shows the whole circuit for the project.

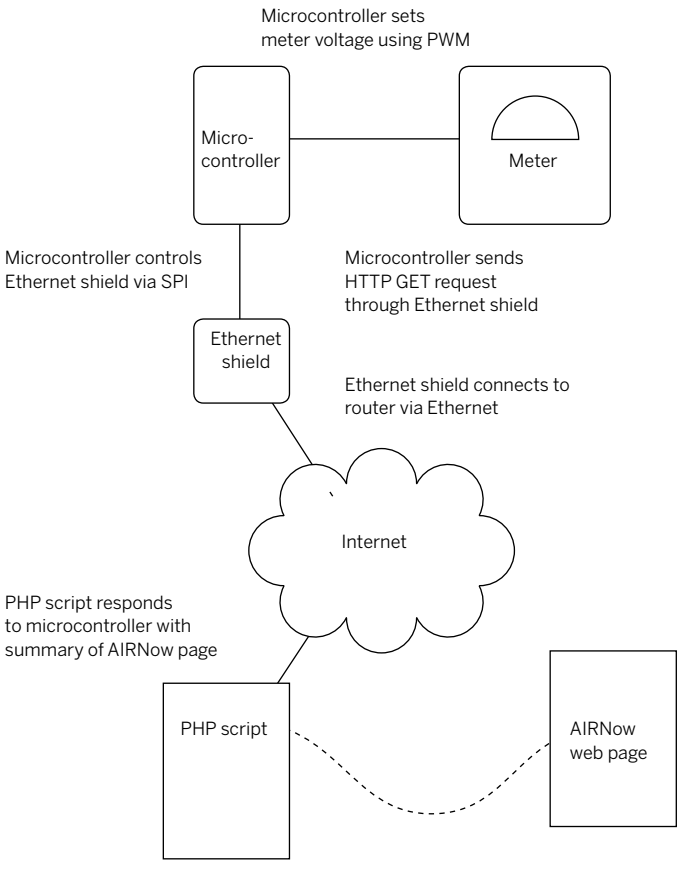
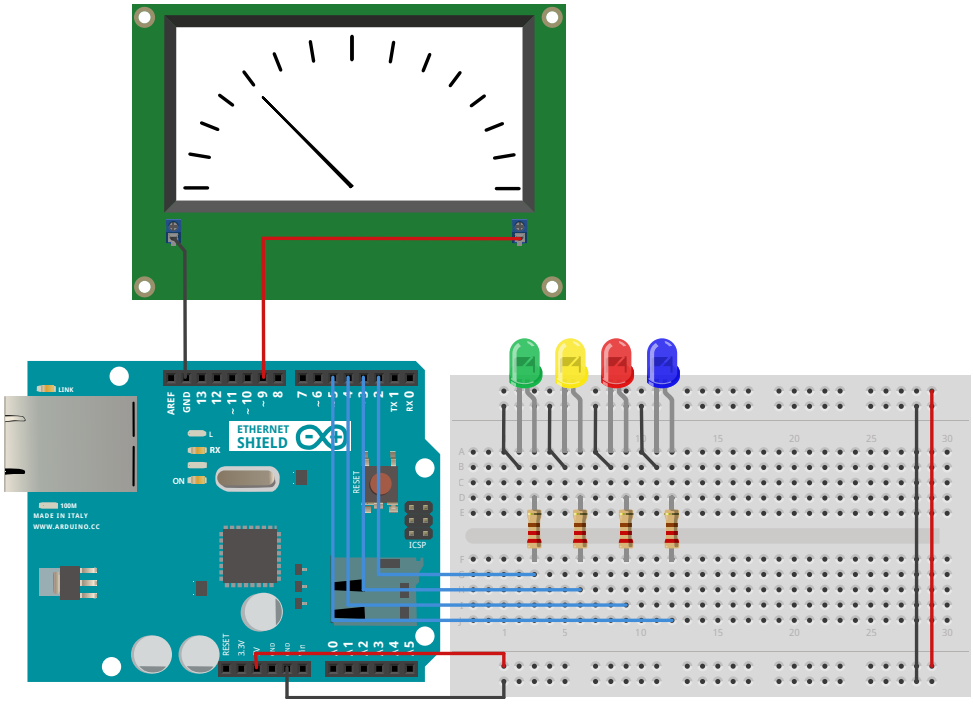


Figure 4-8
The networked air-quality meter.



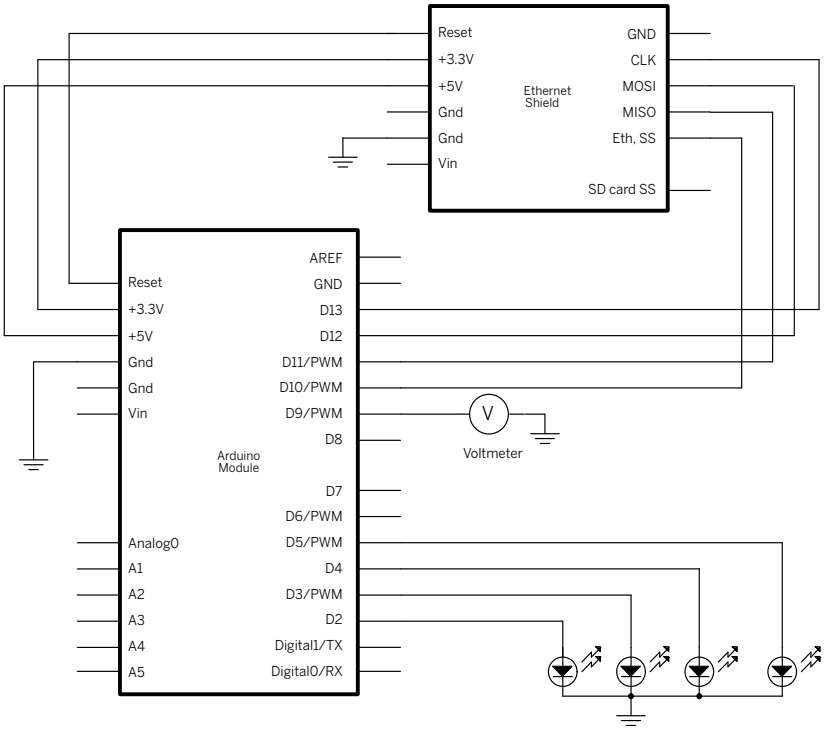


Figure 4-9
The circuit for a networked meter. The Ethernet controller shown in the schematic is on the shield or the Arduino Ethernet board.

Test It The program to the right tests whether you can control the meter.

You will need to adjust the range of pwmValue depending on your meter's sensitivity. The meters used to design this project had different ranges. The meter 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. Note your meter's minimum and maximum values. You'll use them later to scale the air-quality reading to the meter's range.

```
/*
    Voltmeter Tester
    Uses analogWrite() to control a voltmeter.
    Context: Arduino
*/
const int meterPin = 9;

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

void setup() {
  Serial.begin(9600);
}

void loop() {
  // move the meter from lowest to highest values:
  for (pwmValue = 0; pwmValue < 255; pwmValue ++) {
    analogWrite(meterPin, pwmValue);
    Serial.println(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. Using a microcontroller to parse through all the text of a web page is possible, but a bit complicated. However, it's the kind of task for which PHP was made. The program that follows reads the AIRNow page, extracts the current air-quality index (AQI) reading, and makes a simpler summary page that's easy to read with the microcontroller. The Ethernet controller is the microcontroller's gateway to the Internet, allowing it to open a TCP connection to your web host, where you will 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 4-10 shows AIRNow's page for New York City (http://airnow.gov/?action=airnow.local_city&zipcode=10003&submit=Go). AIRNow's page is formatted well for extracting the data. The AQI number is clearly shown in text, and if you remove all the HTML tags, it appears on a line by itself, always following the line Current Conditions.

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. In fact, it happened between the first and second editions of this book. 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

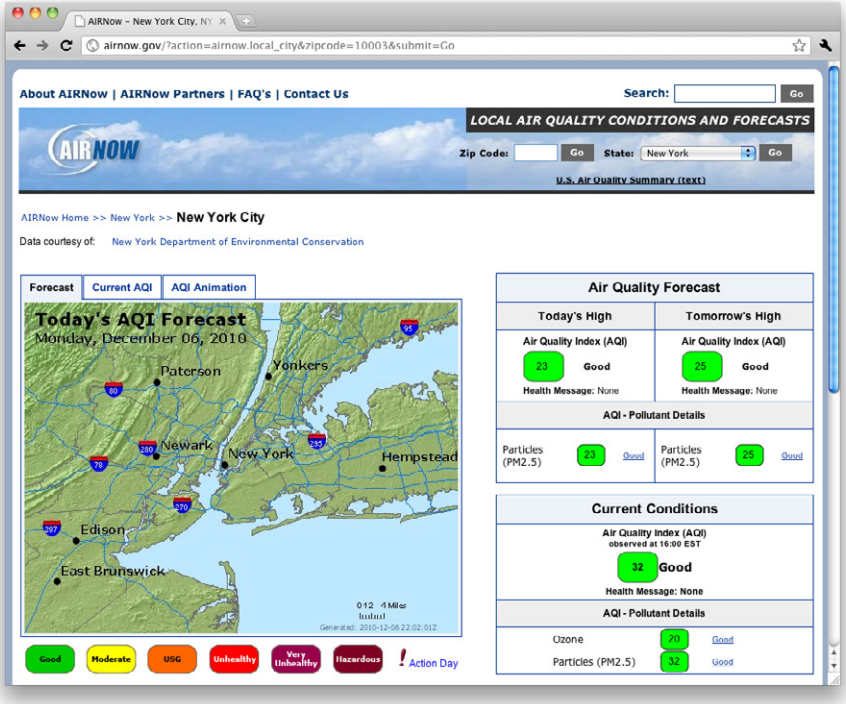


Figure 4-10
AIRNow's page is nicely laid out for scraping. The PHP program used in this project ignores the ozone level.

Fetch It

This PHP script opens the AIRNow web page and prints it line by 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 (use the View Source option in your web browser), it looks a bit better. Scroll down and you'll find some lines like this:

```
Current Conditions
Air Quality Index (AQI)
observed at 17:00 EST
45
```

These are the only lines you care about.

```
<?php
/*
    AIRNow Web Page Scraper
    Context: 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 this code before the fopen() command.

```
// whether you should check for a value
// on the line you're reading:
$checkForValue = false;

// value of the Air Quality reading:
$airQuality = -1;
```

» Replace the command echo $line; in the program with the block of code at right.

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 Current Conditions. When you see that line, you know the next line is the number you want. When the PHP script finds that line, it sets the variable $checkForValue to true.

```
// if the current line contains the substring "Current Conditions"
// then the next line with an integer is the air quality:
if (preg_match('/Current Conditions/', $line)) {
   $checkForValue = true;
}
```

» Now, add the following block of code after the one you just added. This code checks to see whether $checkForValue is true, and whether the line contains an integer and nothing else. If so, the program reads the next line of text and converts it from a string to an integer value. It will only get a valid integer when it reaches the line with the AQI value.

```
if ($checkForValue == true && (int)$line > 0){
   $airQuality = (int)$line;
   $checkForValue = false;
}
```

» Finally, add the following lines at the end of the script, after the while loop. This prints out the air-quality reading and closes the connection to the remote site.

```
echo "Air Quality:". $airQuality;
// close the file at the URL, you're done:
fclose($filePath);
```

The result in your web browser should look like this:

```
Air Quality: 43
```

Now you've got a short string of text that your microcontroller can read. Even if the script got no result, the value -1 will tell you that there was no connection.

Read the PHP Script Using the Microcontroller

Next, it's time to connect to the PHP script through the Net using the Ethernet module. This time, you'll use the shield as a client, not a server. Before you start programming, plan the sequence of messages. Using the Ethernet 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 also have to write a program to manage the exchange of messages. Whether you're writing that program in Processing, 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.

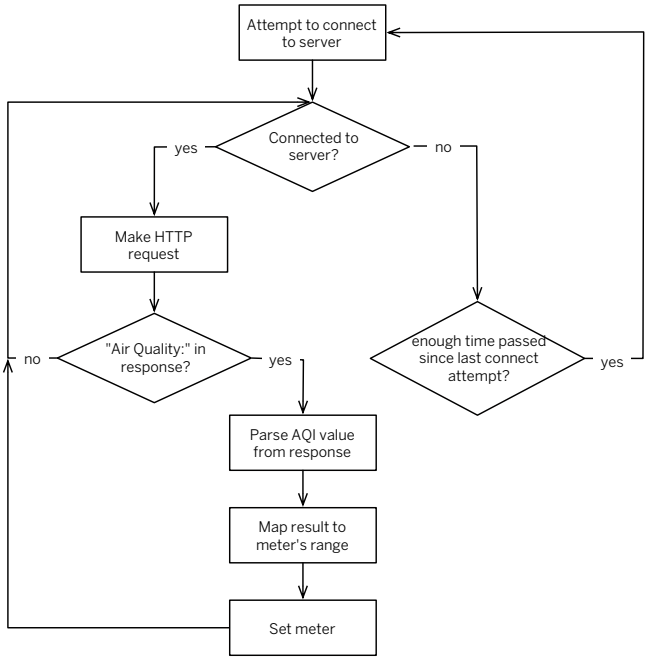


Figure 4-11
A flowchart of the Arduino program for making and processing an HTTP GET request.

Figure 4-11 is a flowchart of what happens in the microcontroller program. The major decisions (if statements in your code) are marked by diamonds; the methods are marked by rectangles. Laying out the whole program in a flowchart like this will help you keep track of what's going on at any given point. It also helps you to see what methods depend on a particular condition being true or not.

The circuit for this project also uses LEDs to keep track of the state of the program. LEDs attached to I/O pins will indicate the state. There's an LED to indicate that it's connected, another to indicate that it's disconnected, a third to indicate if it got a valid reading, and a fourth to indicate that the microcontroller is resetting.

This program will check the PHP script every two minutes. If there's a new value for the air quality, it'll read it and set the meter. If it can't get a connection, it will try again two minutes later. Because it's a client and not a server, there's no web interface to the project, only the meter.

TextFinder Library

For this sketch, you're going to need Michael Margolis' TextFinder library for Arduino. Download it from www.arduino.cc/playground/Code/TextFinder, unzip it, and save the **TextFinder** folder to the **libraries** folder of your Arduino sketches directory (the default location is **Documents/Arduino/libraries/** on OS X, **My Documents\Arduino\libraries** on Windows 7, and **~/Documents/Arduino/libraries/** on Ubuntu Linux. If the **libraries** directory doesn't exist, create

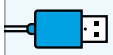
it and put TextFinder inside. Restart Arduino, and the TextFinder library should show up in the Sketch→Import Library menu. TextFinder lets you find a substring of text from the incoming stream of bytes. It's useful for both Ethernet and serial applications, as you'll see.

TextFinder was modified and included in version 1.0.1 of Arduino, after this edition was published. You can find its methods in the Stream class, from which Client, Server, and Serial inherit methods. For alternate versions of these sketches that use Stream instead of TextFinder, see the GitHub repository for this book's code, at https://github.com/tigoe/MakingThingsTalk2. Check the Reference section of www.arduino.cc for details of the Stream class.

Connect It The program starts out by including the SPI and Ethernet libraries, just as in the previous project. Then define the output pins, the minimum and maximum values for the meter that you determined earlier, and the time between HTTP requests using constant integers.

```
/*
  AirNow Web Scraper
 Context: Arduino
 */
#include <SPI.h>
#include <Ethernet.h>
#include <TextFinder.h>

const int connectedLED = 2;          // indicates a TCP connection
const int successLED = 3;            // indicates a successful read
const int resetLED = 4;              // indicates reset of Arduino
const int disconnectedLED = 5;       // indicates connection to server
const int meterPin = 9;              // controls VU meter
const int meterMin = 0;              // minimum level for the meter
const int meterMax = 200;            // maximum level for the meter
const int AQIMax = 200;              // maximum level for air quality
const int requestInterval = 10000;   // delay between updates to the server
```



Finding a Host's IP Address

This project uses numeric IP addresses rather than names. 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`**, (use** `-n` **instead of** `-c` **on Windows), you will get the following response, listing the IP address you need:**

```
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
```

If you can't use `ping` **(some service providers block it, since it can be used for nefarious purposes), you can also use** `nslookup`**. For example,** `nslookup google.com` **will return the following:**

```
Server:  8.8.8.8
Address: 8.8.8.8#53

Non-authoritative answer:
Name:        google.com
Address:     173.194.33.104
```

▸▸ nslookup returns the Domain Name Server it used to do the lookup as well.

Any one of the addresses listed will point to it's associated name, so you can use any of them.

Next, initialize a few array variables with the network configuration for the module.

```
byte mac[] = { 0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress ip(192,168,1,20);
IPAddress server(208,201,239,101 );
```

» Change these to match your own device and server.

The last global variables you need to add are for the EthernetClient class, and a few variables relating to the transaction with the server.

```
// Initialize the Ethernet client library
EthernetClient client;

boolean requested;           // whether you've made a request
long lastAttemptTime = 0;    // last time you connected to the server
int airQuality = 0;          // AQI value
boolean meterIsSet = false;  // whether the meter is set
```

setup() starts the serial and Ethernet connections, sets all the LED pins to be outputs, blinks the reset LED, and makes an initial attempt to connect to the server.

```
void setup() {
  // start the Ethernet connection:
  Ethernet.begin(mac, ip);
  // start the serial library:
  Serial.begin(9600);
  // set all status LED pins:
  pinMode(connectedLED, OUTPUT);
  pinMode(successLED, OUTPUT);
  pinMode(resetLED, OUTPUT);
  pinMode(disconnectedLED, OUTPUT);
  pinMode(meterPin, OUTPUT);

  // give the Ethernet shield a second to initialize:
  delay(1000);
  // blink the reset LED:
  blink(resetLED, 3);
  // attempt to connect:
  connectToServer();
}
```

The blink() method called in the setup blinks the reset LED so you know the microcontroller's main loop is about to begin.

```
void blink(int thisPin, int howManyTimes) {
  //     Blink the reset LED:
  for (int blinks=0; blinks< howManyTimes; blinks++) {
    digitalWrite(thisPin, HIGH);
    delay(200);
    digitalWrite(thisPin, LOW);
    delay(200);
  }
}
```

The loop() contains all the logic laid out in Figure 4-11. If the client is connected to the server, it makes an HTTP GET request. If it's made a request, it uses TextFinder to search the response for the air-quality string and then sets the meter. If the client isn't connected to the server, it waits another two minutes until the request interval's passed, and tries to connect again.

```
void loop()
{
  // if you're connected, save any incoming bytes
  // to the input string:
  if (client.connected()) {
    if (!requested) {
      requested = makeRequest();
    }
    else {
      // make an instance of TextFinder to search the response:
      TextFinder response(client);
      // see if the response from the server contains the AQI value:
      if(response.find("Air Quality:"))  {
        // convert the remaining part into an integer:
        airQuality = response.getValue();
        // set the meter:
        meterIsSet = setMeter(airQuality);
      }
    }
  }
  else if (millis() - lastAttemptTime > requestInterval) {
    // if you're not connected, and two minutes have passed since
    // your last connection, then attempt to connect again:
    client.stop();
    connectToServer();
  }

  // set the status LEDs:
  setLeds();
}
```

Both setup() and loop() attempt to connect to the server using the connectToServer() method. If it gets a connection, it resets the requested variable so the main loop knows it can make a request.

```
void connectToServer() {
  // clear the state of the meter:
  meterIsSet = false;

  // attempt to connect, and wait a millisecond:
  Serial.println("connecting...");
  if (client.connect(server, 80)) {
    requested = false;
  }
  // note the time of this connect attempt:
  lastAttemptTime = millis();
}
```

Once the client's connected, it sends an HTTP GET request. Here's the makeRequest() method. It returns a true value to set the requesting variable in the main loop, so the client doesn't request twice while it's connected.

```
boolean makeRequest() {
  // make HTTP GET request and fill in the path to
  // the PHP script on your server:
  client.println("GET /~myaccount/scraper.php HTTP/1.1\n");
  // fill in your server's name:
  client.print("HOST:example.com\n\n");
  // update the state of the program:
  client.println();
  return true;
}
```

» Change these to match your own server name and path.

The server replies to makeRequest() like so:

```
HTTP/1.1 200 OK
Date: Fri, 14 Nov 2010 21:31:37 GMT
Server: Apache/2.0.52 (Red Hat)
Content-Length: 10
Connection: close
Content-Type: text/html; charset=UTF-8

 Air Quality: 65
```

Once the client's found the air-quality string and converted the bytes that follow into an integer, it calls setMeter() to set the meter with the result it obtained. You might want to adjust the AQIMax value to reflect the typical maximum for your geographic area.

```
boolean setMeter(int thisLevel) {
  Serial.println("setting meter...");
  boolean result = false;
  // map the result to a range the meter can use:
  int meterSetting = map(thisLevel, 0, AQIMax, meterMin, meterMax);
  // set the meter:
  analogWrite(meterPin, meterSetting);
  if (meterSetting > 0) {
    result = true;
  }
  return result;
}
```

The last thing you need to do in the main loop is set the indicator LEDs so that you know where you are in the program. You can use the client.connected() status and the meterIsSet variable to set each of the LEDs.

```
void setLeds() {
  // connected LED and disconnected LED can just use
  // the client's connected() status:
  digitalWrite(connectedLED, client.connected());
  digitalWrite(disconnectedLED, !client.connected());
  // success LED depends on reading being successful:
  digitalWrite(successLED, meterIsSet);
}
```

That's the whole program. Once you've got this program working on the microcontroller, the controller will make the HTTP GET request once every ten minutes and set the meter accordingly.

The Finished Project

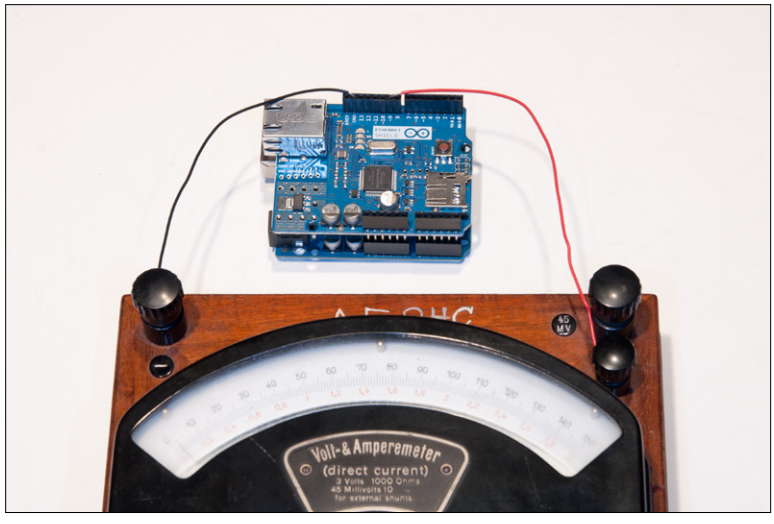


Figure 4-12
The completed networked air-quality meter.



DNS and DHCP

In Chapter 3 you learned that most Ethernet-connected devices obtain an IP address from their router using Dynamic Host Control Protocol (DHCP), and that IP-connected devices learn the name of other servers and clients using the Domain Name System (DNS). You can use DHCP and DNS with the Arduino Ethernet library as well. It's very simple—you just put the MAC address in the Ethernet.begin() **method, like so:**

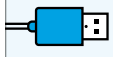
```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
Ethernet.begin(mac);
```

If the server grants an IP address, it returns true. If not, it returns false. The returned address is stored in Ethernet.localIP()**. You may want a few other variables to use DHCP as well, including** Ethernet.subnetMask()**,** Ethernet.gatewayIP()**, and** Ethernet.dnsServerIP()**.**

Using DNS is just as simple. Instead of passing a numeric server to the client.connect() **command, give it a text string, like so:**

```
client.connect("www.example.com", 80);
```

The disadvantage of using DHCP and DNS is that they take a lot of RAM, so you may prefer to use the relevant numerical addresses (once you know them).



DNS and DHCP (cont'd)

The following example shows how to request an address using DHCP; it thenreports back the address obtained:

```
/* DHCP
 Context: Arduino
 */
#include <SPI.h>
#include <Ethernet.h>

byte mac[] = {
  0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01};
IPAddress ip(192,168,1,20);             // an address to use if DHCP fails

void setup() {
  // start the serial library:
  Serial.begin(9600);
  // start the Ethernet connection:
  Serial.println("Asking for an IP address using DHCP...");
  if (!Ethernet.begin(mac)) {
    // if DHCP fails, set your own address:
    Ethernet.begin(mac, ip);
  }
  // print the bytes of the IP address, separated by dots:
  Serial.print("I got an IP address. It's ");
  Serial.println(Ethernet.localIP());
}

void loop() {
}
```

The begin() **technique is useful in any Ethernet sketch because it uses DHCP if available, and sets an IP address manually if not. If you know you're going to set the address manually and want to save program memory, just comment out the** if() **line and the corresponding bracket, and leave the command inside it, like so:**

```
//if (!Ethernet.begin(mac)) {
   // if DHCP fails, set your own address:
   Ethernet.begin(mac, ip);
// }
```

Programming and Troubleshooting Tools for Embedded Modules

You may have hit a number of problems when making the connections in the last section. Probably the most challenging thing about troubleshooting them was that neither the Ethernet shield nor the microcontroller gave any indication that a problem occurred. This is the norm when you're working with embedded modules 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 Ethernet modules or some other network or communications module. You'll use these methods over and over again in the rest of the book and beyond.

The Three Most Common Mistakes

Check Power and Ground

Always check whether you have made the power and ground connections correctly. This is less of an issue when you've got a nice plug-in module like the Ethernet shield that can connect only one way. But even with plug-in modules, you can get a pin misaligned and not notice. 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.

Check the Connections

When you're wiring a module to a microcontroller by hand, it's fairly common to get the wires wrong the first time. Make sure you know what each pin does, and double-check that the pin that one transmits on connects to the pin that the other receives on, and vice versa. If it's a synchronous serial connection, make sure the clock is connected.

Check the Configuration

If you're certain about the hardware connections, check the device's configuration to make sure it's all correct. Did you get the IP address correct? Is the router address correct? Is the netmask?

Diagnostic Tools and Methods

Once you know the device is 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 as desired.

It's a good idea to test your module first using sample code—if it's provided. Every good communications module maker gives you a starting example. Use it, and keep it handy. When things go wrong, return to it to make sure that basic communications still work.

Physical Debugging Methods

Writing code makes things happen physically. It's easy to forget that when you're working on an exchange of information like the examples shown here. So, it's helpful to put in physical actions that you can trigger; this way, you can see that your code is working. In the last project, you saw LEDs turn on when the client connected or disconnected, when it made a successful request, and when it reset. Triggering an LED is the most basic and reliable thing you can do from a microcontroller—even more basic than sending a serial debugging message—so use LEDs liberally to help make sure each part of your code is working. You can always remove them later.

Serial Debugging Methods

Besides their physical form factor, the Ethernet shield and Arduino Ethernet make a number of things easy for you. For example, the fact that they use synchronous serial

communication to talk to the Ethernet controller means you can still use the serial port to get messages about what's going on—as you saw in the previous example. Following are a few tips for effective serial debugging of networking code using serial messages.

It's important to keep in mind that serial communication takes time. When you have serial statements in your code, they slow it down. The microcontroller has to save each bit of information received, which takes time away from your program. Even though the time is miniscule for each bit, it can add up. After a few hundred bytes, you might start to notice it. So, make sure to take out your serial transmission statements when you're done with them.

Debug It

One way to manage your debugging is to have a variable that changes the behavior of your program, like this. When every debugging statement is preceded by the if (DEBUG) conditional, you can easily turn them all off by setting DEBUG to false.

```
const boolean DEBUG = true;

void setup() {
  Serial.begin(9600);
}

void loop() {
  if (DEBUG) Serial.println("this is a debugging statement");
}
```

Announce It

In the Air-Quality Index client project, you had a lot of different methods in your code. It's not easy to tell whether every method is being called, so during troubleshooting, make it a habit to "announce" serially when every method is being called. To do so, put a serial print statement at the beginning of the method, like the code shown on the right.

```
void connectToServer() {
  if (DEBUG) Serial.print("running connectToServer()...");
  // rest of the method goes here
}

void makeRequest() {
  if (DEBUG) Serial.print("running makeRequest()...");
  // rest of the method goes here
}

boolean setMeter(int thisLevel) {
  if (DEBUG) Serial.print("running setMeter()...");
  // rest of the method goes here
 }
```

Check Conditionals

It's common in a communications application like these to have multiple conditions that affect a particular result. As you're working, you can easily make mistakes that affect these nested if statements. As you work, check that they're still coming true when you think they are by announcing it, as shown here.

```
  if (client.connected()) {
    if (DEBUG) Serial.print("connected");
    if (!requested) {
      if (DEBUG) Serial.print("connected, not requested.");
      requested = makeRequest();
    }
    else {
       if (DEBUG) Serial.print("not connected");
    }
  }
```

Separate It Programmers like efficiency, so they will often combine several commands into one line of code. Sometimes the problem is in the combination. For example, this line attempts to connect using client.connect(), and then it checks the result in a conditional statement. It gave me all kinds of trouble until I separated the connection attempt from the check of the connection.

```
if (client.connect()) {
  // if you get a connection, report back via serial:
  Serial.println("connected");
}
else {
  // if you didn't get a connection to the server:
  Serial.println("connection failed");
}
```

» Here's the less efficient code that solved the problem. Checking the connection immediately after connecting was apparently too soon. The delay stabilized the whole program.

```
client.connect();   // connect
delay(1);           // wait a millisecond
if (client.connected()) {
  // if you get a connection, report back via serial:
  Serial.println("connected");
}
else {
  // if you didn't get a connection to the server:
Serial.println("connection failed");
}
```

Just Watch It Sometimes it helps to step back from your complicated program and just watch what you're receiving from the other end. One technique I returned to again and again while developing this program was to simply print out what the Ethernet module was receiving. Several times it revealed the problem to me. Sometimes it was because I had logic problems in my code, and other times it was because I'd neglected some character that the server was sending as part of the protocol. Familiarity can blind you to what you need to see. So, remind yourself of what you are actually receiving, as opposed to what you think you are receiving.

```
// if you're connected, save any incoming bytes
// to the input string:
if (client.connected()) {
  if (client.available()) {
    char inChar = client.read();
    Serial.write(inChar);
  }
}
```

Write a Test Client Program

It's easiest to work through the steps of the program if you can step through the sequence of events. More complex development environments allow you to step through a program one line at a time. Arduino doesn't give you that ability, but it can link to other environments. The following program simply passes anything that comes in the serial port to the Ethernet port, and vice versa. It turns your Arduino into a serial-to-Ethernet gateway. Using this code, you can connect to a serial terminal or to Processing—or to any other development environment that can communicate serially—to test the Ethernet connection.

Test It The handy thing about this program is that you can test the exchange of messages manually, or by writing a program in a desktop environment. Once you know you have the sequence right, you can translate it into code for the Arduino module.

```
/*
  Serial To Ethernet
  Context: Arduino
 */

#include <SPI.h>
#include <Ethernet.h>

byte mac[] = { 0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress ip(192,168,1,20);
IPAddress server(208,201,239,101 );

// Initialize the Ethernet client library
// with the IP address and port of the server
// that you want to connect to (port 80 is default for HTTP):
EthernetClient client;

void setup() {
  // start the serial library:
  Serial.begin(9600);
  // start the Ethernet connection:
  if (!Ethernet.begin(mac)) {
    Serial.println("DHCP failed, configuring manually.");
    // configure manually with your own IP address:
    Ethernet.begin(mac, ip);
  }
  // give the Ethernet shield a second to initialize:
  delay(1000);
  Serial.println("Ready to go.");
}

void loop() {
  if (client.connected()) {
    //if you're connected, pass bytes from client to serial:
    if (client.available()) {
      char netChar = client.read();
      Serial.write(netChar);
    }
    //pass bytes from serial to client:
    if (Serial.available()) {
      char serialChar = Serial.read();
```

» Change these to match your own device and server.

»

Continued from previous page.

```
      client.write(serialChar);
    }
  }
  else {
    // in case you were connected, stop the client:
    client.stop();
    // if you're not connected, and you get a serial C,
    // attempt to connect:
    if (Serial.available()) {
      char serialChar = Serial.read();
      if (serialChar == 'C') {
        connectToServer();
      }
    }
  }
}

void connectToServer() {
  // attempt to connect, and wait a millisecond:
  Serial.println("connecting...");
  if (client.connect(server, 80)) {
    // if you get a connection, report back via serial:
    Serial.println("connected");

  }
  else {
    // if you didn't get a connection to the server:
    Serial.println("connection failed");
    client.stop();
  }
}
```

Talk To It Here's a simple Processing sketch that will communicate serially with the preceding Arduino sketch to make an HTTP request. When you type a key the first time, it will send C to initiate a connection. The second keystroke will send the HTTP request.

```
/*
 Serial-to-ethernet HTTP request tester
 Context: Processing

 */
// 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
```



Continued from previous page.

```
void setup()
{
  // get the list of serial ports:
  println(Serial.list());
  // open the serial port appropriate to your computer:
  myPort = new Serial(this, Serial.list()[0], 9600);
  // configure the serial object to buffer text until it receives a
  // linefeed character:
  myPort.bufferUntil(linefeed);
}

void draw()
{
  //no action in the draw loop
}

void serialEvent(Serial myPort) {
  // print any string that comes in serially to the monitor pane
  print(myPort.readString());
}

void keyReleased() {
  // if any key is pressed, take the next step:
  switch (step) {
  case 0:
    // open a connection to the server in question:
    myPort.write("C");
    // add one to step so that the next keystroke causes the next step:
    step++;
    break;
  case 1:
    // send an HTTP GET request
    myPort.write("GET /~myaccount/index.html HTTP/1.1\n");
    myPort.write("HOST:myserver.com\n\n");
    step++;
    break;
  }
}
```

» Change these to match your own server.

Starting with this sketch as a base, you could work out the whole logic of the HTTP exchange in Processing, then convert your sketch into an Arduino sketch.

This approach allows you to work out the logical flow of an application in an environment with which you might be more familiar. It removes the complications of the microcontroller environment, allowing you to concentrate on the sequence of messages and the actions that trigger them.

Don't get too caught up in the technical details of your program when working this way. You're still going to have to deal with the details of the microcontroller program eventually. However, it often helps to see the similarities and differences between two languages when you're working on a problem. It helps you concentrate on the logic that underlies them.

X

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, write your own server program to which it can connect.

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 an HTTP response and a simple web page.

To use this, first make sure your Ethernet module and your PC are on the same network. Then run this program, and connect to it from a browser, using the URL http://localhost:80.

When you know that works, write a client program for the Arduino that connects to your PC's IP address, port 80. You'll be able to see both sides of the communication, and determine where the problem lies. 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 of their server diagnostic tools, especially any error logs for your server.

X

If this doesn't work for you, change to 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 inthis program.

```
/*
  Test Server Program
 Context: 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 = 80;        // the port the server listens on
Server myServer;        // the server object
int counter = 0;
void setup()
{
  myServer = new Server(this, port); // Start the server
}

void draw()
{
  // get the next client that sends a message:
  Client thisClient = myServer.available();
  // if the message is not null, display what it sent:

  if (thisClient != null) {
    // read bytes incoming from the client:
    while(thisClient.available() > 0) {
      print(char(thisClient.read()));
    }
    // send an HTTP response:
    thisClient.write("HTTP/1.1 200 OK\r\n");
    thisClient.write("Content-Type: text/html\r\n\r\n");
    thisClient.write("<html><head><title>Hello</title></head>");
    thisClient.write("<body>Hello, Client! " + counter);
    thisClient.write("</body></html>\r\n\r\n");
    // disconnect:
    thisClient.stop();
    counter++;
  }
}
```

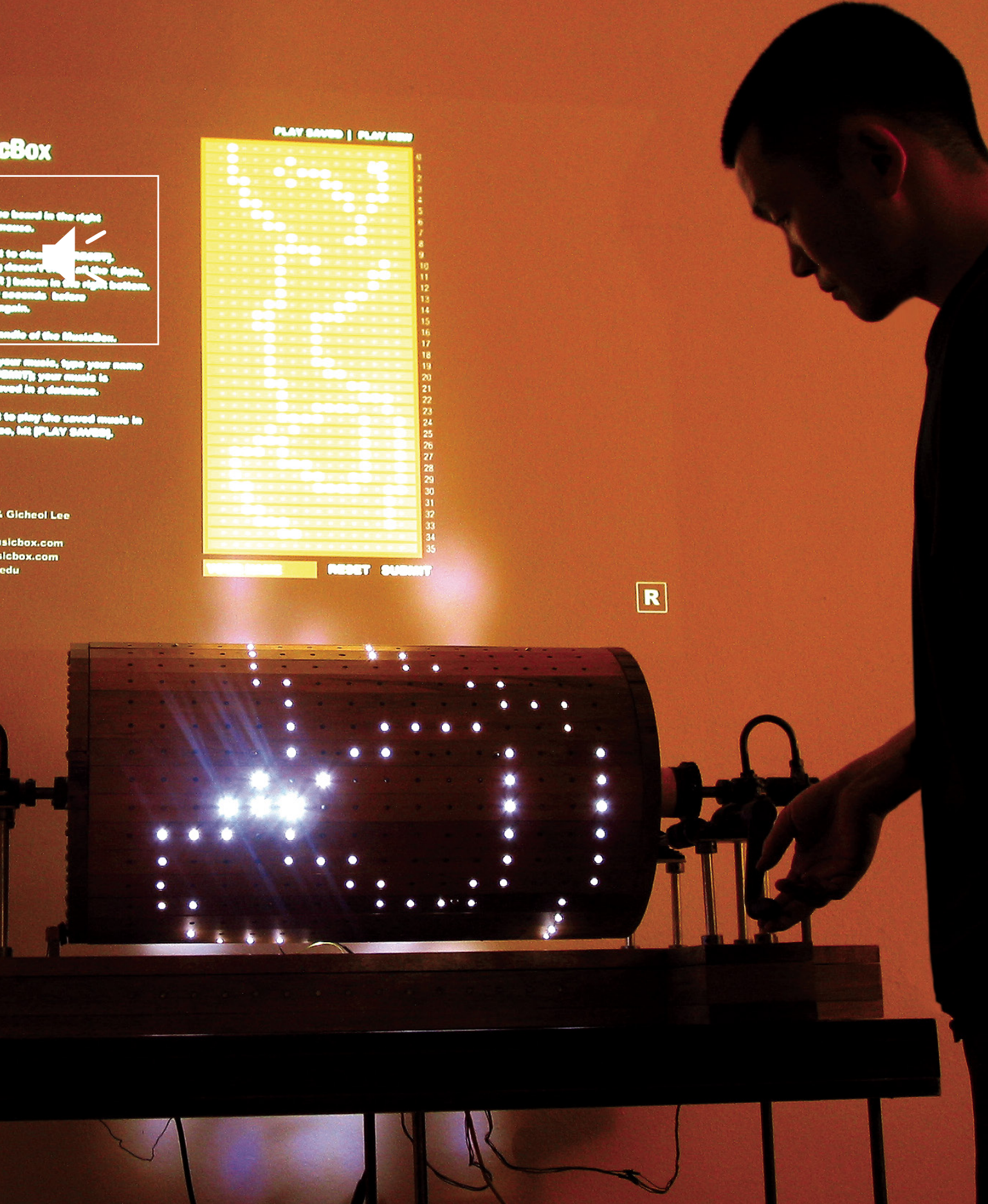
Conclusion

The activities in this chapter show a model for networked objects that's very flexible and useful. The object is basically a browser or a server, requesting information from the Web and extracting the information it needs, or delivering information to a client. You can use these models in many different projects.

The advantage of these models is that they don'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 earlier in 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 Chapter 5, 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. This is because they let you see what should happen, and then modify what actually is happening to match that.

X


cBox
PLAY SAVED | PLAY NEW
RESET
SUBMIT
R

5

MAKE: PROJECTS 

Communicating in (Near) Real Time

So far, most of the networked communications you've seen worked through a web browser. Your object made a request to a remote server, the server ran a program, and then it sent a response. This transaction worked by making a connection to the web server, exchanging some information, and 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 gives the player feedback. *Photo courtesy of Jin-Yo Mok.*

Supplies for Chapter 5

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **P** Pololu (www.pololu.com)
- **RS** RS (www.rs-online.com)
- **SF** SparkFun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 8: Networked Pong

NOTE: You'll see two devices you can build in this chapter. If you plan to build both, double the quantities on all parts except the joystick, the accelerometer, and the triple-wall cardboard.

» **1 Arduino Ethernet board** **A** A000050 Alternatively, an Uno-compatible board (see Chapter 2) with an Ethernet shield will work. **SF** DEV-09026, **J** 2124242, **A** A000056, **AF** 201, **F** 1848680

» **1 Ethernet connection to the Internet** Your home router most likely has Ethernet jacks in the back. If you've hooked up your computer to the Internet using Ethernet, you know where the ports are.

» **1 100-ohm resistor** **D** 100QBK-ND, **J** 690620, **F** 9337660, **RS** 707-8625

» **3 220-ohm resistors** **D** 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842

» **1 2-axis joystick (for joystick client)** **J** 2082855, **SF** COM-09032, **AF** 245, **F** 1428461

» **1accelerometer (for tilt board client)** The circuit shown uses an ADXL330 accelerometer, but most any analog accelerometer should do the job. **J** 28017, **SF** SEN-00692, **AF** 163, **RS** 726-3738, **P** 1247, **MS** MKPX7

» **1 perforated printed circuit board** **D** V2018-ND, **J** 616673, **SS** STR125C2B, **F** 4903213, **RS** 159-5420

» **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. **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590

» **1 push button** Use one that's robust and can stand a good stomping.

» **1 project enclosure** Get creative with materials you have laying around the house.

» **1 sheet triple-wall cardboard (for tilt board client)**

» **1 tab of Velcro** Get any Velcro from your closest fabric or hardware store.



Figure 5-1. New parts for this chapter: **1.** Triple-wall cardboard **2.** 3-axis accelerometer **3.** 2-axis joystick **4.** Pushbutton **5.** Velcro **6.** Project enclosure, made from mat board **7.** 1/16" mat board template for project enclosure **8.** Perforated circuit board. Don't forget plenty of male header pins for the breakout boards.

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 longer for the paddles to move when you move Monski's arms, it would 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 distributed equally, 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 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 previous two chapters. The pipe is opened when the server acknowledges the client's contact, and it remains open until the server has finished sending the file. If there are multiple files needed for a web page, such as images and style sheets, then multiple socket connections are opened and closed.

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: network hardware calculates the optimal route is differently, which 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 resent. 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 Arduino Ethernet boards 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: User Datagram Protocol (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 8

Networked Pong

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 *ping* pong. The server will be a Processing program, and the clients will be physical interfaces that connect through Ethernet-enabled Arduinos. The clients and the server's screen have to be physically close 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 aware of 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
```

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 Chat 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), so the following is a simple server with all the basic elements to handle network communications. It will let you listen for new clients, and then 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, 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. However, you'll have to press Return after each character in order for the server to see it—unless you make a change after you connect.

On Mac OS X or Linux, press the telnet escape key combination (Ctrl-]), type the following, and then press Return:

```
mode character
```

On Windows, telnet should not require any special configuration, but if you find otherwise, press Ctrl-], type the following, and press Return twice:

```
set mode stream
```

Now every character you type will be sent to the server as soon as you type it.

X

Try It First, start with the variable declarations and definitions. The main variables are an instance of the Server class, a port number to serve on, and an ArrayList to keep track of the clients.

```
/*
  Test Server Program
 Context: 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
ArrayList clients = new ArrayList(); // list of clients
```

» This program uses a data type you may not have seen before: ArrayList. 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. There is an introduction to ArrayLists on the Processing website at www.processing.org.

» The setup() method starts the server.

```
void setup()
{
  myServer = new Server(this, port);
}
```

» The draw() method listens for new messages from clients and prints them. If a client says "exit," the server disconnects it and removes it from the list of clients.

```
void draw()
{
  // get the next client that sends a message:
  Client speakingClient = myServer.available();

  if (speakingClient !=null) {
    String message = trim(speakingClient.readString());
    // print who sent the message, and what they sent:
    println(speakingClient.ip() + "\t" + message);

    if (message.equals("exit")) {
      myServer.disconnect(speakingClient);
      clients.remove(speakingClient);
    }
  }
}
```

» The serverEvent() message is generated by the server when a new client connects to it. serverEvent() announces new clients and adds them to the client list.

```
void serverEvent(Server myServer, Client thisClient) {
  println("We have a new client: " + thisClient.ip());
  clients.add(thisClient);
}
```

» Finally, the keyReleased() method sends any keystrokes typed on the server to all connected clients.

```
void keyReleased() {
    myServer.write(key);
}
```

The Clients

The pong client listens to local input and remote input. The local input is from you, the user. The remote input is from the server. The client is constantly listening to you, but it only listens to the server when it's connected.

To listen to you, the client needs:

- 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:

- 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 to give local feedback when it receives input from the user. Even if there is no connection to the server, local feedback lets the user know that the client device is responding to her actions. 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 determine whether the problem is with the connection to the server, or with the client device itself.

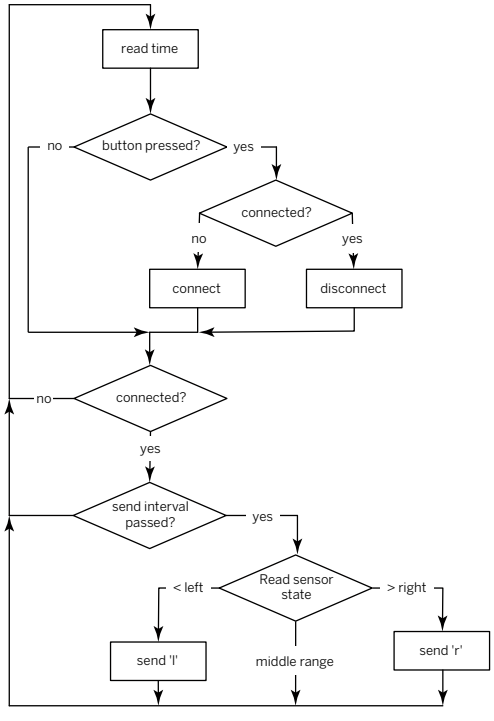


Figure 5-2
Logic flowchart for the pong clients

Use outputs to indicate the client's status.In the following code, you can see 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 states, you'd need more status indicators.

For this project, I built two clients. They have different methods of physical interaction and different input sensors, but they 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 them 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. Both of the clients use logic shown in Figure 5-2.

Client #1: A Joystick Client

The first client is a fairly traditional game controller, shown in Figure 5-3. It's got a joystick to control the left/right action, and a pushbutton to log in or out of the server. There are also four LEDs to indicate the controller's state. When it's logged in, the white LED is lit. When it's sending a left command, the red LED is lit. When it's sending a right command, the green LED is lit. When you're pressing the connect/disconnect button, the yellow LED lights up.

This client uses one axis of a two-axis joystick as its main input. The joystick contains two potentiometers: one for up/down movement and the other for left/right movement. You'll only need the output from the latter. Connect the pin marked L/R+ to +5V, the one marked GND to ground, and the one marked L/R to analog input 0 of your Arduino (the full circuit is shown in Figures 5-4 through 5-8). Then write a short program to read its output using analogRead(), just as you did with the Monski Pong sensors in Chapter 2 and the cat sensors in Chapter 3. Once you've got it working, note the values that indicate left and right, and save them for use in the client program.

You'll notice that the connection switch is wired differently from a normal switch. Instead of a 10-kilohm pulldown resistor, it's got a 100-ohm resistor and an LED connecting it to ground. This way, when you push the button, the LED lights up. There's no need to add any code to make this happen—the pushbutton will send current through the LED automatically, and you'll get local feedback on when the button is being pressed.

These two clients are laid out on printed circuit prototyping boards (commonly known as perf boards). The perfboards make it possible for the circuits to be a little smaller and more physically robust. They're a bit harder to assemble than a solderless breadboard circuit, though.

MATERIALS FOR BOTH CLIENTS

- **1 Arduino Ethernet board or Arduino board with the Ethernet shield**
- **1 Ethernet connection to the Internet**
- **1 100-ohm resistor**
- **3 220-ohm resistors**
- **1 2-axis joystick (for joystick client)**
- **1 accelerometer (for tilt board client)**
- **1 perforated printed circuit board**
- **4 LEDs**
- **1 pushbutton**
- **1 project enclosure**
- **1 sheet extra-thick cardboard (for tilt board client)**

Both the perfboard and breadboard layouts are shown so you can choose whichever method you like.

You'll need an enclosure for the circuit. You can buy a project case from your favorite electronics retailer, but you can also get creative. For example, a pencil box from your friendly neighborhood stationery store will work well. Drill holes in the lid for the switch and the LEDs, cut a hole for the joystick, cut holes in the side for the Arduino and Ethernet jacks, and you're all set. If you're skilled with a mat knife, you can also make your own box from cardboard or mat board. Figure 5-7 shows the box I made, and Figure 5-9 shows the template from which I made it. If you're lucky enough to have access to a laser cutter, use it; if not, a mat knife and a steady hand can do the job.

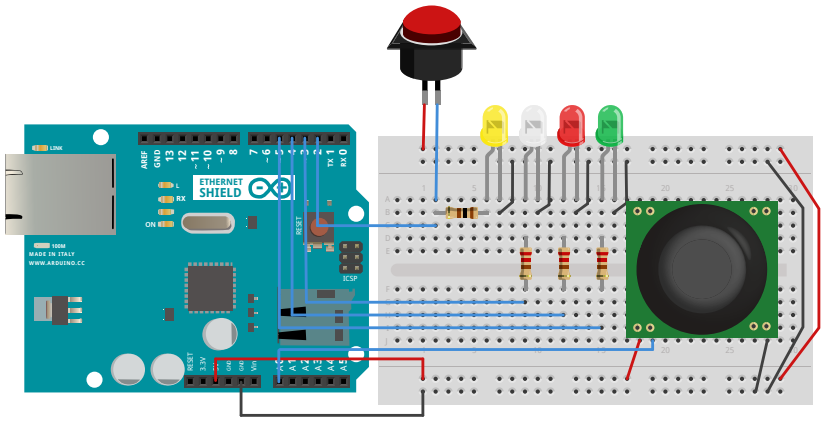
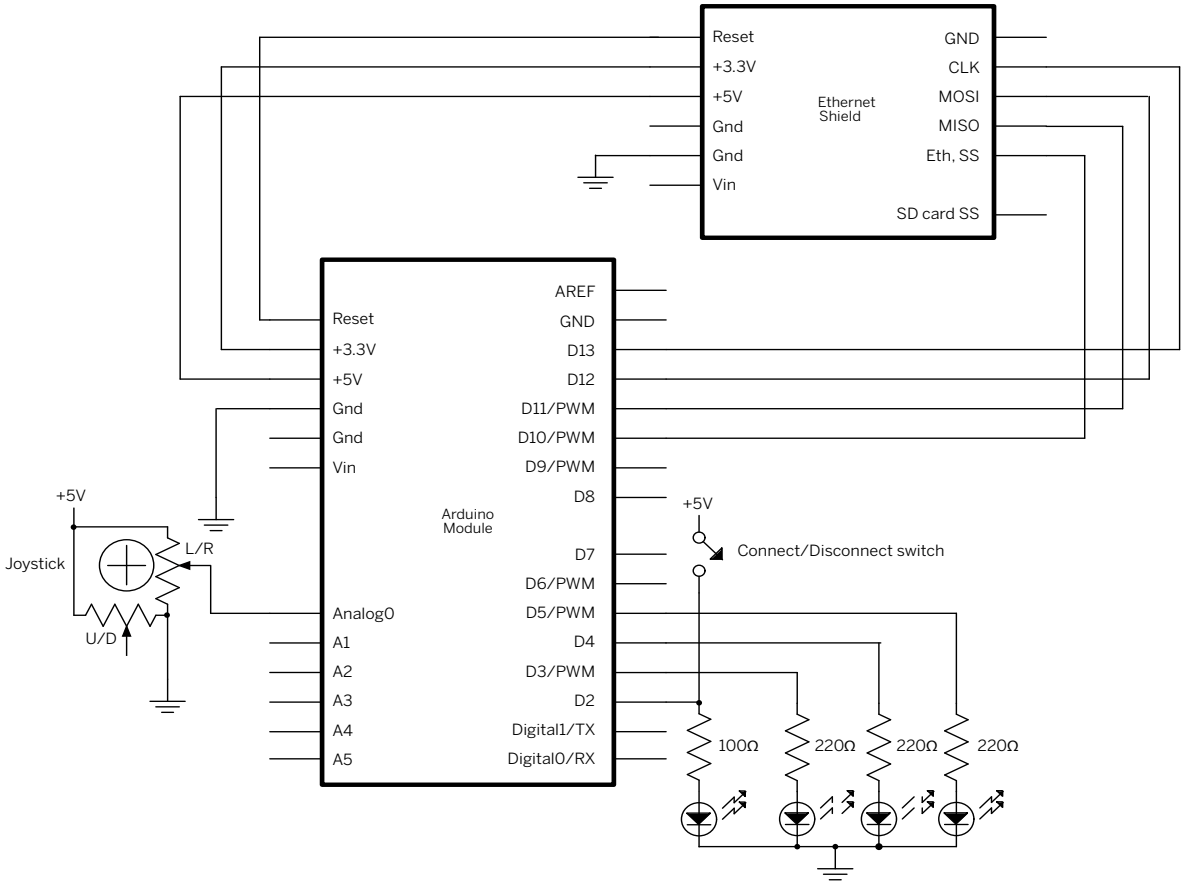


Figure 5-3
The joystick client, breadboard layout. Note the red and green LEDs to indicate left and right tilts. This client follows nautical tradition: port (left) is red and starboard (right) is green. Feel free to be less jaunty in your own choices, as long as they're clear.

Figure 5-4
The joystick client schematic. The following page shows the layout for the circuit on a perforated circuit board.



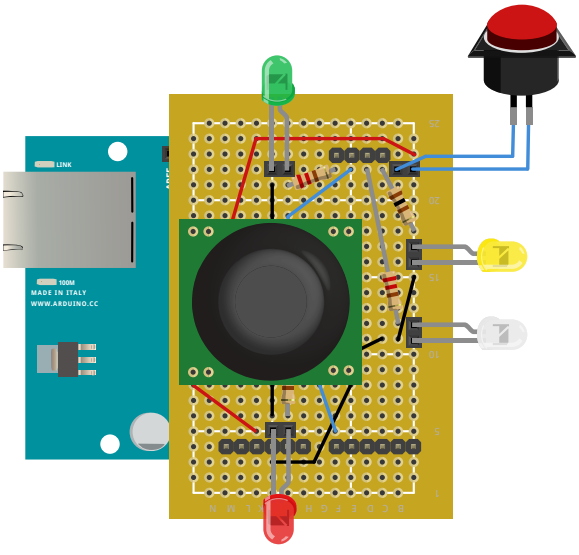
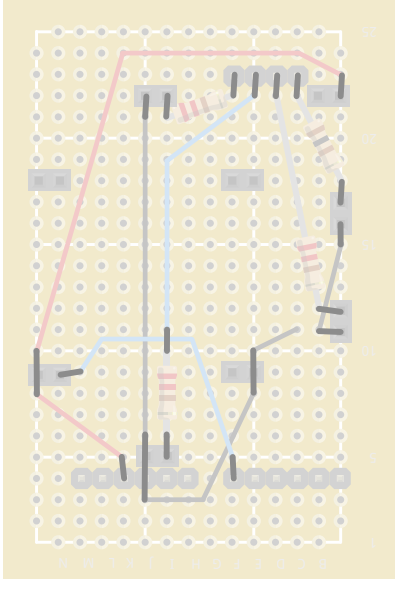


Figure 5-5
The circuit as laid out on a perforated circuit board (perfboard). Note that male headers are only attached to the Arduino's analog input pins, power and ground pins, and digital pins 2 through 5. The rest aren't needed.

Figure 5-6
The circuit board layout showing where the solder joints are connected underneath the board.



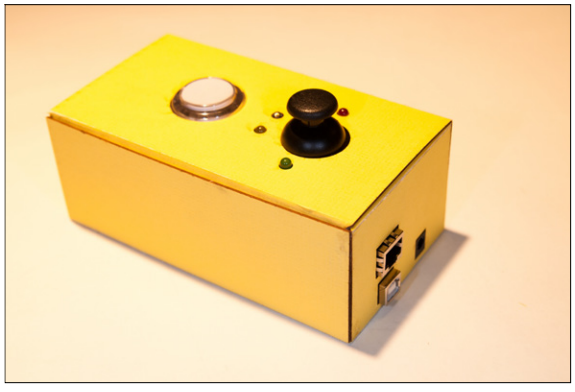
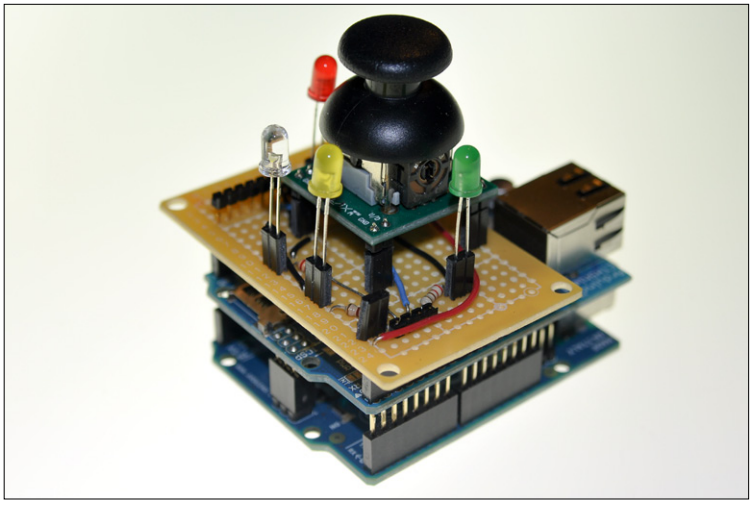


Figure 5-7
The assembled joystick controller. The joystick circuit inside the housing was designed to be just tall enough so that the LEDs and joystick would stick through the top. Assemble the circuit first so you know how much space you need for the housing. A cable made from scrap stranded wire and male headers to connect the pushbutton makes it easier to open and close the box with the button mounted. The inner wires from a telephone cable work nicely for this.

Figure 5-8
Detail of the Arduino, Ethernet shield, and circuit board. Use female headers to mount the LEDs and the switch. For the LEDs, it makes the job of getting them to the right height much easier. You can trim them bit by bit until they are the right height, then stick them in the headers.



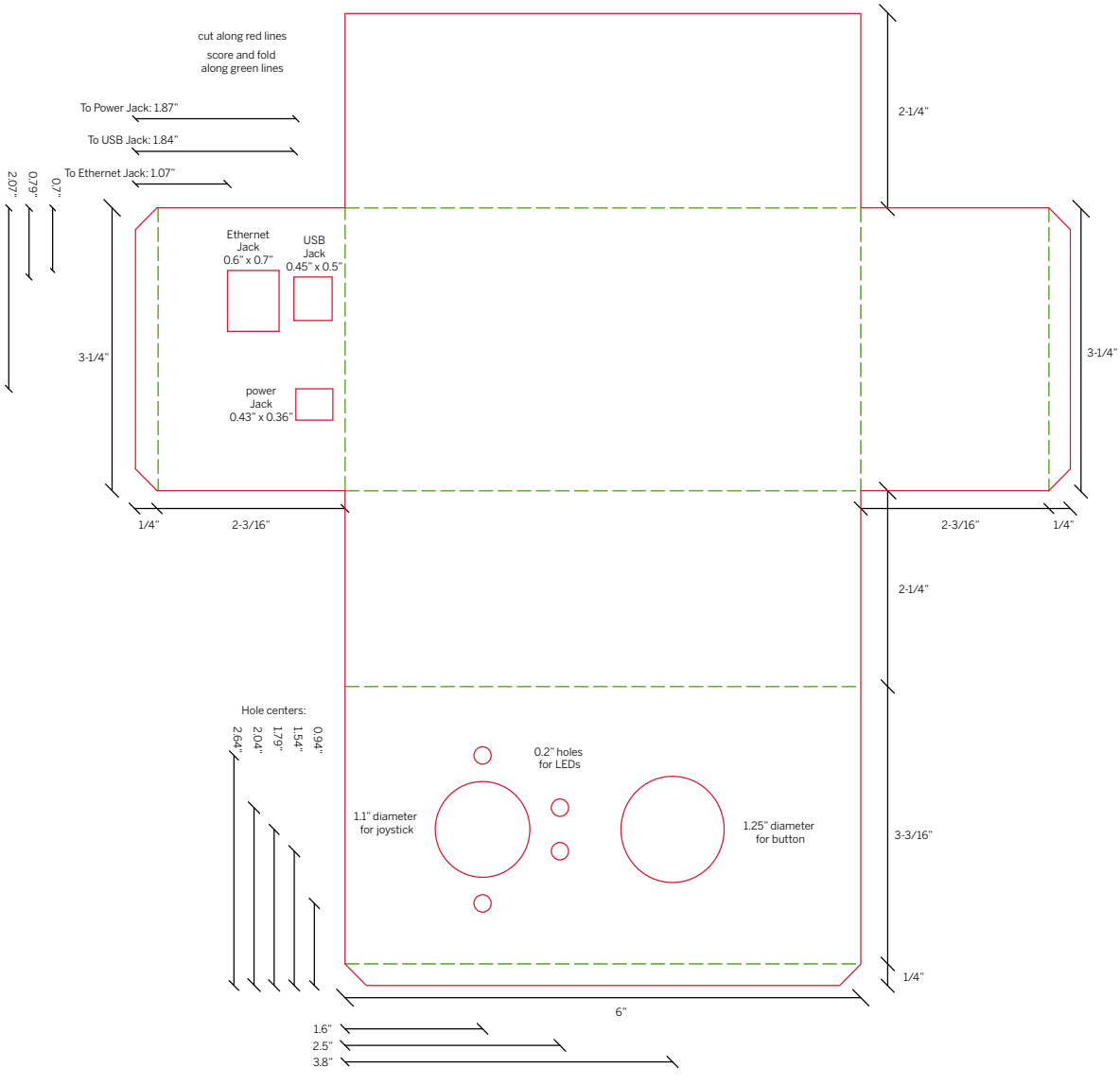


Figure 5-9

Template for the joystick housing. This template can be cut out of poster board or mat board and then folded to make a housing for the joystick controller. The dimensions will depend on how you assemble your circuit, so modify them as needed. Although they look precise here, they were measured with a caliper after the fact.

The first two prototypes were drawn by hand and cut out of paper, then scrap cardboard. I didn't move to a final version cut out of mat board until I had tested the fit of the sides with a paper version. Score the folds on the opposite side of the board so they fold better.

Try It First, start by importing the libraries you need and configuring your network connection for the client. This is similar to what you did for the Air-Quality Meter in Chapter 4, but the server address will be your computer's addesss.

```
/*
  Joystick client
 Context:  Arduino

 This program enables an Arduino to control one paddle
 in a networked Pong game.
 */

#include <SPI.h>
#include <Ethernet.h>

byte mac[] = { 0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress ip(192,168,1,20);

// Enter the IP address of the computer on which
// you'll run the pong server:
IPAddress server(192,168,1,100);
```

» Change these to match your own device and network.

» You need a number of constants to keep track of the input and output pin numbers, the joystick's thresholds for left and right, and the minimum time between messages to the server. You'll need global variables for the connection Client, the previous state of the connect button, and the timestamp of the last message sent to the server.

You should update the left and right thresholds with the values you discovered earlier while testing the joystick.

```
const int connectButton = 2;  // the pushbutton for connecting/disconnecting
const int connectionLED = 3;  // this LED indicates whether you're connected
const int leftLED = 4;        // this LED indicates that you're moving left
const int rightLED = 5;       // this LED indicates that you're moving right
const int left = 200;         // threshold for the joystick to go left
const int right = 800;        // threshold for the joystick to go right
const int sendInterval = 20;  // minimum time between messages to the server
const int debounceInterval = 15;  // used to smooth out pushbutton readings

EthernetClient client;    // instance of the EthernetClient class for connecting
int lastButtonState = 0;  // previous state of the pushbutton
long lastTimeSent = 0;    // timestamp of the last server message
```

» The setup() method just opens serial and Ethernet communications and initializes the digital I/O pins.

```
void setup()
{
  // initialize serial and Ethernet ports:
  Ethernet.begin(mac, ip);
  Serial.begin(9600);
  // initialize digital inputs and outputs:
  pinMode(connectButton, INPUT);
  pinMode(connectionLED, OUTPUT);
  pinMode(leftLED, OUTPUT);
  pinMode(rightLED, OUTPUT);

  delay(1000);      // give the Ethernet shield time to set up
  Serial.println("Starting");
}
```

The main loop starts by updating the current time, which is used to keep track of how frequently you send to the server. Then, it checks to see whether the connect button has been pushed. If it has, and the client's already connected, the Arduino disconnects. If it's not connected, it tries to connect.

```
void loop()
{
  // note the current time in milliseconds:
  long currentTime = millis();
  // check to see if the pushbutton's pressed:
  boolean buttonPushed = buttonRead(connectButton);

  // if the button's just pressed:
  if (buttonPushed) {
    // if the client's connected, disconnect:
    if (client.connected()) {
      Serial.println("disconnecting");
      client.print("x");
      client.stop();
    } // if the client's disconnected, try to connect:
    else {
      Serial.println("connecting");
      client.connect(server, 8080);
    }
  }
```

Next, check whether the client's connected to the server, and whether enough time has elapsed since the last time you sent to the server. If so, read the joystick and, if it's at one extreme or the other, send the message to the server and turn on the appropriate LED. If the joystick's in the middle, turn off the LEDs. Then save the current time as the most recent time you sent a message.

You may have to adjust your sendInterval value, depending on the responsiveness of your server and the sensitivity of your sensors. 20 milliseconds is a good place to start, but if you find the server slowing down with lots of clients, make it higher.

Finally, set the state of the connection LED using the state of the client itself.

```
  // if the client's connected, and the send interval has elapsed:
  if (client.connected() && (currentTime - lastTimeSent > sendInterval)) {
    // read the joystick and send messages as appropriate:
    int sensorValue = analogRead(A0);
    if (sensorValue < left) {    // moving left
      client.print("l");
      digitalWrite(leftLED, HIGH);
    }

    if (sensorValue > right) {    // moving right
      client.print("r");
      digitalWrite(rightLED, HIGH);
    }
    // if you're in the middle, turn off the LEDs:
    if (left < sensorValue && sensorValue < right) {
      digitalWrite(rightLED, LOW);
      digitalWrite(leftLED, LOW);
    }
    //save this moment as last time you sent a message:
    lastTimeSent = currentTime;
  }

  // set the connection LED based on the connection state:
  digitalWrite(connectionLED, client.connected());
}
```

Finally, you can't just connect or disconnect 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 method checks for a low-to-high transition by comparing the state of the button with its previous state. It's called from the main loop.

This method also debounces the button, which means that it listens for a few extra milliseconds to see whether the button changes after the initial read. Sometimes a pushbutton can give several false readings for a few milliseconds, as the electrical contacts settle against one another. The cheaper the switch, the more common this is. If you find that your pushbutton gives multiple readings each time you push it, increase debounceInterval by a few milliseconds.

```
// this method reads the button to see if it's just changed
// from low to high, and debounces the button in case of
// electrical noise:

boolean buttonRead(int thisButton) {
  boolean result = false;
  // temporary state of the button:
  int currentState = digitalRead(thisButton);
  // final state of the button:
  int buttonState = lastButtonState;
  // get the current time to time the debounce interval:
  long lastDebounceTime = millis();

  while ((millis() - lastDebounceTime) < debounceInterval) {
    // read the state of the switch into a local variable:
    currentState = digitalRead(thisButton);

    // If the pushbutton changed due to noise:
    if (currentState != buttonState) {
      // reset the debouncing timer
      lastDebounceTime = millis();
    }

    // whatever the reading is at, it's been there for longer
    // than the debounce delay, so take it as the actual current state:
    buttonState = currentState;
  }
  // if the button's changed and it's high:
  if(buttonState != lastButtonState && buttonState == HIGH) {
    result = true;
  }

  // save the current state for next time:
  lastButtonState = buttonState;
  return result;
}
```

Client #2: A Balance Board Client

Client #2 is a balance board. To control it, you stand on the board and tilt left or right. To stay in the middle, you have to balance the board. An accelerometer at the center of the board senses it tilt from side to side. The physical interaction for this controller is very different than the last. It requires more action from you, and it takes more physical agility to operate. However, the behavior of this client is identical to the previous one—from the server's point of view.

The balance board itself is based on a design by the Adaptive Design Association (http://adaptivedesign.org), a New York City-based organization that gets people involved—families and community volunteers—in making safe and affordable furniture and equipment for children with disabilities. A balance board is great for helping to strengthen your balance, and it's fun as well.

The physical construction for the balance board client case is similar to the joystick client—you can use the same template with the same LED setup. Just make the joystick hole smaller to hold the pushbutton, and don't cut a second hole.

To construct the board, you'll need heavy-duty triple-wall cardboard. Many packaging supply houses carry it. If you can't get triple-wall cardboard, you can laminate three sheets of regular cardboard together using white glue. When you're gluing layers together, make sure the grain of the corrugations for adjacent layers runs perpendicular to each other, for added stability.

Cut two circles approximately 15 inches in diameter from a sheet of triple-wall cardboard. In one of them, cut two slots, 12 inches long and as thick as your cardboard. In the board shown in Figure 5-12, the slots are roughly an inch thick. Each one is two inches from the center diameter of the circle, as shown in Figure 5-11. Glue the two circles together with their grains opposing each other.

Next, cut two arcs 12 inches across, as shown in Figure 5-10. Glue the arcs into the slots and let them dry. Make sure it's very solid before you try to balance on it.

The case for the balance board was adapted from the joystick client. The differences are that the connect/disconnect button took the place of the joystick, so the second large hole could be removed and the whole box made shorter. Mount the control box at the center of your balance board, as shown in Figure 5-12.

The code for the balance board client is also very similar to the joystick client. All you have to change is the values for the left and right thresholds. To discover them, do as you did with the joystick: put the accelerometer on the balance board, program the Arduino to print out the value of the analog input, and tilt it both ways to learn its extreme. Then fill those values in for left and right. 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 as 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.

X

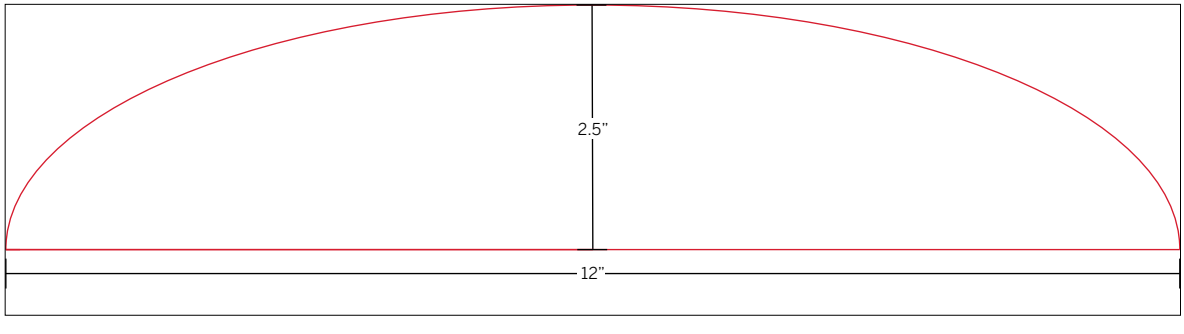


Figure 5-10
The balance board rocker. Make two of these.
Design courtesy of the Adaptive Design Association.

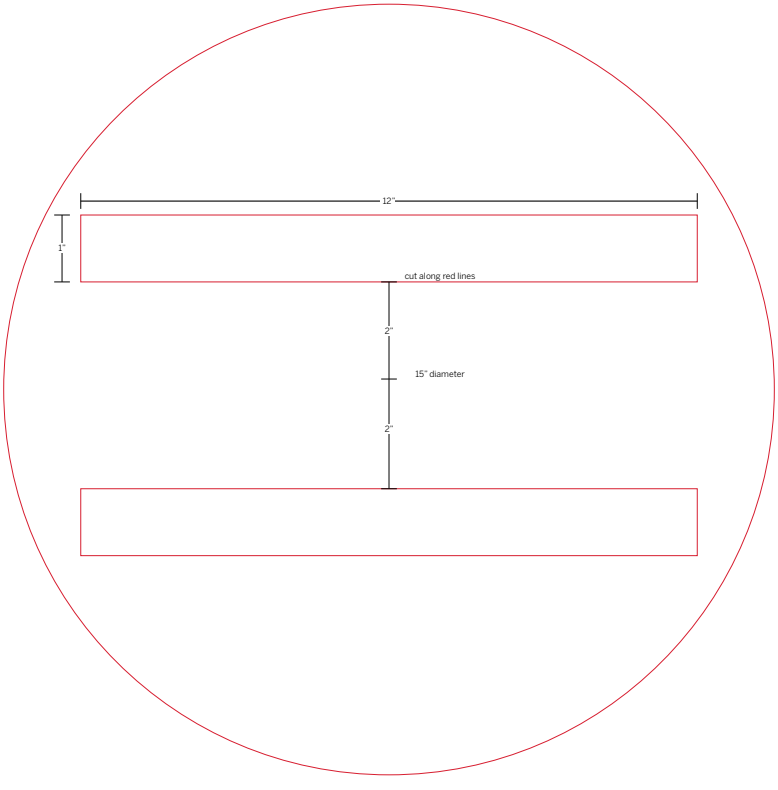


Figure 5-11
The balance board base. Cut two circles this size, and cut the slots for the rockers in only one of them. Then, laminate the two circles together. *Design courtesy of the Adaptive Design Association.*

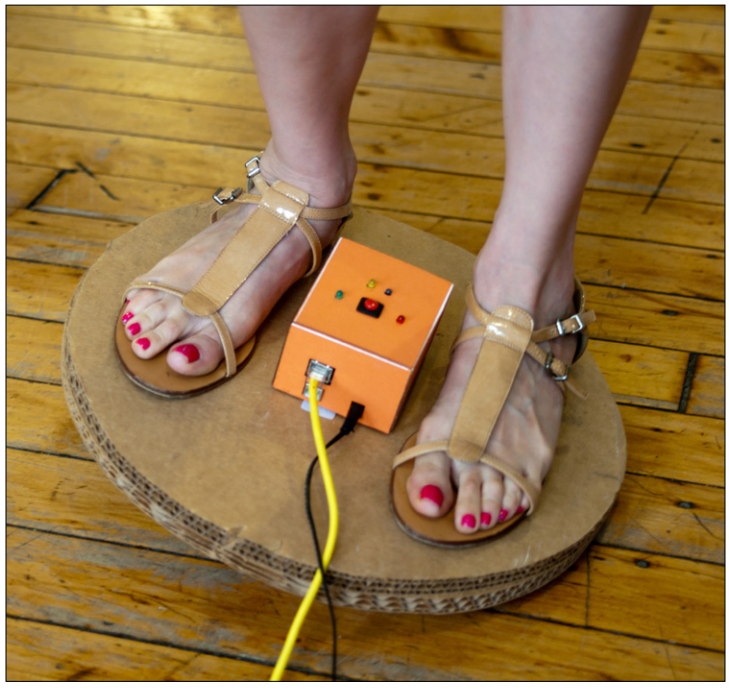


Figure 5-12
The balance board in action. The two layers of triple-wall cardboard are laminated with the grain of their corrugations perpendicular to each other. This gives the board added stability. A couple of tabs of Velcro hold the controller box in place.

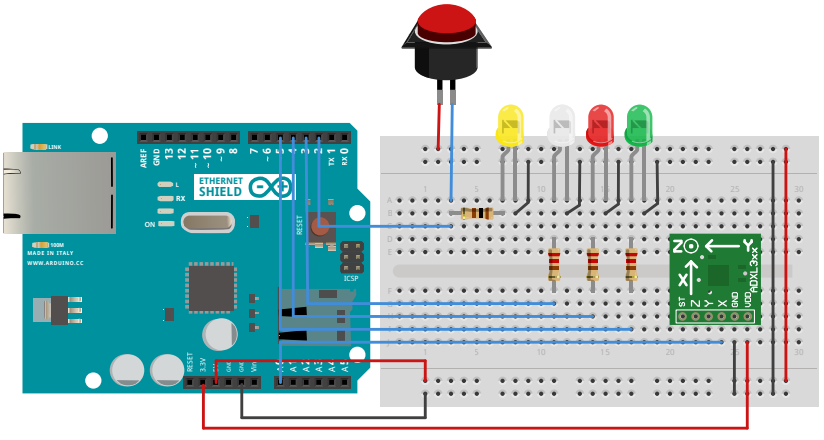


Figure 5-13
The balance board client circuit, breadboard version. You can see it's identical to the joystick client, except for the accelerometer in place of the joystick.

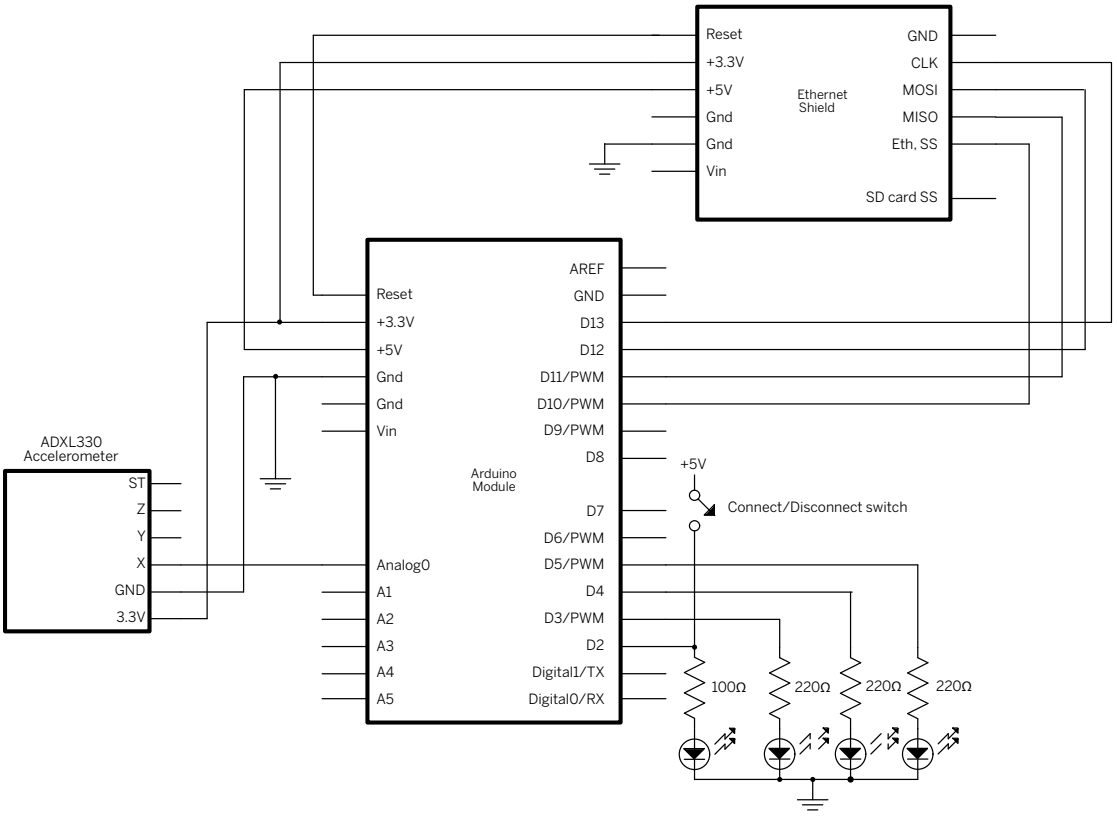


Figure 5-14
The balance board client schematic.

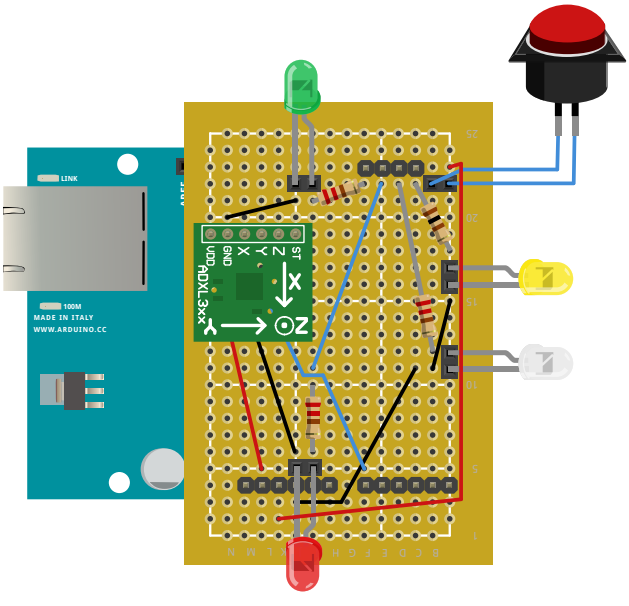


Figure 5-15
The balance board client circuit laid out on a perforated circuit board.

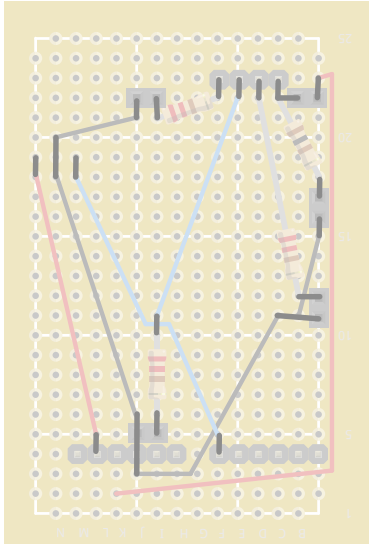


Figure 5-16
The circuit board layout showing where the solder joints are connected underneath the board.

The Server

The server's tasks can be divided into two groups: those related to the game play, like animating the paddles and the ball and scoring; and those 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 vertical positions. 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 an IP address from its client. 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 (or in Java for that matter), 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;
```

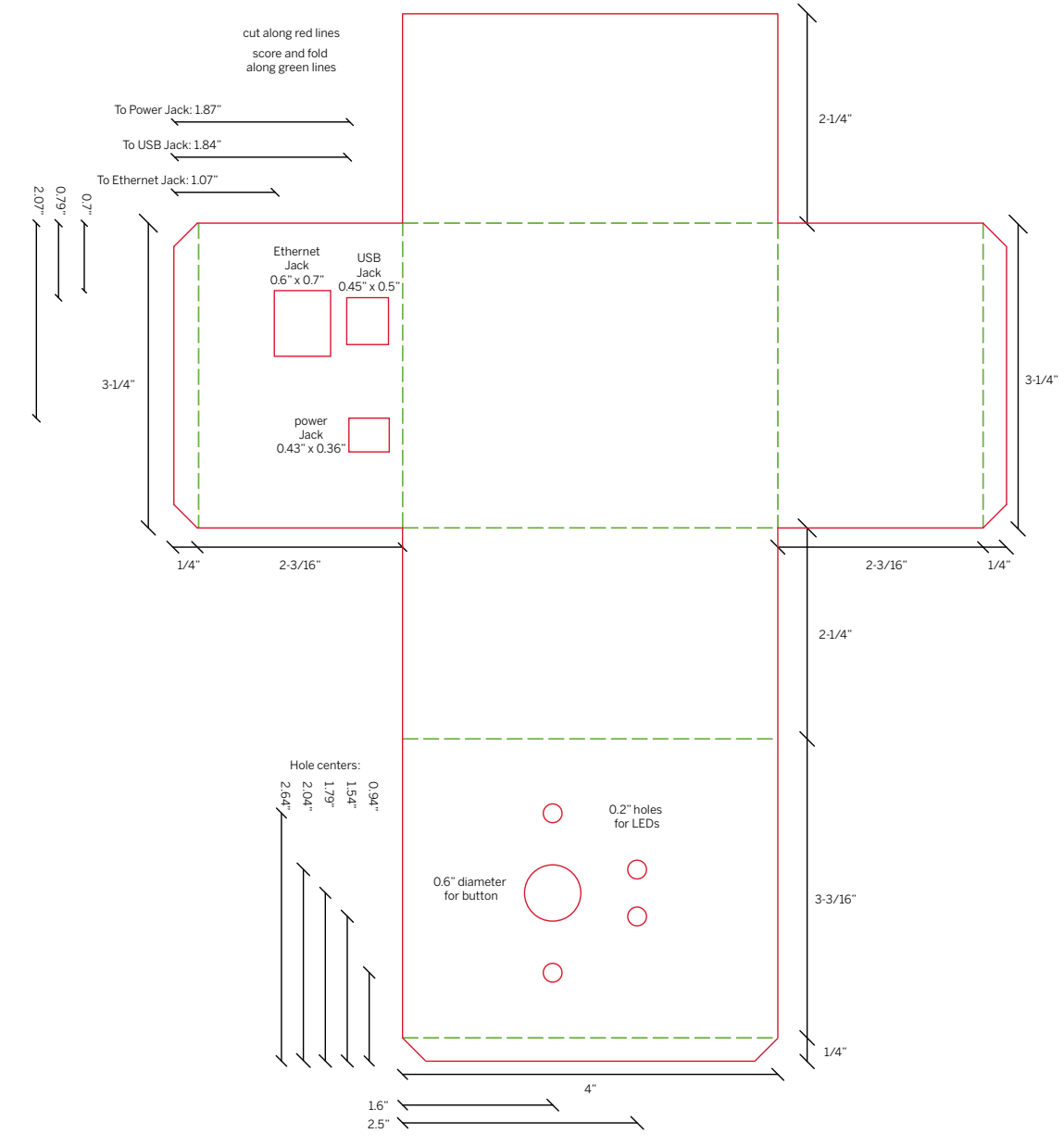


Figure 5-17
Template for the balance board housing. As with the joystick client, the template can be cut from 1/16" mat board. The dimensions were changed to meet the needs of the smaller circuit.

As shown in the example, the variables declared at the beginning of the class are called instance variables. Every new instance of the class created makes its own copies of these variables. Every class has a constructor method, which gets called to bring 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, dataRate).

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 object's instance variables (paddleH, paddleV, and client) 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(xPosition, yPosition, 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 flowchart of what happens. Figure 5-18 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, and 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.

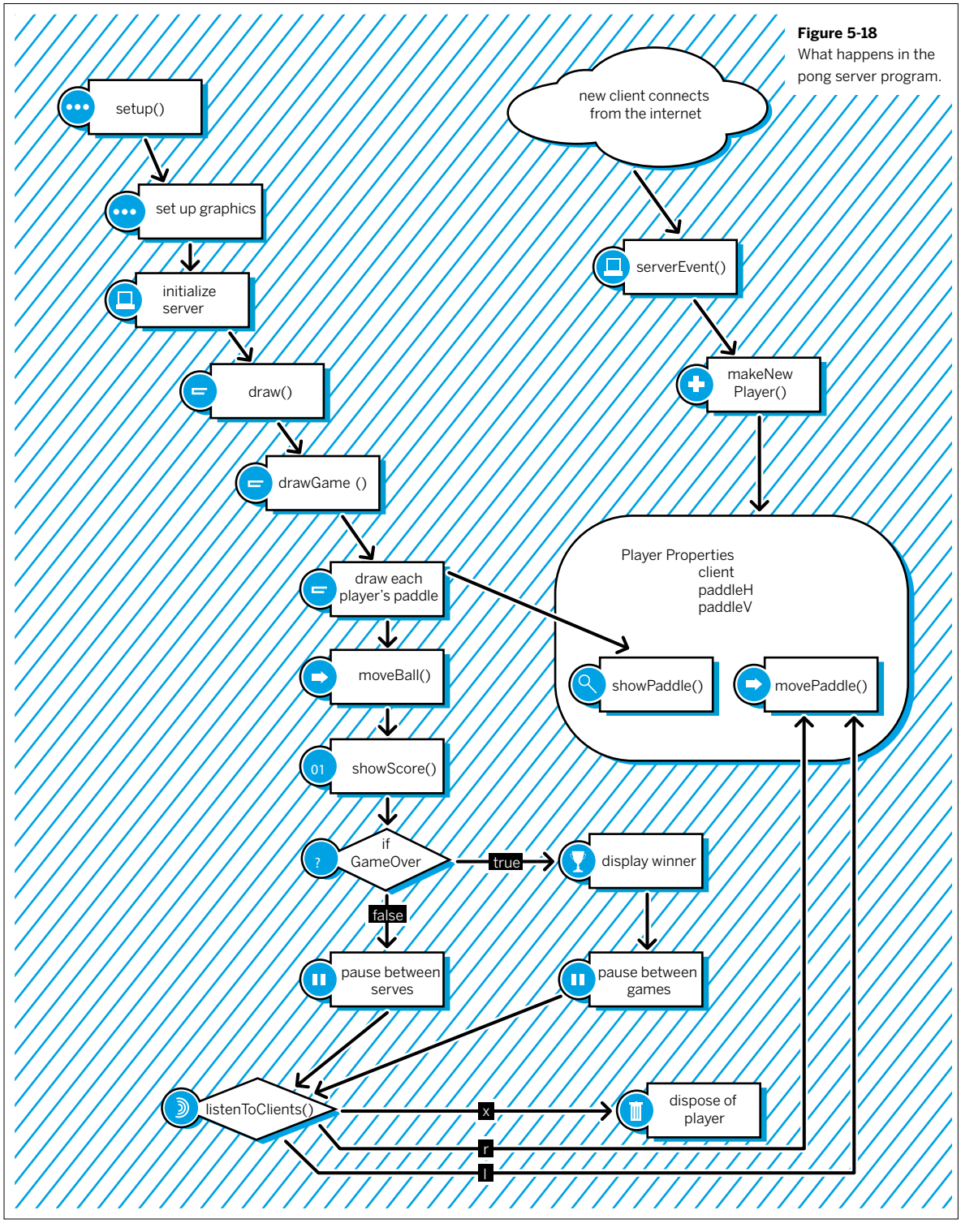


Figure 5-18
What happens in the pong server program.

The first thing to do in the server program 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

// 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 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
long delayCounter;                  // 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 sets all the initial conditions for the game, and then starts the server.

```
void setup() {
  // set the window size:
  size(480, 640);
  // set the frame rate:
  frameRate(90);
  // set the default font alignment:
  textAlign(CENTER);
  // 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);

  // set up all the pong details:
  // initialize the delay counter:
  delayCounter = millis();
  // initialize 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;

  // Start the server:
  myServer = new Server(this, port);
}
```

The draw() method updates the screen using a method called drawGame(), and it listens for any messages from existing clients using the listenToClients() method.

```
void draw() {
  drawGame();
  listenToClients();
}
```

When new clients connect to the server, the net library's serverEvent() method is called automatically; your Processing sketch has to have 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(). At right is the serverEvent() method.

```
// The ServerEvent message is generated when a new client
// connects to the server.
void serverEvent(Server thisServer, Client thisClient) {
  if (thisClient != 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 newPlayer = (Player)playerList.get(p);

      // if thisPlayer's client matches the one that generated
      // the serverEvent, then this client is already a player, so quit
      // out of the method and return:
      if (newPlayer.client == thisClient) {
        return;
      }
    }

    // if the client isn't already a Player, then make a new Player
    // and add it to the playerList:
    makeNewPlayer(thisClient);
  }
}
```

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.

A new Player is added to the bottom team if the last player is on the top, and vice versa.

The variables nextTopPaddleV and nextBottomPaddleV keep track of the positions for the next players on each team.

```
void makeNewPlayer(Client thisClient) {
  // paddle position for the new Player:
  int x = width/2;
  // if there are no players, add to the top:
  int y = nextTopPaddleV;

  /*
  Get the paddle position of the last player on the list.
   If it's on top, add the new player on the bottom, and vice versa.
   If there are  no other players, add the new player on the top.
   */
  // get the size of the list:
  int listSize = playerList.size() - 1;
  // if there are any other players:
  if  (listSize >= 0) {
    // get the last player on the list:
    Player lastPlayerAdded = (Player)playerList.get(listSize);
    // is the last player's on the top, add to the bottom:
    if (lastPlayerAdded.paddleV == nextTopPaddleV) {
      nextBottomPaddleV = nextBottomPaddleV - paddleHeight * 2;
      y = nextBottomPaddleV;
    }
    // is the last player's on the bottom, add to the top:
    else if (lastPlayerAdded.paddleV == nextBottomPaddleV) {
```

Continued from opposite page.

```
      nextTopPaddleV = nextTopPaddleV + paddleHeight * 2;
      y = nextTopPaddleV;
    }
  }

  // make a new Player object with the position you just calculated
  // and using the Client that generated the serverEvent:
  Player newPlayer = new Player(x, y, thisClient);
  // add the new Player to the playerList:
  playerList.add(newPlayer);
  // Announce the new Player:
  println("We have a new player: " + 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 and convert it
    // to a Player:
    Player thisPlayer = (Player)playerList.get(p);
    // compare the client of thisPlayer to the client that sent a message.
    // If they're the same, then this is the Player we want:
    if (thisPlayer.client == speakingClient) {
      speakingPlayer = thisPlayer;
      break;
    }
  }

  // read what the client sent:
  if (speakingPlayer != null) {
    int whatClientSaid = speakingPlayer.client.read();
    /*
  There are a number of things it might have said that we care about:
     x = exit
     l = move left
     r = move right
     */
    switch (whatClientSaid) {
      // If the client says "exit", disconnect it
    case 'x':
      // say goodbye to the client:
      speakingPlayer.client.write("bye\r\n");
```

Continued from previous page.

```
      // disconnect the client from the server:
      println(speakingPlayer.client.ip() + "\t logged out");
      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 an "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.

drawGame(), 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 drawGame() {
  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 each Player's showPaddle() method. This is why the two methods are separated in the object.

Likewise, the moveBall() method, called here, checks to see whether the ball hit a paddle or a wall. It then calculates its new position from there, but it doesn't draw the ball itself because 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.

Continued from previous page.

```
  // 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 playing.

That closes out the drawGame() 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. Those are shown next.

```
// 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);
      // reset the score:
      newGame();
    }
  }
}
```

First, moveBall() checks whether 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 whether the ball position is contained within the rectangle of the paddle. If the ball does intersect a paddle, 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)) {
```

Continued from previous page.

```
    // calculate the vertical edges of the paddle:
    float paddleTop = thisPlayer.paddleV - paddleHeight/2;
    float paddleBottom = thisPlayer.paddleV + paddleHeight/2;

    // check to see if the ball is in the
    // horizontal range of the paddle:
    if ((ballPosV >= paddleTop) && (ballPosV <= paddleBottom)) {
      // reverse the ball's vertical direction:
      ballDirectionV = -ballDirectionV;
    }
  }
}
```

» If the ball goes above the top of the screen or below the bottom, then one team has scored. If any team goes over five points, the game is over.

```
// 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 restarts 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 the resetBall() method, which resets the ball at the end of each point.

```
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 as 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 5-19 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. Try to make the ultimate *ping* pong paddle.

X

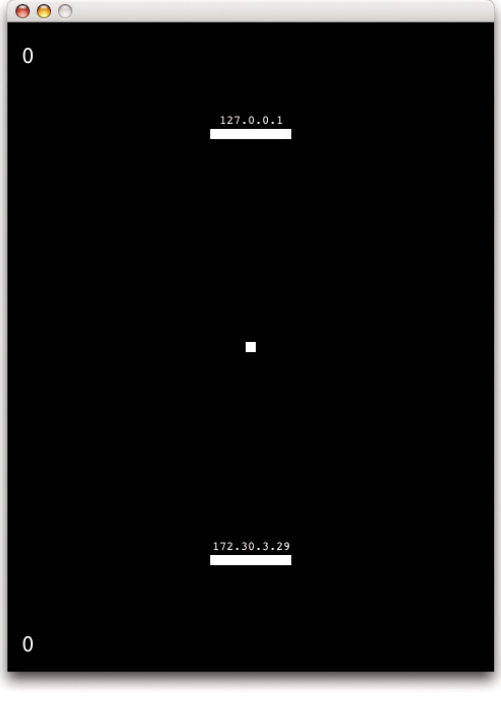


Figure 5-19
The output of the ping pong server sketch.

Conclusion

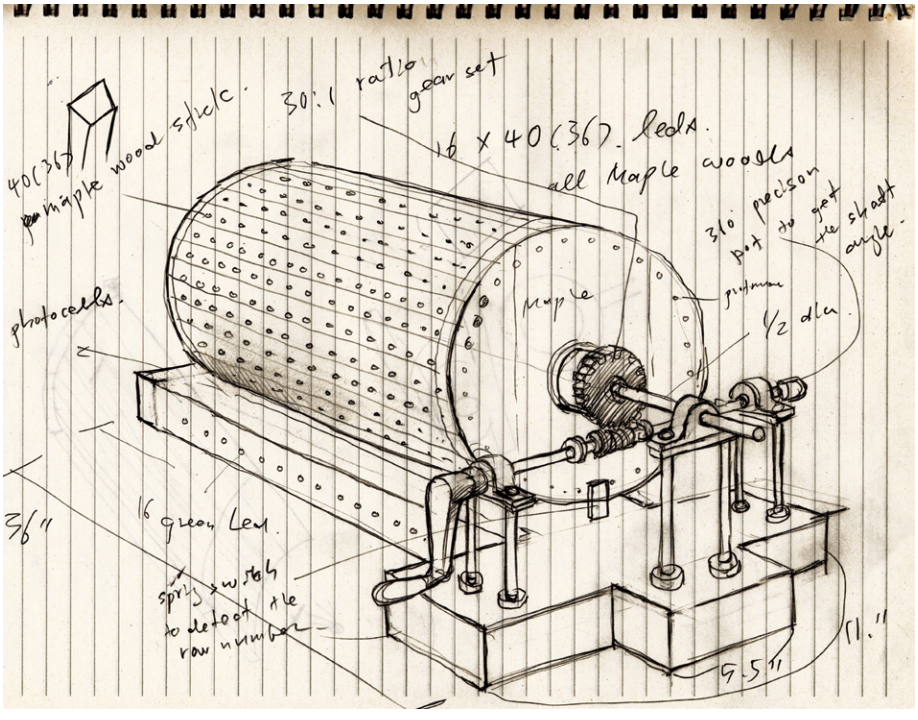
The basic structure of the clients and server in this chapter can be used any time you want to make a system that manages synchronous connections between several objects on the network. The server's main jobs are to listen for new clients, keep track of the existing clients, and make sure that the right 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 for more 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.

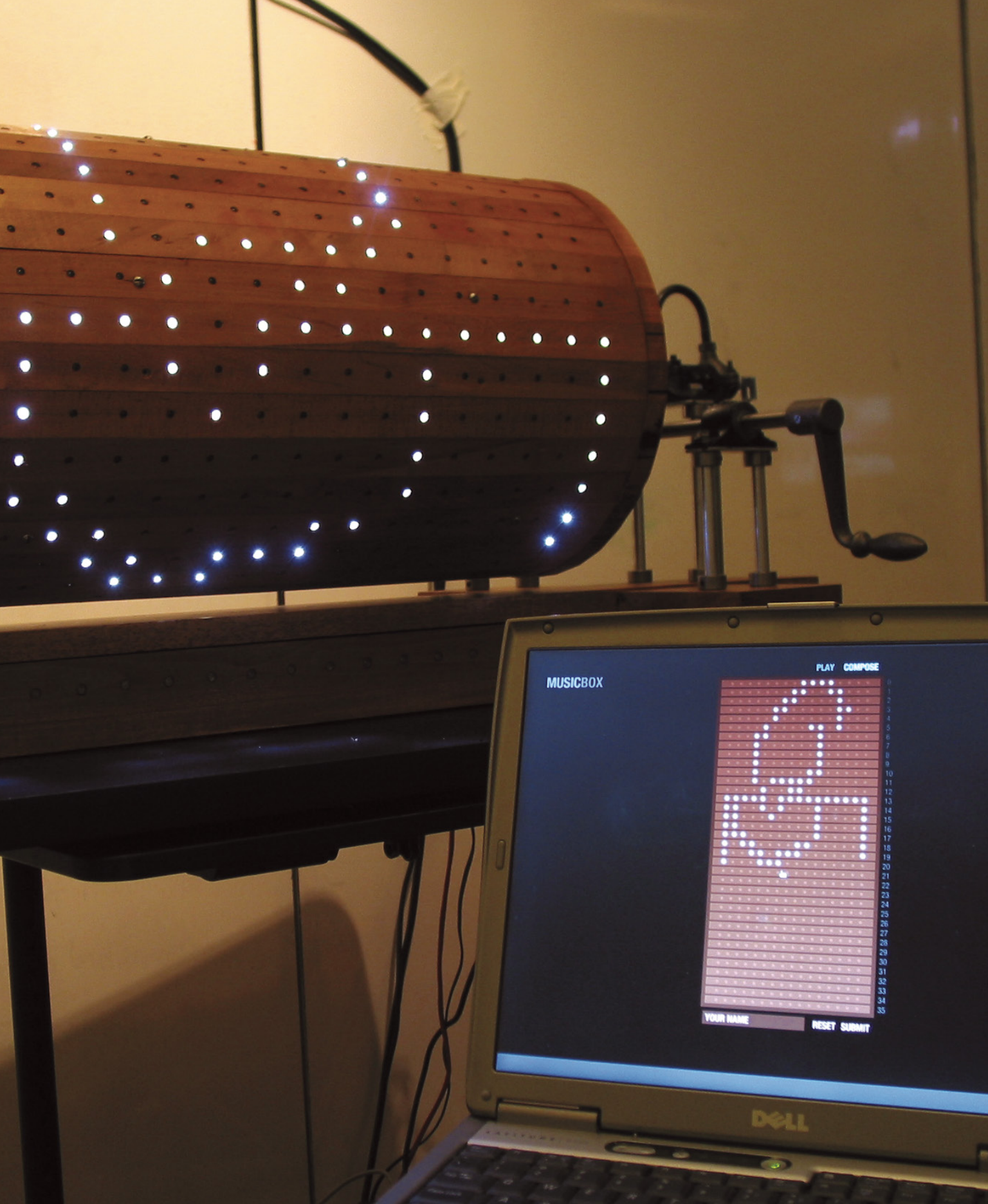
Finally, make sure you've got a reliable way to test the system. Simple tools like a telnet client and test server will save you much time in building every multiplayer server, and help you get to the fun sooner.

Now you've seen examples of both asynchronous client-server exchanges (the HTTP system in Chapter 4) and synchronous exchanges (the chat server on page 153). 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 left*
Jin-Yo Mok's original sketches of the music box.

▶▶ *At right*
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 reading this chapter. This chapter explains the basics of wireless communication between objects. In it, you'll learn about two types of wireless communication, and then 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 they use ZigBee radios to communicate with a central computer. 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.*

Supplies for Chapter 6

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **RS** RS (www.rs-online.com)
- **SF** SparkFun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 9: Infrared Control of a Digital Camera

» **1 Arduino module** An Arduino Uno or something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards. **D** 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 infrared LED** **J** 106526, **SF** COM-09469, **F** 1716710, **RS** 577-538, **SS** MTR102A2B

» **1 pushbutton** Any button will do. **D** GH1344-ND, **J** 315432, **SF** COM-10302, **F** 1634684, **RS** 718-2213

» **1 220-ohm resistor** **D** 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842

» **1 10-kilohm resistor** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906

» **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

PROJECT 10: Duplex Radio Transmission

» **2 solderless breadboards** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M

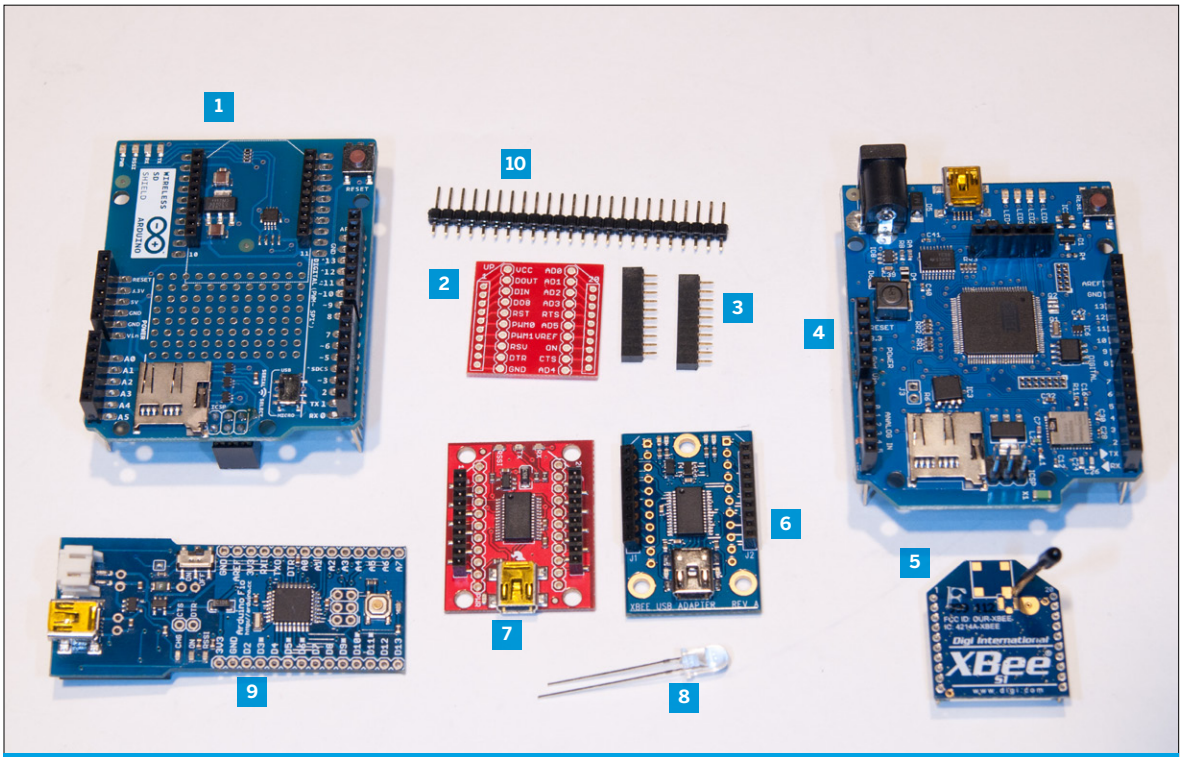


Figure 6-1. New parts for this chapter: **1.** Arduino wireless shield **2.** Spark Fun XBee breakout board **3.** 2mm female header pins **4.** Arduino WiFi shield **5.** Digi XBee 802.15.4 OEM module **6.** Adafruit XBee-to-USB adapter **7.** Spark Fun XBee Explorer **8.** Infrared LED **9.** Arduno Fio. Don't forget plenty of male header pins for the breakout boards.

- **2 Arduino modules** Arduino Fio models are a nice alternative designed to work with XBees, but any Uno-compatible board should work.
  SF DEV-10116
- **2 Digi XBee 802.15.4 RF modules** **J** 2113375, **SF** WRL-08664 , **AF** 128, **F** 1546394, **SS** WLS113A4M, **MS** MKAD14
- **2 Arduino wireless shields** You can opt to not use the shields and use the parts listed below instead.
  A A000064 or A000065. Alternative shields: **SF** WRL-09976, **AF** 126, **F** 1848697, **RS** 696-1670, **SS** WLS114A0P
- **2 potentiometers** **J** 29082, **SF** COM-09939, **F** 350072, **RS** 522-0625
- **1 USB-XBee adapter**
  The following parts are only necessary if you are not using Wireless shields.
  J 32400, **SF** WRL-08687, **AF** 247
- **2 3.3V regulators** **J** 242115, **D** 576-1134-ND, **SF** COM-00526, **F** 1703357, **RS** 534-3021
- **2 1µF capacitors** **J** 94161, **D** P10312-ND, **F** 8126933, **RS** 475-9009
- **2 10µF capacitors** **J** 29891, **D** P11212-ND, **F** 1144605, **RS** 715-1638
- **2 XBee breakout boards** **J** 32403**, SF** BOB-08276, **AF** 127
- **4 rows of 0.1-inch header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411
- **4 2mm female header rows** **J** 2037747, **D** 3M9406-ND, **F** 1776193
- **6 LEDs** **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590

PROJECT 11: Bluetooth Transceivers

- **2 Arduino modules** Get something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards.
  D 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4
- **2 solderless breadboards** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M
- **2 LEDs** **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590
- **2 potentiometers** Any analog sensor will work.
  J 29082, **SF** COM-09939, **F** 350072, **RS** 522-0625
- **2 pushbuttons** Any will do.
  D GH1344-ND, **J** 315432, **SF** COM-10302, **F** 1634684, **RS** 718-2213
- **2 220-ohm resistors** **D** 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842
- **2 10-kilohm resistors** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906
- **1 FTDI USB-to-Serial adapter** Both the 5V or 3.3V versions will work; these come as cables or standalone modules. **SF** DEV-09718 or DEV-09716, **AF** 70, **A** A000059, **MS** MKAD22, **SS** PRO101D2P, **D** TTL-232R-3V3 or TTL-232R-5V
- **2 Bluetooth Mate modules** **SF** WRL-09358 or WRL-10393

PROJECT 12: Hello WiFi!

- **1 Arduino WiFi shield** **A** A000058
- **1 Arduino module** Get something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards.
  D 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4
- **1 WiFi Ethernet connection to the Internet**
- **3 10-kilohm resistors** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906
- **3 photocells (light-dependent resistors)** **D** PDV-P9200-ND, **J** 202403, **SF** SEN-09088, **F** 7482280, **RS** 234-1050
- **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2
- **3 lighting filters** One primary red, one primary green, and one primary blue. Available from your local lighting- or photo-equipment supplier.

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 covers 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 forms of wireless device.
X

Why Isn't Everything Wireless?

The advantage of wireless communication seems obvious: 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 this 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 WiFi radios (802.11b, g, and n), and ZigBee (802.15.4) radios all work in the same frequency range: 2.4 gigahertz (802.11n will also work at 5GHz). 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 WiFi 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 first make sure that you have the communication working 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. These work regardless of whether the transmitter and receiver are facing each other. 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; Figure 6-2 illustrates this difference.

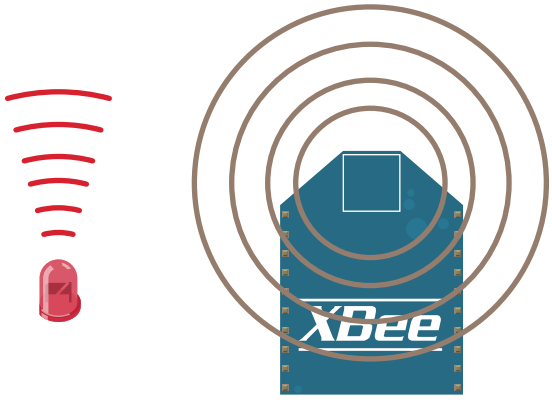


Figure 6-2
The signal from the LED at left radiates out in a beam from the LED, while the signal from a radio antenna like on the XBee radio at right radiates omnidirectionally.

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 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. It's more complex to make a transceiver than it is to make 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. For many applications, it's cheaper to use a transmitter-receiver pair and handle any errors by just transmitting the message many times until the receiver gets it. That's how TV remote controls work, for example. It makes the components much cheaper.

It's increasingly common in radio applications to just make every device a transceiver, and incorporate a microcontroller to manage the transmitter-receiver filtering. All Bluetooth, ZigBee, and WiFi radios work this way. However, it's still possible to get transmitter-receiver pair radios, and they are still cheaper than their transceiver counterparts.

Keep in mind the distinction between transmitter-receiver pairs and transceivers when you plan your projects, and when you shop. Consider whether the communication in your project must 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, 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 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, 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. However, 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 carrier wave's frequency 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 2,000 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: 12 bit, 15 bit, and 20 bit. 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 good sites on the Web that 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. There are also a number of libraries written for Arduino to help you send and receive IR signals for different protocols. Many

Making Infrared Visible

There are two tools that are really helpful when you're working with IR transmitters and receivers: a camera and an oscilloscope.

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. If it's working, you'll see the LED light up. Figure 6-3 shows 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 or close the curtains to see this effect. Some webcams have a built-in IR filter, so it's good to first check with an IR device that you know works, like a remote control, before you use it to detect whether your project is working.

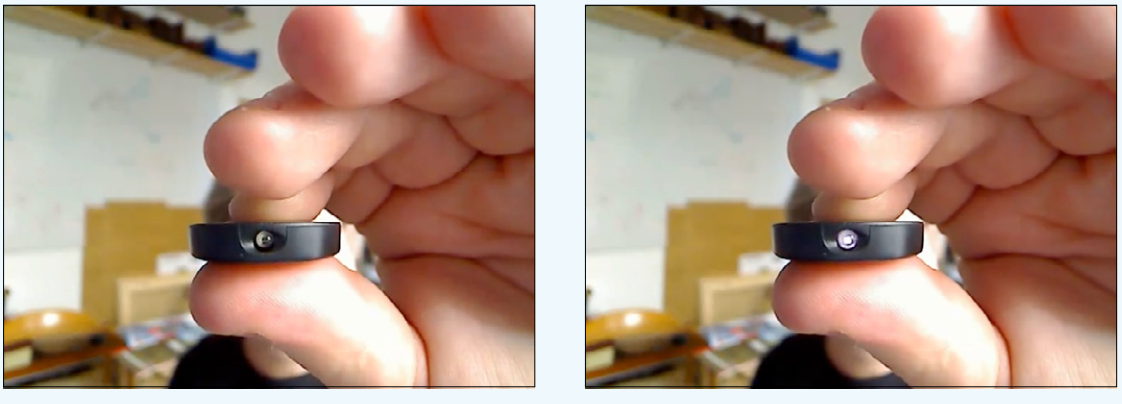


Figure 6-3. Having a camera at hand is useful when troubleshooting IR projects.

of them are listed on the Arduino playground at http://arduino.cc/playground/Main/InterfacingWithHardware. You'll see one in action in the next project.

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 well.

X

Sniffing Infrared Signals

An oscilloscope is also useful when you're trying to decipher an IR signal (see Figure 6-4). You may not know the protocol for your receiver, but you can work it out by looking at the signal it sent. Connect an infrared phototransistor, a resistor, and a regular LED in series, as shown in Figure 6-5, and you should see the LED light up when you point your remote at the phototransistor.

To see the remote's signal on an oscilloscope, connect the scope probes to ground and to the phototransistor's emitter, and fire the remote at the phototransistor. Once you see activity, adjust the voltage and time divisions on the scope until you see readable activity. Most scopes will tell you the frequency of the signal automatically. Putting your scope in single-shot trigger mode will help you capture the actual signal. Once you can see the timing of each signal's pulse, you can work out how to duplicate it by generating your own pulses on an IR LED. For more on this, see many of the excellent blog posts on IR remote control using an Arduino. For example, read Ken Shirriff's very good explanation on his blog, at www.arcfn.com.

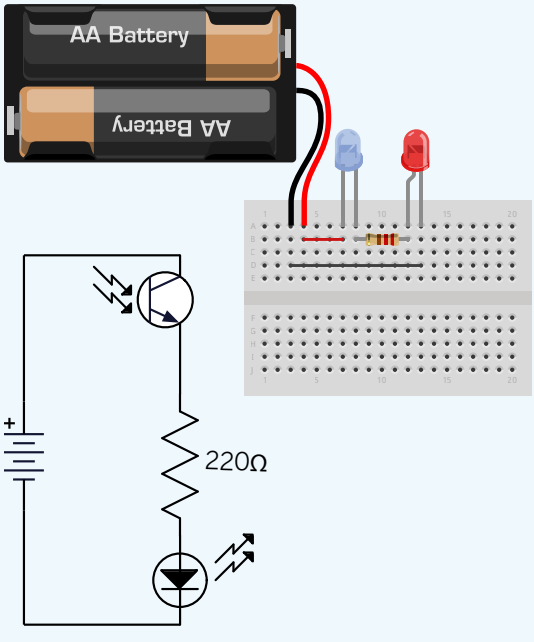


Figure 6-5. An IR phototransistor and LED in series work well to test IR reception.

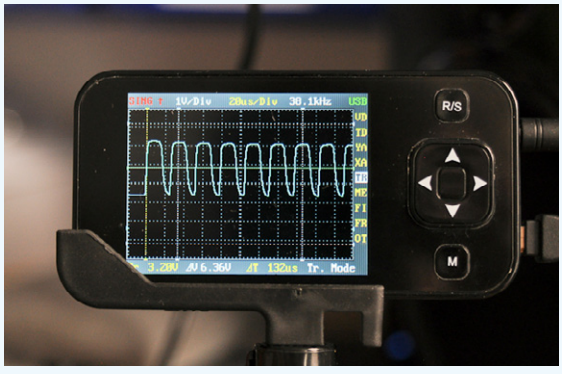


Figure 6-4. An oscilloscope can help you see the pattern of an IR signal.

MATERIALS

- **1 solderless breadboard**
- **1 220-ohm resistor**
- **1 phototransistor** Digi-Key part no. 365-1068-ND.
- **1 LED**
- **1 battery or power source** 5V or less.
- **1 oscilloscope** DSO nano shown here.

Project 9

Infrared Control of a Digital Camera

This example uses an infrared LED and an Arduino to control a digital camera. It's about the simplest IR control project you can do.

MATERIALS

- **1 Arduino module**
- **1 infrared LED**
- **1 pushbutton**
- **1 220-ohm resistor**
- **1 10-kilohm resistor**
- **1 solderless breadboard or prototyping shield**

Most digital SLR cameras on the market today can be controlled remotely via infrared. Each brand uses a slightly different protocol, but they all tend to have the same basic commands: trigger the shutter, trigger after a delay, and auto-focus. Sebastian Setz has written an Arduino library that can send the signals for most common cameras. It's been tested with Canon, Nikon, Olympus, Pentax, and Sony. If you have an SLR from any of these brands, you should be able to control it with this library.

The circuit for this project is simple. Connect the push-button to pin 4 of the microcontoller (with a 10-kilohm pulldown resistor), and connect the infrared LED to pin 3 of the microcontroller, as shown in Figure 6-6.

Download the Multi Camera IR Control library from http://sebastian.setz.name/arduino/my-libraries/multi-camera-ir-control and copy it to the **libraries** directory of your Arduino sketch directory. If you've never installed a library before, you'll need to create this directory. One it's there, restart the Arduino application, and you should see a new library in the Sketch menu's Import Library submenu called MultiCameraIrControl. Now you're ready to get going.

X

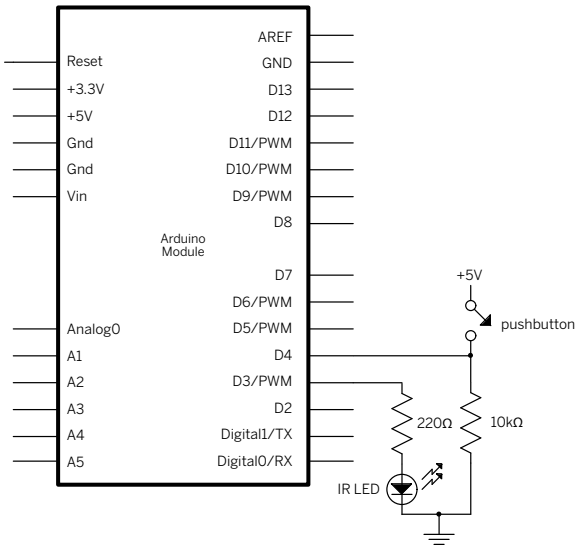
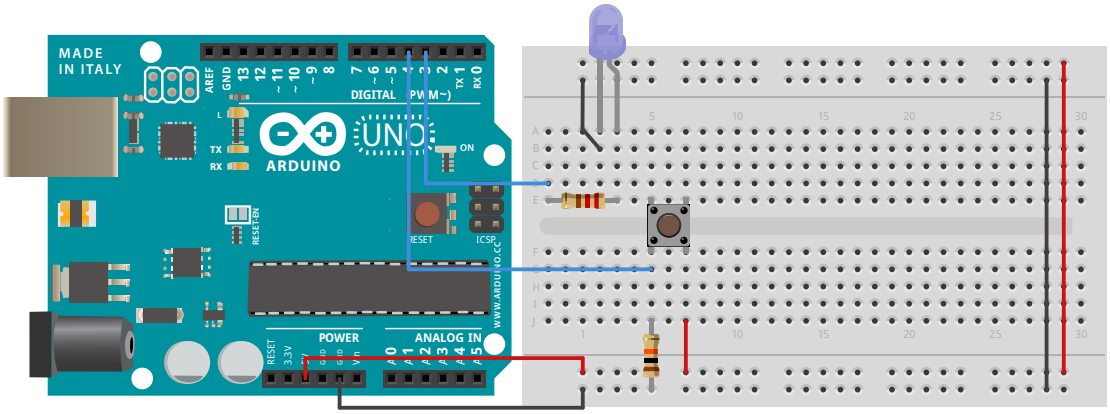


Figure 6-6
The microcontroller with an IR LED and pushbutton attached.



Try It To start your sketch, import the MultiCameraIrControl library. Initialize the library to send signals on pin 3, to which the LED is connected. Then, set up a few variables to keep track of the state of the pushbutton.

```
/*
 IR Camera control
 Context: Arduino

 This sketch controls a digital camera via an infrared LED.
 */

// include the library for camera control:
#include <multiCameraIrControl.h>

const int pushButtonPin = 4;
// set up pin 3 to control the IR LED.
// change this depending on the brand of your camera:
Nikon camera(3);

// Variables will change:
int buttonState = 0;         // current state of the button
int lastButtonState = 0;     // previous state of the button
```

» The setup method initializes the pushbutton as an input.

```
void setup(){
  // initialize the pushButton as input:
  pinMode(pushButtonPin, INPUT);
}
```

» The main loop listens for the pushbutton's state to change. Since you don't want the camera firing all the time, trigger the camera only when the pushbutton changes from OFF to ON. To do that, compare the button's state to its previous state by storing the current state as the previous state at the end of each loop.

That's the whole program. Now point the LED at your camera and start taking some pictures remotely. You may have to set your camera to remote control mode. Check the camera manual for how to do this, as it's different from camera to camera.

```
void loop(){
  // read the pushbutton input pin:
  buttonState = digitalRead(pushButtonPin);

  // compare the buttonState to its previous state
  // if it's changed, and it's high now, then the person
  // just punched the button:
  if (buttonState != lastButtonState && buttonState == HIGH) {
    // send the signal to open the shutter:
    camera.shutterNow();
  }
  // save the current state as the last state,
  //for next time through the loop
  lastButtonState = buttonState;
}
```



A. PIR sensor

Figure 6-7
This intervalometer was built using the same methods as shown above. An Arduino in the box senses a change from the PIR sensor and sends an IR signal to the camera to take a picture.

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 = 14,266.06 cm. / frequency in 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 the antennas you'll use in this chapter are omnidirectional, radio 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 radio frequency (RF) 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 an RF 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. 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. 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. Bodies of water block RF effectively as well. 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 hear 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. WiFi operates at frequencies in the gigahertz range, commonly called the microwave range, because the wavelength of those signals is very short compared to lower frequency signals. To cook food, microwave ovens generate energy in this range to excite (heat up) the water molecules in 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, which 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, there are many ways to interfere with a radio signal. 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 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 know who was talking to you was if each person clearly identified himself at the beginning and end of his conversation, and no one interrupted him during this time. In other words, it's all about protocols.

The first thing everyone at that cocktail party would have to do is agree on who speaks when. That way they could each have your attention for awhile. 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 sync. 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 sending 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 person spoke in a different tone, you could distinguish each individual by her tone. 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 becausethe radios handle it for you.

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. To make sure the message is clear, it's common to use a data protocol on top of using multiplexing. For example, Bluetooth, ZigBee, and WiFi 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, WiFi 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.

Radio Transmitters, Receivers, and Transceivers

How do you know whether to choose a radio transmitter-receiver pair, or a pair of transceivers? The simplest answer is that if you need feedback from the device to which you're transmitting, then you need a transceiver. Most of the time, it's simplest to use transceivers. In fact, as transceivers have become cheaper to make (and therefore sell), transmit-receive pairs are getting harder to find.

There are many different kinds of data transceivers available. The simplest digital radio transceivers on the market connect directly to the serial transmit and receive pins of your microcontroller. Any serial data 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. They're simple to connect, but you have to manage the whole conversation yourself. If the receiving transceiver misses a bit of data, you'll get a garbled message. Any nearby radio device in the same frequency range can affect the quality of reception. As long as you're working with just two radios and no interference, transceivers like this do a decent job. However, this is seldom the case.

Nowadays, most transceivers on the market 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 handled error-checking for you. The XBee radios you'll use in the next project will do the same, and much more, which you'll see in Chapter 7. 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.

The biggest difference between networked radios and simple transceivers is that every device on a network has an address. That means you have to decide which other device you're speaking to, or whether you're speaking to all the other devices on the network.

Because of the complications of network management, all networked radios generally have two modes of operation: command and data modes (as described in Chapter 2). When looking at the communications protocol for a networked radio, one of the first things you'll learn is how to switch from command mode to data mode and back.
X

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 10 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.

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 which to send. 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 they can pass messages back and forth.

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 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 to make the microcontroller send messages when the potentiometer changes, and to 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.

In this example, you won't actually use any of the features of ZigBee or the 802.15.4 protocol. Those protocols are designed to allow many different devices to communicate in a multitiered networking scheme. Each radio has an address, and every time it sends a message, it has to specify the address to which to send. It can also send a broadcast message, addressed to every other radio in range—you'll see how to send broadcast messages in Chapter 7. For now, you'll give each of your two radios the other's address so they can pass messages back and forth.

MATERIALS

- **2 solderless breadboards**
- **1 USB-to-XBee adapter**
- **2 Arduinos** Arduino Fio models are a nice alternative designed to work with XBees
- **2 Digi XBee 802.15.4 RF modules**
- **2 Arduino wireless shields**
  If you choose not to use wireless shields or Arduino Fios, you can use the parts below to connect the XBees to the Arduinos:
- **2 3.3V regulators**
- **2 1µF capacitors**
- **2 10µF capacitors**
- **2 XBee breakout boards**
- **4 rows of 0.1-inch header pins**
- **4 2mm female header rows**
- **6 LEDs**
- **2 potentiometers**

Step 1: Configuring the XBee Modules Serially

The easiest way to connect an XBee to your personal computer is to use an XBee-to-USB serial adapter. Since this book's first edition was published, the popularity of XBees has grown exponentially, and multiple versions are now available (most hobbyist electronics vendors sell a version). They're all basically a USB-to-Serial adapter mounted on a board with pins spaced to fit an XBee radio. The first image in Figure 6-12 shows two options: Adafruit's XBee USB adapter board and Spark Fun's XBee Explorer. Both have LEDs mounted to indicate serial transmit and receive. Adafruit's model also has LEDs to indicate whether the radio is associated with a network or whether it's asleep. The sleep mode indicator LED is attached to pin 13, which goes low when the radio is in sleep mode, and high when it's active. The associate indicator LED is attached to pin 15. When the radio's associated, this LED will blink.

Plug your XBee into the adapter, connect it to your computer's USB port, and open your favorite serial terminal program.

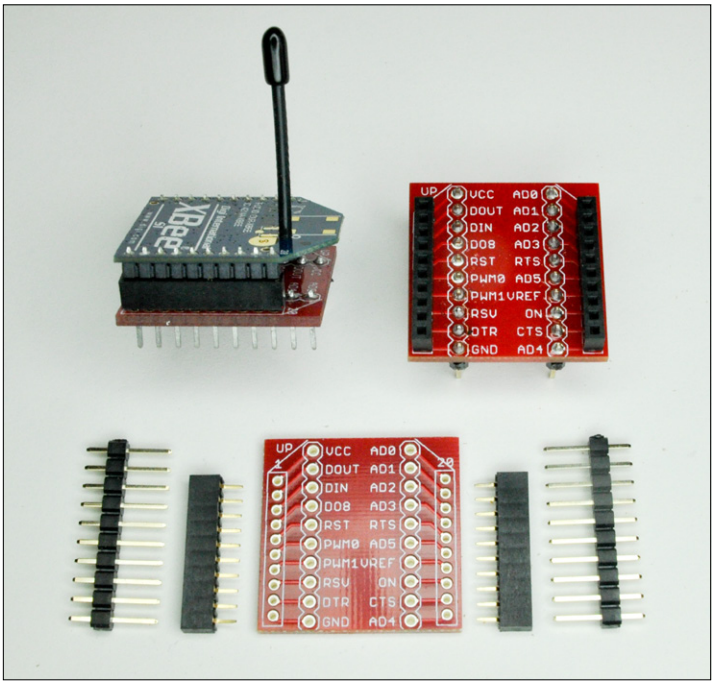


Figure 6-8
XBee breakout board, in various stages. *Bottom:* bare board with necessary headers. *Top right:* finished board. *Top left:* finished board with XBee mounted.

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). Once you've got the breakout board, solder headers to the inner rows. These will plug into your breadboard. Next, attach the 2mm female headers. The XBee will plug into these, so you may find it useful to align them to the board by plugging them into the XBee first, then plugging it with the headers attached into the board.

The XBee command protocol is particular about how you terminate commands, expecting that each command is on a line terminated only with a carriage return (\r or ASCII 13). Most serial terminal programs allow you to control what is sent when you hit the Return key, however.

In CoolTerm for OS X and Windows, click the Options button and change the Enter Key Emulation to CR (see Figure 6-10). In PuTTY for Windows and Ubuntu Linux, choose the Terminal Configuration tab, and check "Implicit LF in every CR" (see Figure 6-11).

Once you've configured your serial terminal, open the port and type:

```
+++
```

Don't hit Return or any other key for at least one second afterward. The XBee should respond like so:

```
OK
```

This step is similar to the Bluetooth modem in Chapter 2, where you typed $$$ to enter command mode. The XBee is using an AT-style command set, 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 10 seconds. So, if you're reading this while typing, you may need to type +++ again before the next step.

Choosing Which XBee Radios to Buy

Digi makes several variations on the XBee module, and picking the right one can be confusing. The point-to-multipoint modules are the most basic, featuring the ability to send directed messages or broadcast messages to any radio in the network. They form star networks, as described in Chapter 3, and are the easiest to set up. The mesh modules include the ability to form multi-tier mesh networks, but they are more complex to set up and operate. Robert Faludi's excellent book Building Wireless Sensor Networks (O'Reilly) covers the mesh network radios in depth. For most hobbyist projects, though, a mesh network is overkill, so point-to-multipoint radios are used in this book.

The basic model, the XBee 802.15.4 low-power modules, are the cheapest and have a nominal transmission range of about 300m. The XBee-PRO 802.15.4 extended range models have a longer range of about 1 mile (in practice, I've never gotten more than a quarter-mile), but they consume more power. Both the low-power and the PRO models can be used interchangeably for this book's projects.

The Digi radios have several antenna options. Only two options, the wire antenna or the chip antenna, require no extra parts, so I recommend them for these projects.

Their model numbers are:

Digi XBee 802.15.4 low-power module: XB24-AWI-001 or XB24-ACI-001

Digi XBee-PRO 802.15.4 extended range module: XBP24-AWI-001 or XBP24-ACI-001



Figure 6-9. XBee 802.15.4 modules. XBee low-power module with chip antenna on the left, and XBee-PRO extended range module with wire antenna on the right.
Pictures courtesy of Digi International.

Once you get the OK response, set the XBee's address. The XBee 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 (in computer memory, two or more bytes 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 to do this. Type:

```
ATMY1234\r
```

(Remember that `\r` indicates you should press Enter or Return.) To confirm that you set it, type:

```
ATMY\r
```

The module should respond:

```
1234
```

Next, set the XBee's destination address (the address to which it will send messages). 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
```

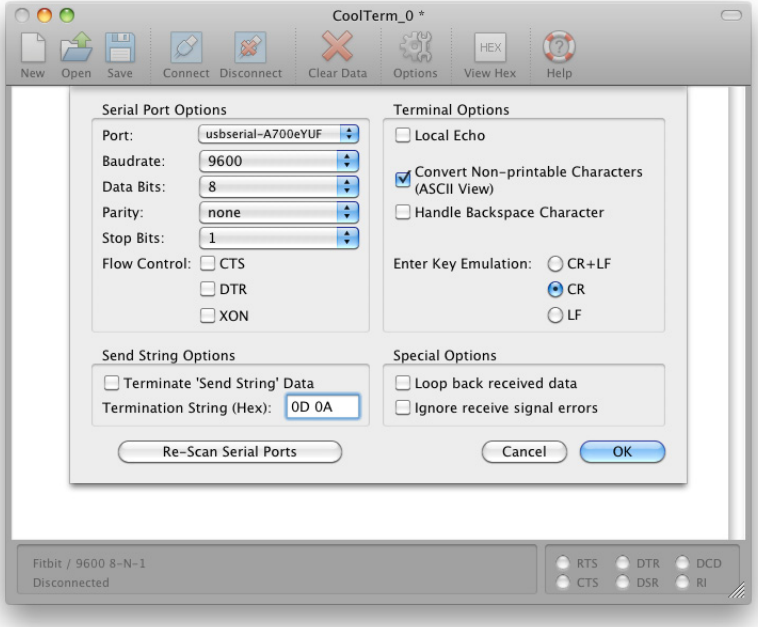


Figure 6-10
CoolTerm options menu. To use CoolTerm to configure Digi radios (XBee and otherwise), set Enter Key Emulation to CR.

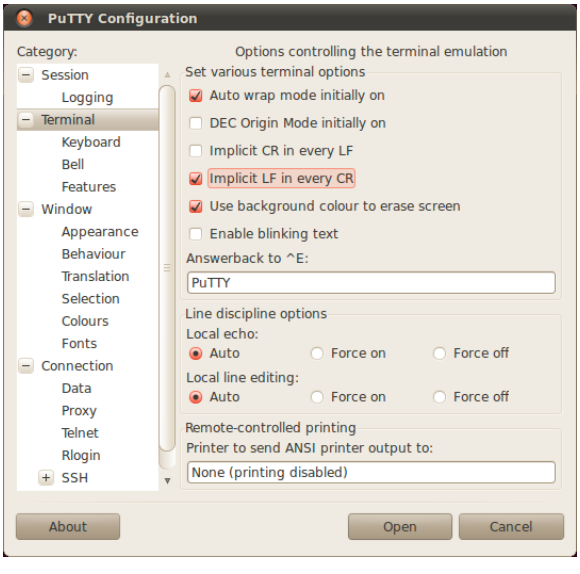


Figure 6-11
PuTTY Terminal Configuration menu. To use PuTTY to configure Digi radios (XBee and otherwise), choose "Implicit LF in every CR."

This pair of commands can also be used to set the destination address:

```
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 WR after your last command, which writes 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
```

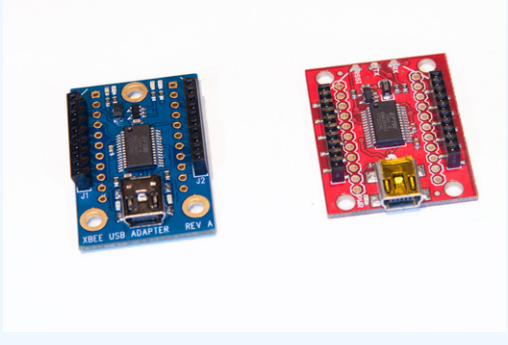
Choosing Which XBee Accessories to Buy

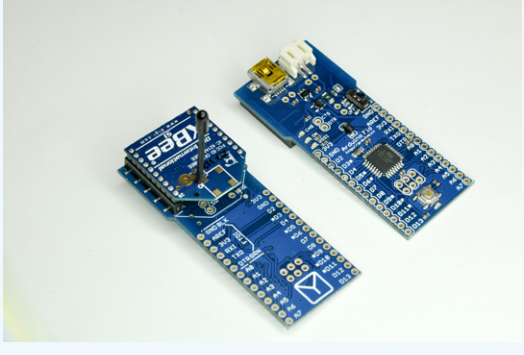
There are so many XBee accessories on the market now that picking the right parts can be confusing. There are basically three ways you'll use XBees in this book's projects:

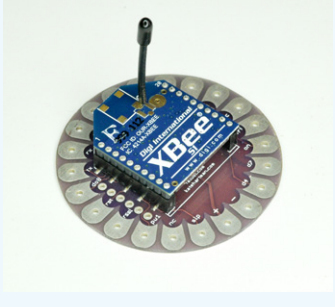
XBee to computer via USB-to-XBee adapter. There are many of these available. Figure 6-12 shows two models, the SparkFun XBee Explorer in red, and the XBee USB adapter board—available from Adafruit and Parallax—in blue.

XBee to microcontroller. For this, you can either use a breakout board on which to mount the XBee, or a wireless shield for a regular Arduino. Or, you can use an Arduno Fio, which has an XBee mount built in.

XBee standalone. For this, XBee breakout boards work well, as does the XBee LilyPad board. The XBee LilyPad, the XBee Explorer Regulated (Spark Fun part no. WRL-09132) and the Adafruit XBee adapter kit (Adafruit part no. 126) all feature built-in voltage regulators, so you can use your XBee with a wider range of power supplies.







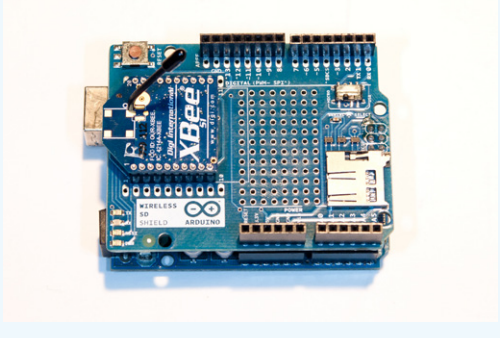


Figure 6-12. XBee accessories. Clockwise from top left: XBee USB adapter, XBee Explorer, Arduino Fio, Arduino wireless shield, XBee LilyPad.

Once you've configured one of your radios, 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:

```
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
```

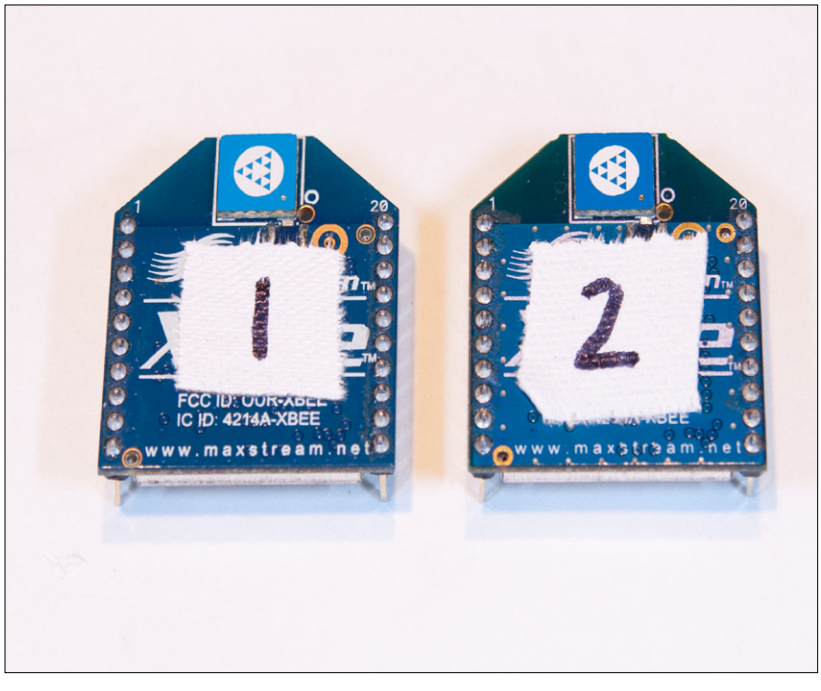
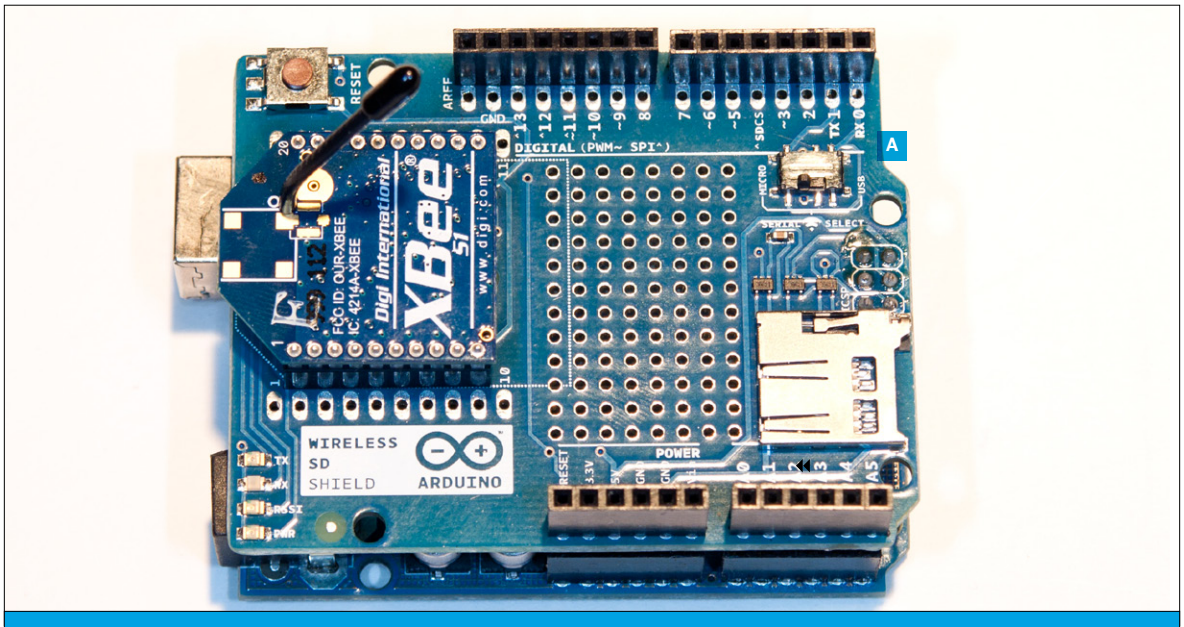


Figure 6-13
To keep track of the radios, label them with a piece of tape. You'll be switching them between the USB-to-Serial adapter and the microcontroller a few times, and it's easy to lose track of which is which.



A. Serial Switch set to Micro (to the left)

Arduino Wireless Shield

Several companies now make wireless shields for Arduino that can work for this project. The original Arduino shield for XBees used in the first edition of this book has been substantially redesigned. Now called the Arduino wireless shield (because other radio devices with the same footprint can also work on this shield), it has a few nice features, such as a prototyping area, an optional microSD card slot (you'll see an SD card example later in the book), and a serial select switch to allow you to change the XBee's serial pin connections.

When the wireless shield's serial select switch is set to "Micro," the XBee will be connected to communicate with the ATMega328 microcontroller on the Arduino. When switched to "USB," it will be connected to communicate directly through the USB-to-Serial processor on the Arduino, bypassing the microcontroller. In this position, you can use the Arduino's USB-to-Serial connection to configure your XBees.

When you're programming the Arduino, it's a good idea to remove the XBee so that the radio's serial communications don't interfere with the program upload.

To configure the XBee radio on the shield using your Arduino board as a USB-to-Serial converter, program the Arduino with a blank sketch, just like this:

```
void setup() {
}
void loop() {
}
```

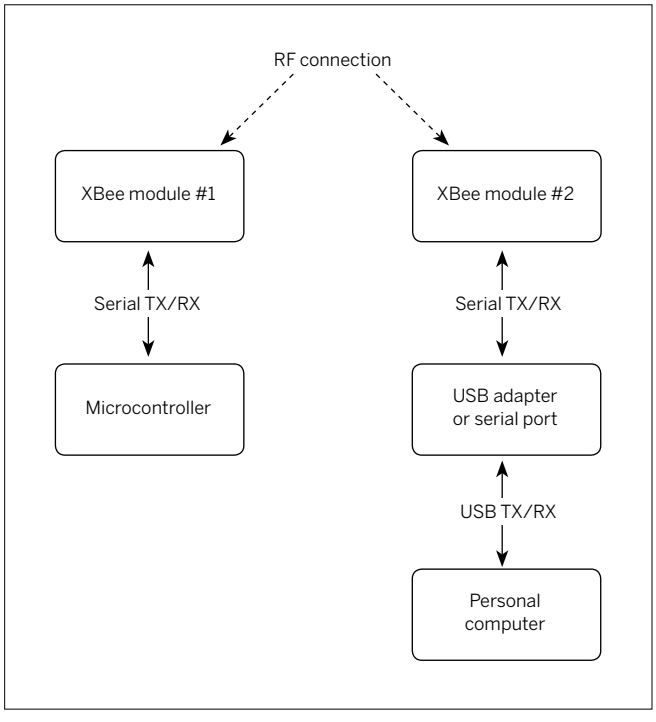
Then switch the serial select switch to USB. Open a serial terminal connection to the Arduino board's serial port, and send commands as shown in "Step 1: Configuring the XBee Modules Serially." Once you've configured the radio, unplug the XBee from the shield, set the serial select switch back to "Micro," and program your Arduino as usual.

Step 2: Programming the Microcontroller to Use the XBee Module

OK! Now you're ready to get two microcontrollers to talk to each other wirelessly via XBee radios. For now, you're going to leave one XBee connected to your computer via the USB adapter. Once you confirm that's working, you'll replace your computer with a second Arduino and XBee. Figure 6-14 shows a diagram of what's connected to what in this step. In Step 3, you'll remove the personal computer and replace it with a second Arduino.

Figure 6-15 shows an XBee module attached to a regular Arduino using the Arduino wireless shield. Figure 6-16 shows how you'd do it if you were using just an XBee breakout board instead of a shield. Note that the XBee is attached to a 3.3V regulator. The XBee's serial I/O connections are 5-volt tolerant, meaning they can accept 5-volt data signals, but the module operates at 3.3 volts. The Arduino Uno and newer models have a more robust 3.3V regulator that can power the XBee, but it doesn't hurt to add a dedicated one, as shown here.

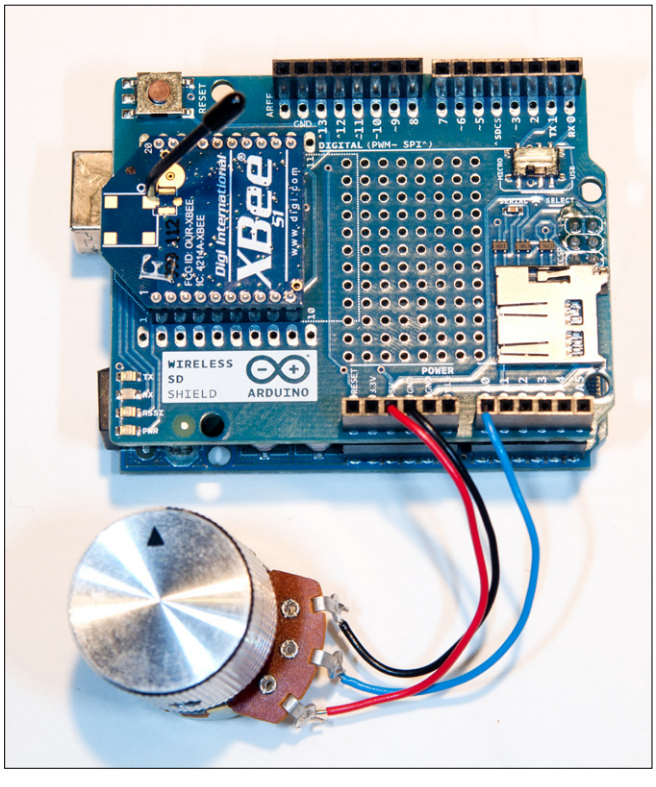
Once your module is connected, it's time to program the microcontroller to send data through the XBee. In this program, the microcontroller will configure the XBee's destination address on startup, and then send an analog reading when the reading on analog pin 0 changes significantly.



At right, above

Figure 6-14

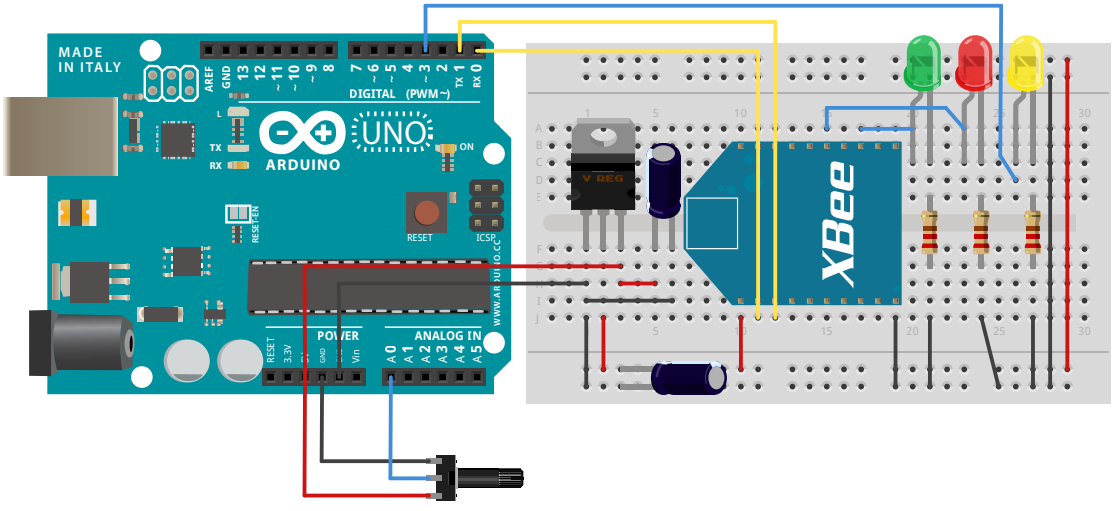
XBee #1 is connected to the microcontroller. XBee #2 is connected via USB or serial to the PC. This enables a wireless link between the PC and the microcontroller.



At right, below

Figure 6-15

Arduino and wireless shield with potentiometer attached to analog pin 0. This circuit is the same as the one shown in Figure 6-16, but it's without the LED on pin 9. The LEDs attached to the XBee are built into the shield.



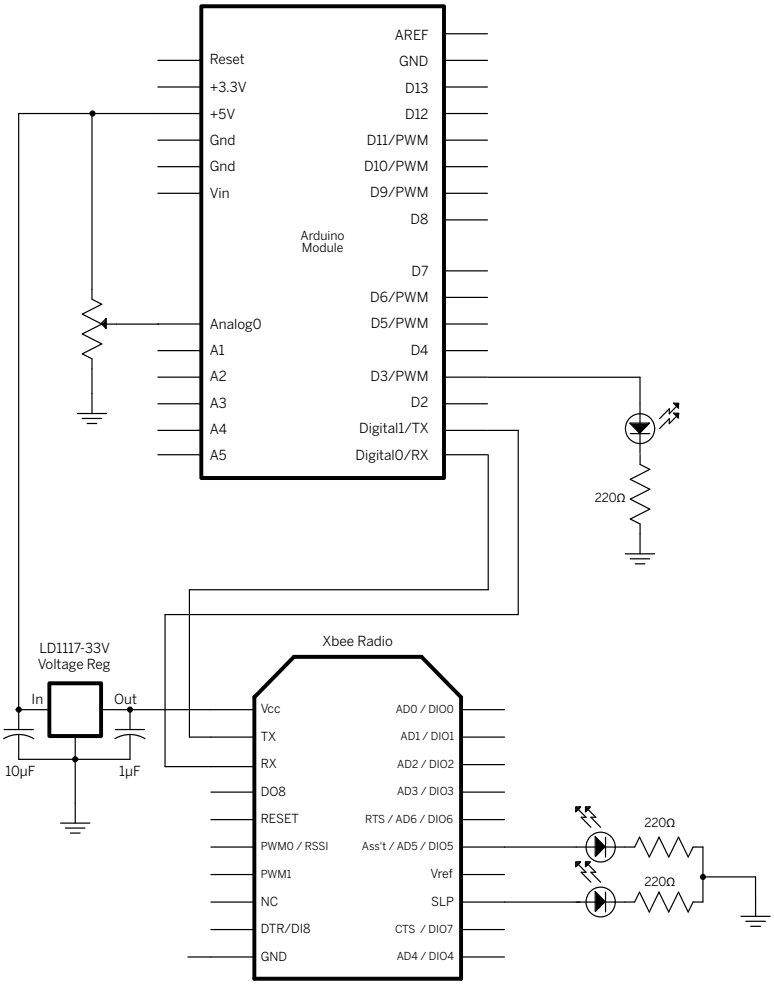


Figure 6-16
Top: XBee connected to an Arduino. If you don't want to use a shield, you can connect the two with just a breakout board. Note that this circuit assumes you have a bare breakout board with no voltage regulator. If you use Spark Fun's XBee Explorer Regulated (part no. WRL-09132) or Adafruit's XBee adapter kit (part no. 126), you won't need the voltage regulator because those two boards have built-in 3.3V regulators.

At left: 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 potentiometer.

```
const int sensorPin = A0;    // input sensor
const int analogLed = 3;     // LED that changes brightness w/incoming value
const int threshold = 10;    // threshold for sensor's change

int lastSensorReading = 0;  // previous state of the sensor

String inputString = "";
```

» Next, in the setup() method, configure serial transmission, set the modes on the I/O pin, and configure the XBee's destination address.

```
void setup() {
  // configure serial communications:
  Serial.begin(9600);

  // configure output pin:
  pinMode (analogLed, OUTPUT);

  // set XBee's destination address:
  setDestination();
  // blink the TX LED indicating that the main program's about to start:
  blink(analogLed, 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, using 16-bit addressing.
  // if you're using two radios, one radio's destination
  // should be the other radio's MY address, and vice versa:
  Serial.print("ATDH0, DL5678\r");
  // set my address using 16-bit addressing:
  Serial.print("ATMY1234\r");
  // set the PAN ID. If you're working in a place where many people
  // are using XBees, you should set your own PAN ID distinct
  // from other projects.
  Serial.print("ATID1111\r");
  // put the radio in data mode:
  Serial.print("ATCN\r");
}
```

» Change the destination address to that 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 this book. It blinks an LED to indicate that setup is over.

```
void blink(int thisPin, int howManyTimes) {
  //     Blink the LED:
  for (int blinks=0; blinks< howManyTimes; blinks++) {
    digitalWrite(thisPin, HIGH);
    delay(200);
    digitalWrite(thisPin, LOW);
    delay(200);
  }
}
```

The main loop handles incoming serial data, reads the potentiometer, and sends out da if there's a sufficient change in the potentiometer's reading.

```
void loop() {
  // listen for incoming serial data:
  if (Serial.available() > 0) {
     handleSerial();
  }

  // listen to the potentiometer:
  int sensorValue = readSensor();

  // if there's something to send, send it:
  if (sensorValue > 0) {
    Serial.println(sensorValue, DEC);
  }
}
```

Two other methods are 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() {
  char inByte = Serial.read();

   // save only ASCII numeric characters (ASCII 0 - 9):
  if (isDigit(inByte)){
    inputString = inputString + inByte;
  }

  // if you get an ASCII newline:
  if (inByte == '\n') {
    // convert the string to a number:
      int brightness = inputString.toInt();
   // set the analog output LED:
    analogWrite(analogLed, brightness);
    // clear the input string for the
    // next value:
    inputString = "";
    Serial.print(brightness);
  }
}

int readSensor() {
  int result = analogRead(sensorPin);
```

NOTE: You should disconnect the XBee's receive and transmit connections to the microcontroller while programming (if you're using the Arduino wireless shield, use the serial select switch). 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 previous page.

```
  // look for a change from the last reading
  // that's greater than the threshold:
  if (abs(result - lastSensorReading) > threshold) {
    lastSensorReading = result;
    result = result/4; // divide by 4 to fit in one byte
  } else {
   // if the change isn't significant, return 0:
  result = 0;
  }
  return result;
}
```

In the main loop, notice that 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, as well as almost any device that implements this kind of 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 might overwrite itself in the serial window—depending on your serial terminal application—because you're not sending a newline character. 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 replace the computer in the previous step with a second microcontroller. Connect an Arduino to your second XBee module, as shown in Figure 6-16, or use a wireless shield. 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 3.

First, connect the second XBee module to the second microcontroller. Then, program both microcontrollers with the previous program, making sure to set the destination addresses as noted in the program.

When you've programmed both modules, power them on and turn the potentiometer several times. As you turn the potentiometer, the LED on pin 3 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 possibilitiesfor 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 power it from a battery. There's a simple way to do this for the standard Arduino boards, and there are a few models of Arduinos—and some derivative models—that are designed for mobile battery-powered use.

The simplest option is to connect a battery to the power input terminals, as shown in Figure 6-17. The Vin pin (Voltage input) can take from 6-15V input (the Arduino Uno will run off lower voltage—I've run one off 3.7V—but it's not always reliable). You can either plug into the ground and Vin pins, or make a plug adapter and plug into the power plug.

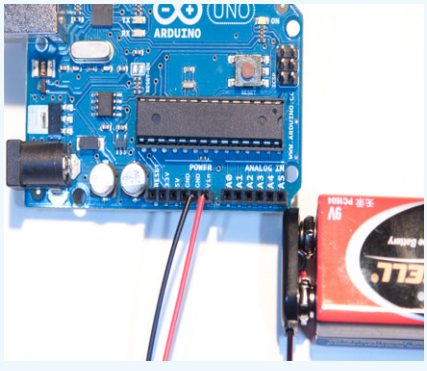


Figure 6-17
Arduino module powered by a 9V battery.

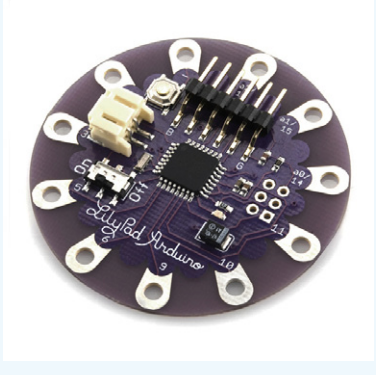


Figure 6-19
LilyPad Arduino Simple, with the LiPo battery connector. *Photo courtesy of Spark Fun.*

The Arduino Fio is great for XBee projects. It has a socket for an XBee and a battery connector for 3.7V Lithium Polymer batteries, as shown in Figure 6-18. The mini-USB jack on the Fio doesn't actually communicate with the microcontroller—it just charges the battery. To program the Fio, you either need an FTDI-style USB adapter, as described in Chapter 2, or you can program it wirelessly over the XBee. Programming the Fio wirelessly requires an understanding of XBees, so it's a good idea to program it using a wire first. For more on Fio programming, see http://arduino.cc/en/Main/ArduinoBoardFioProgramming.

The LilyPad Arduinos are all made for use in clothing and soft goods. They're also programmed using an FTDI-style serial adapter. There are a couple LilyPad power adapters that take LiPo batteries. The LilyPad Arduino Simple has a jack so you can add a LiPo battery right to the board (see Figure 6-19).

It's a good idea to keep your microcontroller module connected to a power adapter or USB power while programming and debugging. When a battery starts to weaken, your module will operate inconsistently, which can make debugging impossible.

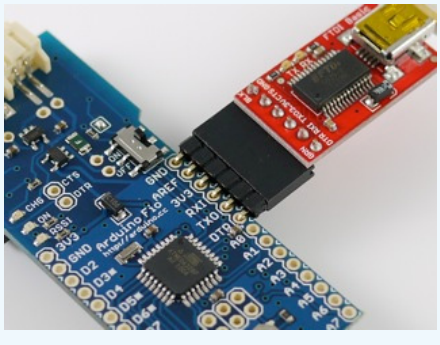


Figure 6-18
Detail of the Fio, showing battery connector and USB-to-Serial adapter connected for programming.

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 out a signal with no awareness of whether the receiver got the message, and it could send to a different receiver just by changing the destination address. In contrast, Bluetooth radios must establish a connection before sending data over a given channel, and they 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 Bluetooth Mate radios from Spark Fun—use a radio from Roving Networks. The command set used here was defined by Roving Networks. Other Bluetooth modules from other manufacturers use command sets with a similar style, and they may execute similar functions, but their syntax is not the same. Unfortunately, Bluetooth radio manufacturers haven't set a standard syntax for their devices.

MATERIALS

- **2 solderless breadboards**
- **1 USB-to-TTL serial adapter**
- **2 Arduino modules**
- **2 Bluetooth Mate modules**
- **2 potentiometers** or other analog sensors
- **2 LEDs**
- **2 220-ohm resistors**
- **2 10-kilohm resistors**
- **2 pushbuttons**

Step 1: The Circuits

Instead of pairing a Bluetooth radio with your computer's Bluetooth radio as you did in Chapter 2, you're going to pair two radios attached to microcontrollers. When you're done, your computer won't be needed.

Because the Bluetooth connection process involves many steps, it's easiest to learn and understand it using a serial terminal program before you start to write code. Even when you are programming, the serial terminal program will be a useful diagnostic tool. You can use the USB-to-serial modules shown for Project 4, Negotiating in Bluetooth in Chapter 2, but the goal of this project is to get two microcontrollers talking to each other over Bluetooth with no personal computer in the middle. So, instead of using the USB-to-Serial adapter, you can set up an Arduino board to pass the serial data from your computer to a Bluetooth radio. Then, when you're ready to remove the personal computer in Step 3, you'll just have to change the sketch and make a minor circuit change to remove the PC.

To set up an Arduino as a USB-to-Serial passthrough, first program the Arduino with a blank sketch. The BareMinimum sketch from the Basics examples will do fine. It looks like this:

```
void setup() {
}
void loop() {
}
```

When you upload this sketch, the microcontroller will do nothing, so you can use its connection to the USB-to-serial processor to communicate with the Bluetooth radio. Connect the radio as shown on the left (we'll call it Arduino #1 and radio #1) in Figure 6-20; then, connect it to your computer and using your serial terminal program, open a serial connection to it at 115200 bits per second.

For the moment, you'll use the second Arduino just to send a simple message so you can see things are working. Connect it to the second Bluetooth Mate using the circuit shown at right in Figure 6-20 (we'll call it Arduino #2 and radio #2). Eventually, they will both be wired this way.

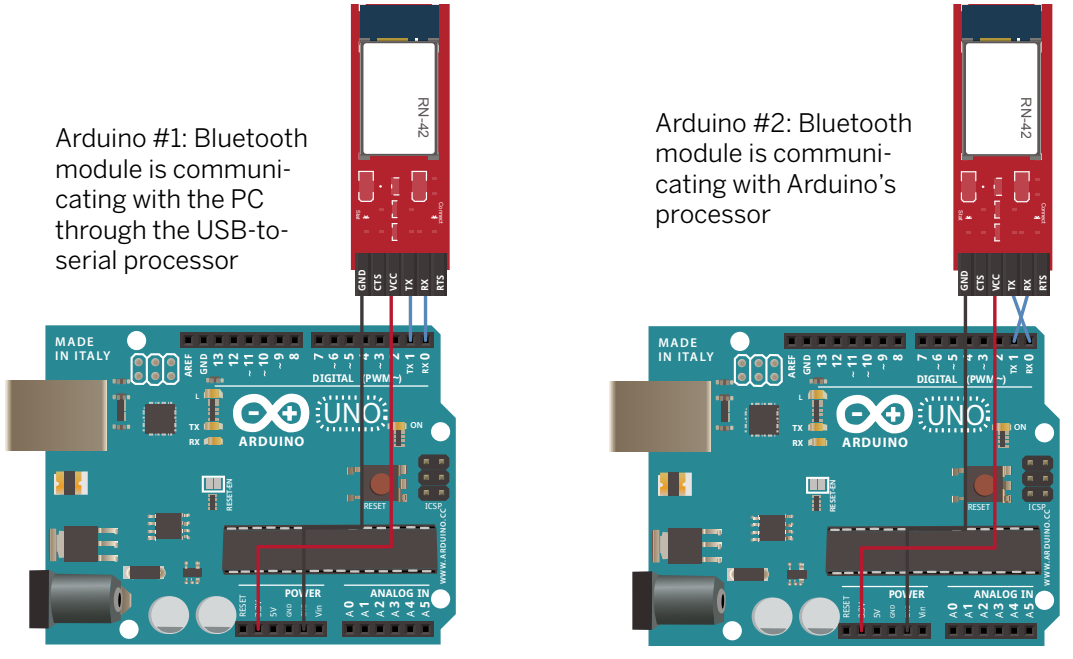


Figure 6-20
The Arduino module on the left is used as a USB-to-Serial adapter for a Bluetooth Mate module. Notice how RX is attached to RX and TX to TX. That's because the Bluetooth Mate is actually talking to the PC through the USB-to-Serial adapter, not the Arduino's main processor, so the serial connections are reversed. When you want the Bluetooth module to talk to the Arduino's main processor, you'll swap these two connections, as shown at right.

Program Arduino #2 using a basic serial sketch, as follows:

```
void setup() {
   Serial.begin(115200);
}

void loop() {
   Serial.println("Hello Bluetooth!");
}
```

When you open a Bluetooth connection to this radio, you'll see the following message over and over:

```
Hello Bluetooth!
Hello Bluetooth!
Hello Bluetooth!
```

Step 2: Getting to Know the Commands

The Bluetooth Mate radios use a serial command set for command and configuration, which has two modes—command mode and data mode—just like the XBee radios. When you first power up a Bluetooth Mate and connect to its serial interface (using Arduino #1 in Figure 6-20, for example), it's in data mode. To see that it's alive, type: `$$$`. It will respond:

```
CMD\r
```

All of the radio's responses will be followed by a carriage return, as shown here. All of your input commands should be followed by a carriage return (press Enter or Return).

Once you're in command mode, you can get some basic information about the radio by typing D, just as you did in Chapter 2. You'll get the radio's status, which includes its address:

```
***Settings***

BTA=000666112233
BTName=FireFly-7256
Baudrt(SW4)=115K
Parity=None
Mode  =Slav
Authen=0
Encryp=0
PinCod=1234
Bonded=0
Rem=NONE SET
```

The first line beginning with BTA= is the radio's address in hexadecimal notation. 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 GK\r. It will respond like so:

```
0
```

When it's connected and you do this, it will respond with 1 instead.

Now you want to see which other radios are available. Type the following (the capital letter i):

```
I\r
```

The radio will respond with a list of radios, as it did in Chapter 2:

```
Found 2
442A60F61837,Tom Igoe...s MacBook Air,38010C
000666481ADF,RN42-1FDF,1F00
Inquiry Done
```

You can see that each device has a different address, name, and device code. You can get a list of only the ones with a particular address code by typing:

```
IN 0,001F00\r
```

This is handy when you want only the other Bluetooth Mates in the area. As you might have guessed, the device code is 001F00. The IN command also eliminates the text names, so you can get just the addresses and device codes.

Now that you have the address of your other Bluetooth radio (which should have shown in the lists above, if you have it plugged into the other Arduino), you can connect to it like so:

```
C, 000666481ADF\r
```

Replace 000666481ADF with the address of the radio attached to Arduino #2. Radio #1 will respond:

```
TRYING
```

Once it makes a good connection, the green connect light on both Bluetooth Mates should come on, the radios will shift automatically to data mode, and you should see the message from Arduino #2 as follows:

```
Hello Bluetooth!
Hello Bluetooth!
Hello Bluetooth!
```

The connection goes both ways, of course. Anything you type in the serial terminal window gets sent to Arduino #2. It's not programmed to respond, though, so you won't see anything come back from it. When you're ready to close the connection, you first have to get back into command mode by typing:

```
$$$\r
```

The radio will give you a CMD again, after which you type:

```
K,
```

The connection will be broken, and you'll see:

```
KILL
```

There are other status commands as well, but these ones are most important at first.

X

Step 3: Connecting Two Bluetooth Radios

Now that you've gotthe basics of connecting and disconnecting, it's time to get the microcontrollers to do it. For this step, you'll connect via the same physical setup, adding in a few extra parts. You'll work on Arduino #2 instead of Arduino #1. For now, leave the latter wired up as a USB-to-Serial converter for the Bluetooth radio. Also leave the serial terminal to it open, so you can see what happens.

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. While you're getting the addresses, make one extra configuration on both radios, as follows:

```
SO,BT\r
```

This sets the status string to BT, so that when the radios connect or disconnect, they'll send a string to let you know, like this:

```
BTCONNECT\r
```

or:

```
BTDISCONNECT\r
```

Next, add a potentiometer and pushbutton, as shown in Figure 6-21. Just like the XBee example, it's got a potentiometer attached to the analog pin so that you can send its values. You can use any analog sensor on pin A0. The pushbutton will make or break the connection between the two radios. This is a big difference between the XBees and the Bluetooth radios—the latter have to be paired before they can communicate.

NOTE: It's a good idea to to remove your Bluetooth Mate while programming the Arduino boards, just as you've had to for other serial devices.

When the pushbutton is pressed for the first time, the following program connects to another Bluetooth Mate with a set address. When it connects, it sends its potentiometer value as an ASCII string, terminated by a carriage return.

Just like the XBee example, this program also looks for incoming ASCII strings and converts them to use as a PWM value to dim an LED on pin 3. It will take advantage of the TextFinder library that you saw in Chapter 4. You can also modify this sketch to use the Stream class' text parsing methods. For alternate versions, see the GitHub repository for this book's code, at https://github.com/tigoe/MakingThingsTalk2.

First, here are the constants and variables for this program.

Change this to the address of the other radio.

```
/*
  Bluetooth Analog Duplex sender
 Context: Arduino
 */
#include <TextFinder.h>

const int sensorPin = A0;          // analog input sensor
const int analogLed = 3;           // LED that changes brightness
const int threshold = 20;          // threshold for sensor's change
const int debounceInterval = 15;   // used to smooth out pushbutton readings
const int connectButton = 2;       // the pushbutton for connecting
int lastButtonState = 0;           // previous state of the pushbutton
int lastSensorReading = 0;         // previous state of the sensor
long lastReadingTime = 0;          // previous time you read the sensor

// address of the remote BT radio. Replace with the address
// of your remote radio
String remoteAddress = "112233445566";
String messageString = "";         // messages coming in serial port

boolean connected = false;         // whether you're connected or not
boolean commandMode = false;       // whether you're in command or data mode

TextFinder finder(Serial);         // for searching the serial input
```

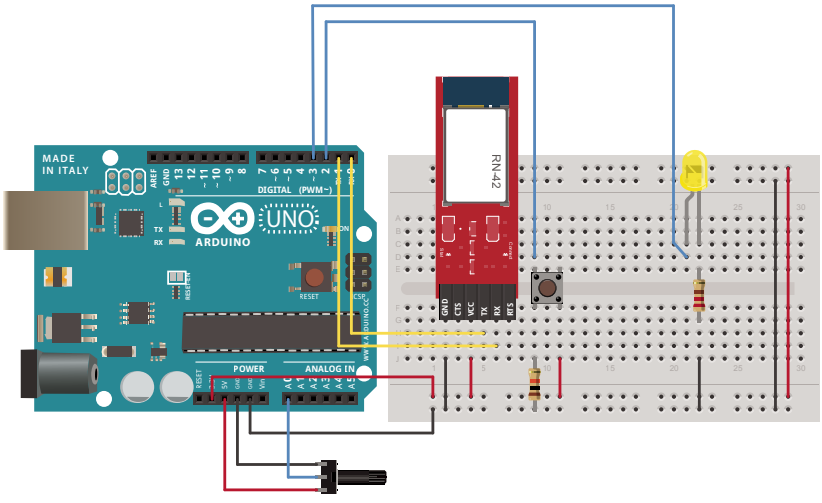
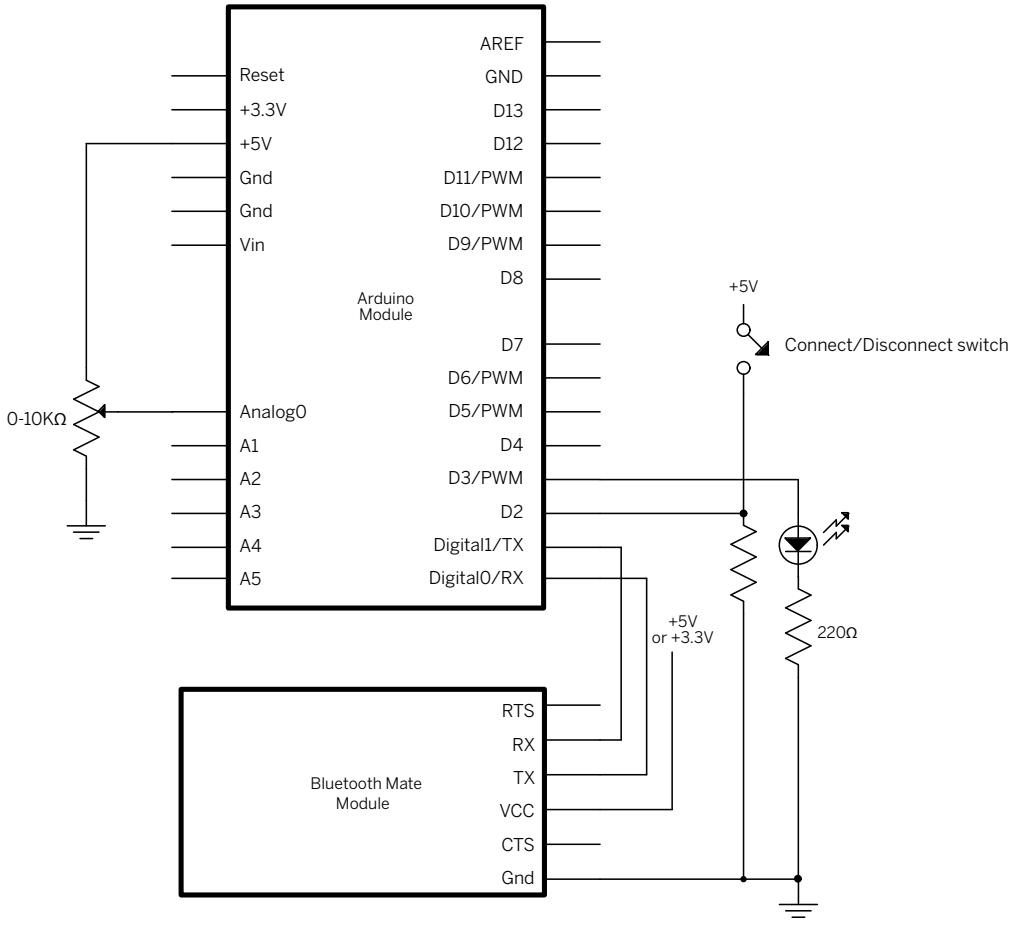


Figure 6-21
Bluetooth Mate radio attached to an Arduino. This circuit is similar to the XBee microcontroller circuit discussed earlier. The pushbutton will connect or disconnect the two radios.



The setup() method sets the states of the pins, initializes serial, and blinks an LED, as usual.

```
void setup() {
  // configure serial communications:
  Serial.begin(115200);

  // configure output pins:
  pinMode (analogLed, OUTPUT);

  // blink the TX LED indicating that
  // the main program's about to start:
  blink(analogLed, 3);
}
```

The main loop listens for incoming serial data and checks to see whether the button's been pushed. If so, it connects or disconnects as appropriate. If the radios are connected, it reads the analog input and sends it out if there's been a significant change.

```
void loop() {
  // read incoming serial and parse it:
  handleSerial();

  // check to see if the pushbutton's pressed:
  boolean buttonPushed = buttonRead(connectButton);

  // if the button's just pressed:
  if (buttonPushed) {
    // if the client's connected, disconnect:
    if (connected) {
      BTDisconnect();
    } // if the client's disconnected, try to connect:
    else {
      BTConnect();
    }
  }
 // if connected, take sensor readings:
  if (connected) {
    // note the current time in milliseconds:
    long currentTime = millis();
    // if enough time has passed since the last reading:
    if (currentTime - lastReadingTime > debounceInterval) {
      // read the analog sensor, divide by 4 to get a 0-255 range:
      int sensorValue = analogRead(A0)/4;
      // if there's a significant difference between the
      // current sensor reading and the last, send it out:
      if (abs(sensorValue - lastSensorReading) > threshold) {
        Serial.println(sensorValue, DEC);
      }
      // update the last reading time
      // and last sensor reading:
      lastReadingTime = currentTime;
      lastSensorReading = sensorValue;
    }
  }
}
```

» The blink() method is the same as it was in the earlier XBee example.

```
void blink(int thisPin, int howManyTimes) {
  for (int i=0; i< howManyTimes; i++) {
    digitalWrite(thisPin, HIGH);
    delay(200);
    digitalWrite(thisPin, LOW);
    delay(200);
  }
}
```

» The buttonRead() method will look familiar, too,—it's the same one from the pong clients in Chapter 5.

```
// this method reads the button to see if it's just changed
// from low to high, and debounces the button in case of
// electrical noise:

boolean buttonRead(int thisButton) {
  boolean result = false;
  // temporary state of the button:
  int currentState = digitalRead(thisButton);
  // final state of the button:
  int buttonState = lastButtonState;
  // get the current time to time the debounce interval:
  long lastDebounceTime = millis();

  while ((millis() - lastDebounceTime) < debounceInterval) {
    // read the state of the switch into a local variable:
    currentState = digitalRead(thisButton);

    // If the pushbutton changed due to noise:
    if (currentState != buttonState) {
      // reset the debouncing timer
      lastDebounceTime = millis();
    }

    // whatever the reading is at, it's been there for longer
    // than the debounce delay, so take it as the actual current state:
    buttonState = currentState;
  }
  // if the button's changed and it's high:
  if(buttonState != lastButtonState && buttonState == HIGH) {
    result = true;
  }

  // save the current state for next time:
  lastButtonState = buttonState;
  return result;
}
```

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 Bluetooth Mates send a serial message before dropping into data mode, thanks to your last configuration change. The serial message looks like this:

```
BTCONNECT\r
```

When the connection's broken, it sends this message and stays in command mode:

```
BTDISCONNECT\r
```

While trying to connect, the Mate sends:

```
TRYING\r
```

You can ignore the TRYING message because it'll always be followed by CONNECT or—if the Mate doesn't succeed in connecting—it sends:

```
CONNECT failed\r
```

Look for these to come in and use them to track the connection status. The Bluetooth Mates drop into data mode immediately after sending either of these two messages, so they're a handy way to keep track of the mode as well. The BTConnect(), BTDisconnect(), and handleSerial() methods use these strings to do their work, as follows.

» BTConnect() checks the command/data mode status, then tries to make a connection. If the attempt fails, it stays in command mode.

```
void BTConnect() {
  // if in data mode, send $$$
  if (!commandMode) {
    Serial.print("$$$");
    // wait for a response:
    if (finder.find("CMD")) {
      commandMode = true;
    }
  }
  // once you're in command mode, send the connect command:
  if (commandMode) {
    Serial.print("C," + remoteAddress + "\r");
    // wait for a response:
    finder.find("CONNECT");
    // if the message is "CONNECT failed":
    if (finder.find("failed")) {
      connected = false;
    }
    else {
      connected = true;
      // radio automatically drops into data mode
      // when it connects:
      commandMode = false;
    }
  }
}
```

BTDisconnect() is similar to BTConnect(), but in reverse. It drops into command mode and sends the disconnect message you used earlier.

```
void BTDisconnect() {
  // if in data mode, send $$$
  if (!commandMode) {
    Serial.print("$$$");
    // wait for a response:
    if (finder.find("CMD")) {
      commandMode = true;
    }
  }
  // once you're in command mode,
  // send the disconnect command:
  if (commandMode) {
    // attempt to connect
    Serial.print("K,\r");
    // wait for a successful disconnect message:
    if (finder.find("BTDISCONNECT")) {
      connected = false;
      // radio automatically drops into data mode
      // when it disconnects:
      commandMode = false;
    }
  }
}
```

handleSerial() also looks for text messages, but it doesn't use TextFinder because it needs to be able to see one of three different options. TextFinder makes only one pass through the serial stream, so you can't check for a second string if it doesn't find the first string.

If a valid number is found, this method uses it to set the brightness of the LED on pin 3.

```
void handleSerial() {
  // look for message string
  // if it's BTCONNECT, connected = true;
  // if it's BTDISCONNECT, connected = false;
  // if it's CONNECT failed, connected = false;
  // if it's a number, set the LED
  char inByte = Serial.read();

  // add any ASCII alphanumeric characters
  // to the message string:
  if (isAscii(inByte)) {
    messageString = messageString + inByte;
  }

  // handle CONNECT and DISCONNECT messages:
  if (messageString == "BTDISCONNECT") {
    connected = false;
  }
  if (messageString == "BTCONNECT") {
    connected = true;
  }

  if (connected) {
    // convert the string to a number:
    int brightness = messageString.toInt();
    // set the analog output LED:
```

Finally, handleSerial() looks for a carriage return; when it finds one, it clears the messageString in order to get the next message.

Continued from previous page.

```
    if (brightness > 0) {
      analogWrite(analogLed, brightness);
    }
  }

  // if you get an ASCII carriage return:
  if (inByte == '\r') {
    // clear the input string for the
    // next value:
    messageString = "";
  }
}
```

That's the whole program. Run this on your microcontroller, filling in the address of the radio on Arduino #2 for the address in the remoteAddress array above. When you press the button on the first microcontroller, it will make a connection to the second and start sending sensor values through. In the serial terminal, your initial messages should look like this:

```
BTCONNECT
121
132
83
```

When you're connected, you can respond as if you were sending your own sensor values, and the microcontroller will fade the LED on pin 3 accordingly. Type:

```
12\r
120\r
255\r
1*
```

The LED should start dim, get brighter, then brightest, then get very dim. You should get four sensor readings in response. Then, when you press the button again, you should get:

```
BTDISCONNECT
```

When you've connected and disconnected a few times, you're ready for the final step.

Step 4: Connecting Two Microcontrollers via Bluetooth

If you've been noticing the parallels between the XBee example and this one, you probably know what's coming. Build the same circuit for your second microcontroller, using the second radio by adding the switch and potentiometer, and swapping the serial TX and RX connections. Then change the Bluetooth address in the sketch above to be the address of the first radio, and program the second microcontroller. Then reset both microcontrollers. When you press the button on either one, it will attempt to connect to the other and begin exchanging data. When you press the button on either one a second time, they'll disconnect.

Now that you've got the basics down, you can modify this to handle a variety of situations with Bluetooth radios.
X

Buying Radios

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

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

Consider the data rate you need for your application—more specifically, for the wireless part of it. You may not need high-speed wireless. One common use for wireless communication in the performance world is to get data off the bodies of performers, in order to control MIDI performance devices like samplers and lighting dimmers. You might think that you need your radios to work at MIDI data rates to do this, but you don't. You can send the sensor data from the performers wirelessly at a low data rate to a stationary microcontroller, then have the microcontroller send the data via MIDI at a higher data rate.

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

X

What About WiFi?

So far, you've seen the most basic serial radios in action in the transmitter-receiver project, and more advanced radios at work in the transceiver projects. If you're thinking about networks of microcontrollers, you're probably wondering whether you can connect your projects to the Internet and to each other using WiFi. You can, but there are complications to consider.

Until recently, WiFi wasn't very common in microcontroller projects for a couple reasons: cost and power. Microcontroller-to-WiFi modules on the market are more expensive than equivalent transceivers implementing other protocols. That's starting to change. Many earlier WiFi modules were also power hungry, but that too is changing.

There are currently a few WiFi solutions on the market. Spark Fun makes the WiFly shield, and Digi just announced WiFi versions of their XBee modules. There's also a forthcoming Arduino WiFi shield, which is designed to be easy to incorporate into an existing Ethernet project with only minor code changes. A short introduction follows.

X

Project 12

Hello WiFi!

Remember the daylight color server you built in Project 6? In this project, you'll rebuild it using an Arduino WiFi shield. You'll see that most of the code and the circuit is exactly the same. It's only the physical communications layer that changes.

MATERIALS

- **1 Arduino WiFi shield**
- **1 Arduino microcontroller module**
- **1 WiFi Ethernet connection to the Internet**
- **3 10-kilohm resistors**
- **3 photocells (light-dependent resistors)**
- **1 solderless breadboard**
- **3 lighting filters**

Making the Connections

The WiFi shield communicates with the Arduino via SPI just like the Ethernet shield, so build the circuit just as you did in Figure 4-4, but replace the Ethernet shield with a WiFi shield.

NOTE: The WiFi shield can't work with the Arduino Ethernet or Ethernet shield because all three use the same SPI chip select pin.

To make the network connection, you'll need to know the name of the WiFi network you're connecting to (also called SSID), and what type of security it uses. This is the same information you use to connect other wireless devices to your WiFi router. The WiFi shield can connect to open networks or networks secured with WEP (both 40-bit and 128-bit), WPA, or WPA2 encryption. For WPA and WPA2, you'll need the password. For WEP, you'll need the key and the key index. A WEP key is a long string of hexadecimal digits that is used like a password. 40-bit WEP keys are 10 ASCII characters long, and 128-bit WEP keys are 26 characters long. WEP routers can store up to four keys, so the key index indicates which one you're using. Most of the time, you'll use key index 0. Below are some typical examples of WEP and WPA combinations:

WPA network name: noodleNet
WPA password: m30ws3rs!

WEP network name: sandbox
WEP key index: 0
WEP 40-bit key: 1234567890

WEP network name: sandbox
WEP key index: 0
WEP 128-bit key: 1A2B3C4D5E6FDADADEEDFACE10

If you're using a home router, chances are you or a family member set up the wireless router so you know the information. If you're connecting to a school or institutional router, ask your network administrator for the details. Once you have this information, you're ready to start programming.

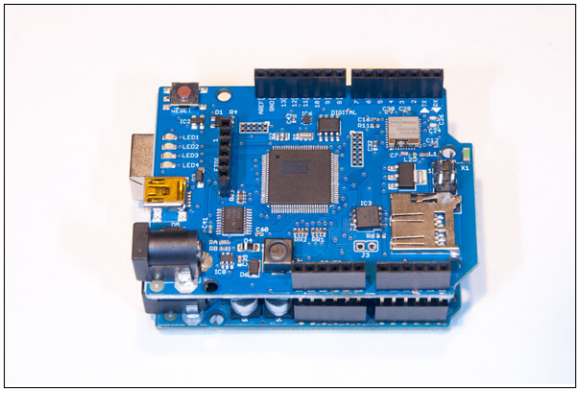


Figure 6-22
Arduino WiFi shield.

The Arduino WiFi shield, shown in Figure 6-22, is a new product, so its programming interface is subject to change as it develops. For the latest updates on the WiFi shield, the WiFi library, and examples of how to use it, see the Hardware section at http://arduino.cc.

Configure It (WPA)

The first thing you need to do is take the network info as described above and put it into variables so you can connect. You'll still need the server and lineLength global variables from the previous RGB server sketch as well.

```
/*
  WiFi RGB Web  Server
  Context: Arduino
 */

#include <SPI.h>
#include <WiFi.h>

char ssid[] = "myNetwork";        // the name of your network
char password[] = "secretpassword"; // the password you're using to connect
int status = WL_IDLE_STATUS;      // the WiFi radio's status

WiFiServer server(80);
int lineLength = 0;    // length of the incoming text line
```

» Change these to match your own network.

Configure It (WEP)

If you're using WPA encryption, your configuration variables will look more like the code at right (changes are shown in blue).

```
char ssid[] = "myNetwork";        // the name of your network
char keyIndex = 0;                // WEP networks can have multiple keys.
// the 128-bit WEP key you're using to connect:
char key[] = "FACEDEEDDADA01234567890ABC";
int status = WL_IDLE_STATUS;      // the WiFi radio's status
```

» Change these to match your own network. If your router has values for more than one key index, use key index 0.

» The setup() looks different from the previous sketch because you're setting up a WiFi connection instead of a wired Ethernet connection. If you get a connection, print out the network name. If you don't, stop the program right there. The WiFi shield uses DHCP by default, so you don't need to do anything in your code to get an address via DHCP.

The remainder of the sketch is identical to the sketch in Project 6. You can copy the rest of the code from there. Because WiFi is also Ethernet, the Client and Server library interfaces are the same, so you can change between Ethernet and WiFi easily.

```
void setup() {
  // initialize serial:
  Serial.begin(9600);

  Serial.println("Attempting to connect to network...");
  // attempt to connect using WPA encryption:
  status = WiFi.begin(ssid, password);

  // or use this to attempt to connect using WEP 128-bit encryption:
  // status = WiFi.begin(ssid, keyIndex, key);

  Serial.print("SSID: ");
  Serial.println(ssid);

  // if you're not connected, stop here:
  if ( status != WL_CONNECTED) {
    Serial.println("Couldn't get a WiFi connection");
    while(true);
  }
}
```

WiFi Diagnostics

The WiFi shield gives you the ability to make networked projects wireless, but it also gives you a few other useful features for diagnosing the health of your connection. Before you begin a WiFi project, run a few diagnostics to make sure you have a good connection. Like any wireless connection, it's invisible, and it's easy to get lost in trouble-shooting something else when the problem is just that you're not connected.

No matter what WiFi module you're working with, you're likely to run into some troubles when trying to connect your microcontroller. Here are a few common ones to look out for:

If you're working at a school or business where you don't control the WiFi routers, make sure you have all the configuration information in advance, and that it matches what your module can do. Enterprise protocols like WPA2 Enterprise are not available for microcontroller-based modules.

Some public networks use a captive portal, which is an open WiFi network, but you have to sign in through a web page before you can access the wider Internet. These are difficult to handle, because you have to program the microcontroller (not the WiFi module) to make an HTTP call to the captive portal with the login information first. If you can use a network that doesn't use a captive portal, you're better off.

Some WiFi modules, like the Arduino WiFi shield, can't see networks where the SSID is hidden. When possible, make sure your network is publicly visible—even when security is on.

Since you have no interface to get feedback on errors, compare your configuration with that of your laptop, mobile phone, or other device that you can connect successfully.

Here are two quick diagnostic code snippets for the Arduino WiFi shield that may be useful as well: scanning for available networks, and getting the signal strength of the network to which you're attached.

Here's how to scan for available networks:

```
// scan for nearby networks:
byte numSsid = WiFi.scanNetworks();

// print the list of networks seen:
Serial.print("SSID List:");
Serial.println(numSsid);
// print the network number and name
// for each network found:
for (int thisNet = 0; thisNet<numSsid; thisNet++) {
  Serial.print(thisNet);
  Serial.print(") Network: ");
  Serial.println(WiFi.SSID(thisNet));
}
```

Here's how to get the signal strength once you're attached to a network:

```
// print the received signal strength:
 long rssi = WiFi.RSSI();
 Serial.print("RSSI:");
 Serial.println(rssi);
```

Both of these techniques are handy diagnostic tools. Depending on your environment and network setup, your wireless network may not be stable all the time. Likewise, you may want to wrap your whole main loop() in an if statement that checks whether you're still connected to the network, like so:

```
void loop() {
  if ( status == WL_CONNECTED) {
    // do everything here
  } else {
    // let the user know there's no connection
  }
}
```

X

Conclusion

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

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

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

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


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



7

MAKE: PROJECTS

Sessionless Networks

So far, the network connections you've seen in this book have mostly been dedicated connections between two objects. Serial communications involve the control of a serial port; mail, web, and telnet connections involve a network port. In all these cases, there's a device that makes the port available (generally a server), and something that requests access to the port (a client). Project 8, Networked Pong, in Chapter 5 was a classic example of this—in that application, the server handled all the communications between the other devices. In this chapter, you'll learn how to make multiple devices on a network talk to each other directly, or talk to all the other devices at once.

Perform-o-shoes by Andrew Schneider

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

Supplies for Chapter 7

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **P** Pololu (www.pololu.com)
- **PX** Parallax (www.parallax.com)
- **RS** RS (www.rs-online.com)
- **SF** SparkFun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

BASIC XBEE BREADBOARD CIRCUIT

Several of the projects in this chapter use XBee radios. In each project, you have the option of using convenience modules like the Arduino wireless shield, Spark Fun's XBee Explorer Regulated, or the LilyPad XBee. If you choose not to use those parts, you can build a basic XBee breadboard circuit. Variations on this circuit are shown in Figures 7-5, 7-6, 7-8, and 7-14. The basic parts for it are listed below.

» **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2
» **1 Digi XBee 802.15.4 RF module** **J** 2113375, **SF** WRL-08664, **AF** 128, **F** 1546394, **SS** WLS113A4M, **MS** MKAD14
» **1 3.3V regulator** **J** 242115, **D** 576-1134-ND, **SF** COM-00526, **F** 1703357, **RS** 534-3021
» **1 XBee breakout board** **J** 32403, **SF** BOB-08276, **AF** 127
» **2 rows of 0.1-inch header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411
» **2 rows of 2mm female headers** **J** 2037747, **D** 3M9406-ND, **F** 1776193
» **1 1µF capacitor** **J** 94161, **D** P10312-ND, **F** 8126933, **RS** 475-9009
» **1 10µF capacitor** **J** 29891, **D** P11212-ND, **F** 1144605, **RS** 715-1638
» **2 LEDs** **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590
» **2 220-ohm resistors** **D** 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842

PROJECT 13: Reporting Toxic Chemicals in the Shop

» **2 Basic XBee breadboard circuits** As described above.
» **1 5V regulator** **J** 51262, **D** LM7805CT-ND, **SF** COM-00107, **F** 1860277, **RS** 298-8514
» **1 9–12V DC power supply** Either a 9V battery or plug-in supply will do.
J 170245, **SF** TOL-00298, **AF** 63, **F** 636363, **P** 1463
» **1 Hanwei gas sensor** Hanwei makes a wide range of gas sensors, and many of the retailers in this book carry them. MQ-7 detects carbon monoxide; MQ-3 detects alcohol; MQ-6 detects propane and related gases; MG811 detects carbon dioxide. They all have similar operations, so pick one that you're most interested in.
SF SEN-08880, SEN-09404, or SEN-09405, **P** 1480, 1634, 1633, 1481, 1482, or 1483, **PX** 605-00007, 605-00008, 605-00009, 605-00010, or 605-00011
» **1 gas sensor breakout board** The Hanwei sensors have a pin layout that's not friendly to a breadboard, so these boards correct that. Pololu and Spark Fun both make a model.
SF BOB-08891, **P** 1479 or 1639
» **1 10-kilohm resistor** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906
» **1 Arduino Ethernet board** **A** A000050
Alternatively, an Uno-compatible board (see Chapter 2) with an Ethernet shield will work.
SF DEV-09026, **J** 2124242, **A** 139, **AF** 201, **F** 1848680
» **1 Arduino wireless shield** Alternatively, basic XBee breadboard circuit (see above) will work.
A A000064 or A000065. Alternatives: **SF** WRL-09976, **AF** 126, **F** 1848697, **RS** 696-1670, **SS** WLS114A0P
» **1 cymbal monkey** The one used here is a Charlie Chimp, ordered from the Aboyd Company (www.aboyd.com), part number ABC 40-1006.

NOTE: If your monkey uses a 3V power supply (such as two D batteries), you won't need the 3.3V regulator. Make sure that there's adequate amperage supplied for the radios. If you connect the circuit as shown and the radios behave erratically, the monkey's motor may be drawing all the power. If so, use a separate power supply for the radio circuit.

» **1 potentiometer** **J** 29082, **SF** COM-09939, **F** 350072, **RS** 522-0625
» **1 TIP120 Darlington NPN transistor** **D** TIP120-ND, **J** 32993, **F** 9804005
» **1 1N4004 power diode** **D** 1N4004-E3 or 23GI-ND, **J** 35992, **F** 9556109, **RS** 628-9029
» **1 1-kilohm resistor** **D** 1.0KQBK-ND, **J** 29663, **F** 1735061, **RS** 707-8669

» **1 100µF capacitor** **J** 158394, **D** P10269-ND, **F** 1144642, **RS** 715-1657

PROJECT 14: Relaying Solar Cell Data Wirelessly

» **1 USB-XBee adapter** **J** 32400, **SF** WRL-08687, **AF** 247, **PX** 32400

» **1 Basic XBee breadboard circuit** As described above. This will be the base for the solar cell circuit

» **1 5V regulator** **J** 51262, **D** LM7805CT-ND, **SF** COM-00107, **F** 1860277, **RS** 298-8514

» **3 4700µF electrolytic capacitors** **J** 199006, **D** P10237-ND, **F** 1144683, **RS** 711-1526

» **1 MAX8212 voltage monitor** You can order free samples from Maxim (www.maxim-ic.com). **D** MAX8212CPA+-ND, **F** 1610130

» **1 10-kilohm resistor** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906

» **3 100-kilohm resistors** **D** 100KQBK-ND, **J** 29997, **F** 9337695, **RS** 707-8940

» **1 4.7-kilohm resistor** **D** CF14JT4K70CT-ND, **J** 691024, **F** 735033, **RS** 707-8693

» **1 1-kilohm resistor** **D** 1.0KQBK-ND, **J** 29663, **F** 1735061, **RS** 707-8669

» **1 2N3906 PNP-type transistor** **J** 178618, **D** 2N3906D26ZCT-ND, **SF** COM-00522, **F** 1459017, **RS** 294-328

» **1 solar cell** **SF** PRT-07840, **P** 1691

» **1 Arduino Ethernet board** **A** A000050 Alternatively, an Uno-compatible board (see Chapter 2) with an Ethernet shield will work. **SF** DEV-09026, **J** 2124242, **A** 139, **AF** 201, **F** 1848680

» **1 Arduino wireless shield** Alternatively, basic XBee breadboard circuit (see above) will work. **A** A000064 or A000065. Alternatives: **SF** WRL-09976, **AF** 126, **F** 1848697, **RS** 696-1670, **SS** WLS114A0P

» **1 9V battery clip** **D** 2238K-ND, **J** 101470, **SF** PRT-09518, **F** 1650675

» **1 9V battery** You can use three or four AA batteries instead—if you have a battery holder for them. Anything in the 5–12V range should work.

» **XBee LilyPad** This will be the unit for the relay radio. Alternatively, an XBee Explorer Regulated, or a Basic XBee breadboard circuit, as described above, will work. **SF** DEV-08937

» **1 XBee Explorer Regulated** This is an alternative to the XBee LilyPad. **SF** WRL-09132

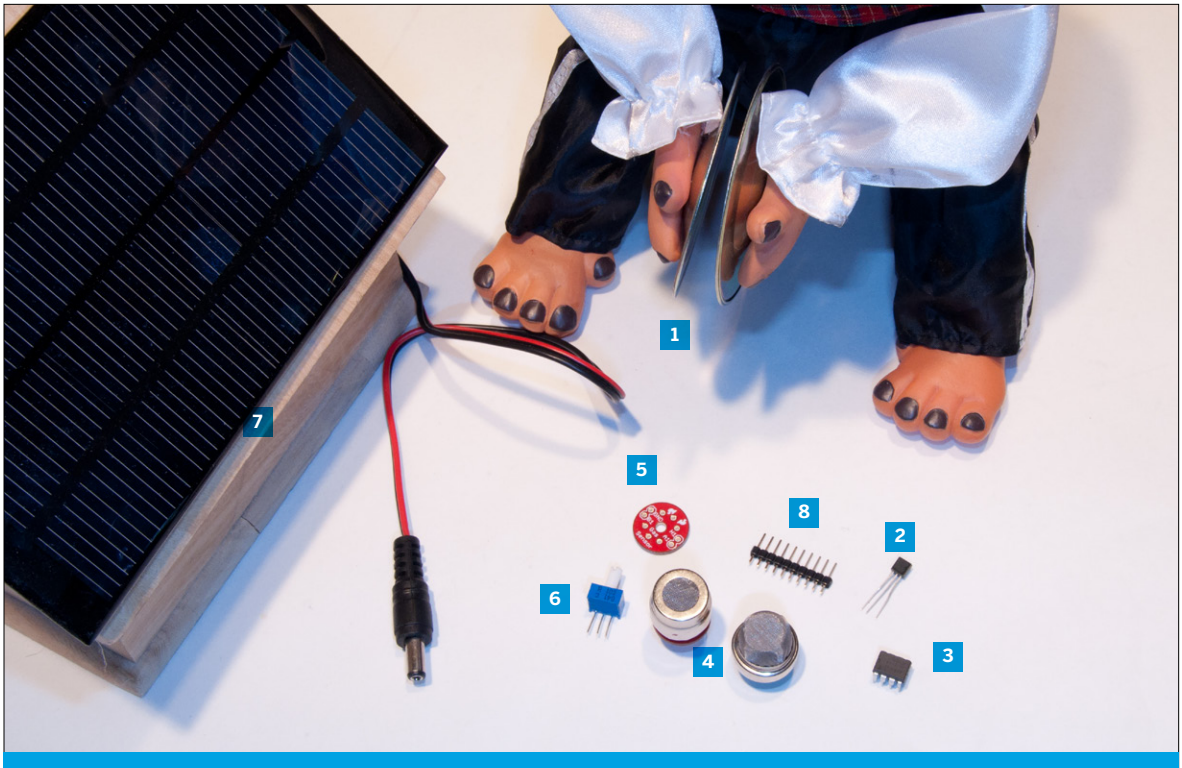


Figure 7-1. New parts for this chapter: **1.** Charley Chimp **2.** 2N3906 transistor **3.** MAX8212 Voltage trigger **4.** Hanwei gas sensors **5.** Gas sensor breakout board **6.** Trimmer potentiometer **7.** Solar cell **8.** Don't forget plenty of male header pins for the breakout boards.

Sessions vs. Messages

So far, most of the communication in this book has involved opening a dedicated connection between two points for the duration of the conversation. This is session-based communication. Sometimes you want to make a network in which objects can talk to each other more freely, switching conversational partners on the fly, or even addressing the whole group if the occasion warrants. For this, you need a more message-based protocol.

Sessions Versus Messages

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

Not all network communication is session-based. There's another protocol used on the Internet, called the User Datagram Protocol, or UDP. With UDP, you compose a message, give it an address, send it, and forget about it.

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

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

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

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

X

Who's Out There? Broadcast Messages

The first advantage to sessionless protocols like UDP and 802.15.4 is that they allow for broadcasting messages to everyone on the network at once. Although you don't want to do this all the time—because you'd flood the network with messages that not every device needs—it's a handy ability to have when you want to find out who else is on your network. You simply send out a broadcast message asking "Who's there?" and wait for replies.

Querying for Other Devices Using UDP

Arduino's Ethernet library includes the ability to send and receive UDP packets, so you can use it to write a simple sketch that listens for broadcast messages and responds to them. It's useful when you want to make a large number of networked devices all respond at once. You can use Processing to send your broadcast query.

There's no way to send UDP messages using the Processing Network library, but there's a good free UDP library by Stephane Cousot available from the Processing.org's libraries page at http://processing.org/reference/libraries/#data_protocols. To use it, download it and make a new directory called **udp** in the **libraries** subdirectory of your Processing sketch directory. Then unzip the contents of the download and drop them in the directory you created. After that, restart Processing and you're ready to use the UDP library.

Any Arduino with an Ethernet shield connected to your network will do for this exercise. You don't need any extra hardware other than the shield.

Try It This Processing sketch sends out a broadcast UDP message on port 43770. This is an arbitrary number that's high enough that it's probably not used by other common applications.

It doesn't matter what the IP address or router of the machine running this sketch is, because it uses the special subnet broadcast address: 255.255.255.255.

Unlike session-based messaging, where you wrote and read bytes from a stream like serial communication, UDP messages are all sent with one discrete message, udp.send(), which includes the address and port for sending.

```
/*
  UDP broadcast query sender/receiver
 Context: Processing
 */
// import the UDP library:
import hypermedia.net.*;

UDP udp;      // initialize the UDP object

void setup() {
  udp = new UDP( this, 43770 );  // open a UDP port
  udp.listen( true );            //  listen for incoming messages
}

void draw()
{
}

void keyPressed() {
  String ip = "255.255.255.255";   // the remote IP address
  int port = 43770;                // the destination port
  udp.send("Hello!\n", ip, port ); // the message to send
}
```



The response is stored in an array called data[], and the receive() method in the UDP library prints it out one character at a time.

Continued from previous page.

```
void receive( byte[] data ) {
  // print the incoming data bytes as ASCII characters:
  for(int thisChar=0; thisChar < data.length; thisChar++) {
    print(char(data[thisChar]));
  }
  println();
}
```

Respond To It

Here's an Arduino sketch that listens for UDP messages on the same port and responds in kind. Load it onto an Arduino connected to your network with an Ethernet shield.

» Change these to match your own device and network.

```
/*
  UDP Query Responder
 Context: Arduino
 */

#include <SPI.h>
#include <Ethernet.h>
#include <Udp.h>

// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = {
  0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress myIp(192,168,1,20);
unsigned int myPort = 43770;      // local port to listen on

// A UDP instance to send and receive packets over UDP
UDP query;

void setup() {
  // start the serial library:
  Serial.begin(9600);

  // start the Ethernet connection:
  Ethernet.begin(mac, myIp);
  // print your address:
  for (int thisByte = 0; thisByte < 4; thisByte++) {
    Serial.print(Ethernet.localIP()[thisByte], DEC);
    Serial.print(".");
  }
  Serial.println();

  query.begin(myPort);
  // give the Ethernet shield a second to initialize:
  delay(1000);
}
```



The loop() just calls the listen() method, which listens for incoming UDP packets on the same port on which the Processing sketch broadcasts.

Receiving UDP packets is a bit different than receiving TCP packets. Each packet contains a header that tells you who sent it and from what port, kind of like an envelope. When you receive a packet, you have to parse out that data. The parsePacket() method in the Arduino UDP library does just that. You can see it in action here. First, you parsePacket() to separate the header elements from the message body. Then, you read the message body byte-by-byte, just like you've done already with TCP sockets and serial ports.

Continued from opposite page.

```
void loop()
{
  listen(query, myPort);
}

void listen(UDP thisUDP, unsigned int thisPort) {
  // check to see if there's an incoming packet, and
  // parse out the header:
  int messageSize = thisUDP.parsePacket();
  // if there's a payload, parse it all out:
  if (messageSize > 0) {
    Serial.print("message received from: ");
    // get remote address and port:
    IPAddress yourIp = thisUDP.remoteIP();
    unsigned int yourPort = thisUDP.remotePort();
    for (int thisByte = 0; thisByte < 4; thisByte++) {
      Serial.print(yourIp[thisByte], DEC);
      Serial.print(".");
    }
    Serial.println(" on port: " + String(thisPort));
    // send the payload out the serial port:
    while (thisUDP.available() > 0) {
      // read the packet into packetBufffer
      int udpByte = thisUDP.read();
      Serial.write(udpByte);
    }
    sendPacket(thisUDP, Ethernet.localIP(), yourIp, yourPort);
  }
}
```

First, you parse the packet to get the header items.

Then, you read the rest of the message body.

The sendPacket() method sends a response packet to whatever address and port it receives a message from. It includes its IP address in the body of the message as an ASCII string.

You can see that UDP messaging is different than the socket-based messaging over TCP. Each message packet has to be started and ended. When you beginPacket(), the Ethernet controller starts saving bytes to send in its memory; when you endPacket(), it sends them out.

```
void sendPacket(UDP thisUDP, IPAddress thisAddress,
          IPAddress destAddress, unsigned int destPort) {
  // set up a packet to send:
  thisUDP.beginPacket(destAddress, destPort);
  for (int thisByte = 0; thisByte < 4; thisByte++) {
    // send the byte:
    thisUDP.print(thisAddress[thisByte], DEC);
    thisUDP.print(".");
  }
  thisUDP.println("Hi there!");
  thisUDP.endPacket();
}
```

When you've got the Arduino programmed, open a serial connection to it using the Serial Monitor. You should see a message like this in the Arduino Serial Monitor when the Arduino obtains an IP address:

```
192.168.1.20.
```

Next, run the Processing sketch and type any key. In the Arduino Serial Monitor, you should see:

```
message received from: 192.168.1.1. on port: 43770
Hello!
```

The address will be your computer's IP address. In the Processing monitor pane, you should see:

```
Hello!

192.168.1.20.Hi there!
```

The first Hello! is from Processing itself. When you send a broadcast message, it comes back to you as well! The second line is from the Arduino. If you have several Arduinos on your network all running this sketch, you'll get a response from each one of them. In fact, this exercise is most satisfying when you have multiple units responding.

This is a handy routine to add to any Ethernet-based project. You have to provide a separate UDP instance and port to listen on, but the sendPacket() method can work with other programs to get a response from your device when you send broadcast queries.

X

Querying for XBee Radios Using 802.15.4 Broadcast Messages

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

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

For the purposes of this exercise, you'll need at least two XBee radios connected to serial ports on your computer. The easiest way to do this is by using a USB-to-XBee Serial adapter like the XBee Explorer.

Once you've got the radios connected and working, open a serial terminal connection to one of them and issue the following command: +++. Then, wait for the radio to respond with OK. Then type (remember, \r means carriage return, so press Enter or Return instead of \r): ATND\r.

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

```
1234
13A200
400842A7
28
TIGOE1

5678
13A200
400842A9
1E
TIGOE3
```

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

Replace myname with the name you want.

Broadcast messages are useful for many reasons. However, they should be used sparingly because they create more traffic than is necessary. In the next project, you'll use broadcast messages to reach all the other objects in a small, closed network.

Upgrading the Firmware on XBee Radios

To use the node discover, node identifier, and some of the other XBee AT commands covered in this chapter, your XBee radios need to be upgraded to at least version 10A1. To check the firmware version of your radios, send the following command: `ATVR\r`. The radio will respond: 10A2. If the number is 10A1 or above (remember, it's in hexadecimal),you're good to go. If not, go to http://www.digi.com/support/kbase/kbaseresultdetl.jsp?kb=125 and download the X-CTU software. Bad news, Mac OS X and Linux users: it only runs on Windows. You can run it under VirtualBox (free), Parallels ($80), VMWare ($80), or Windows using Bootcamp (Mac) or by dual booting (Linux).

Once you've installed the software, launch it. On the PC Settings tab, you'll be able to select the serial port to which your XBee radio is attached. Pick the port and leave the rest of the settings at their defaults. Click the Modem Configuration tab and you'll get to where you can update the firmware. Click the Read button to read the current firmware on your radio. You'll get a screenful of the settings of your radio, similar to that shown in Figure 7-2. The firmware version is shown in the upper-righthand corner. You can pull down that menu to see the latest versions available. Pick the latest one, then check the "Always update firmware" checkbox. Leave the Function Set menu choice set to XBEE 802.15.4. Then click the Write button. The software will download the new firmware to your radio, and you're ready to go. The X-CTU software is useful to keep around, because it also lets you change and record your radio's settings without having to use the AT commands.

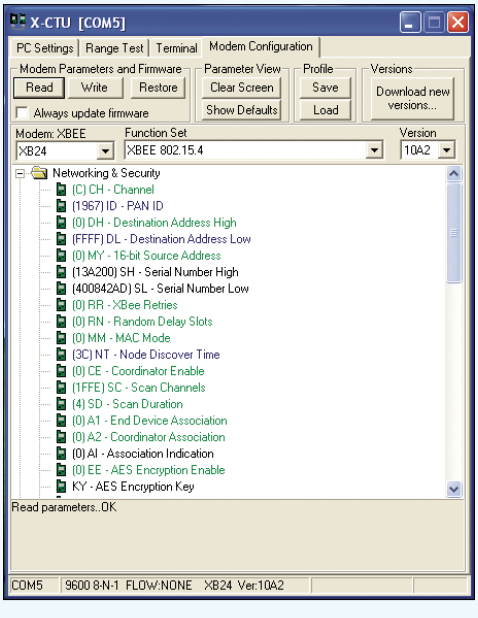


Figure 7-2. The X-CTU Modem Configuration tab.

Project 13

Reporting Toxic Chemicals in the Shop

If you've got a workshop, you'll appreciate this project. By attaching a volatile gas sensor to an XBee radio, you'll be able to sense the concentration of solvents in the air in your shop. When you're working in the shop by yourself, it's common to become insensitive to the fumes of the chemicals with which you're working. This project is an attempt to remedy that issue.



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

The sensor values are sent to two other radios. One is attached to an Arduino with an Ethernet shield, which is connected to the Internet as a web server. The other radio is attached to a cymbal-playing toy monkey, located elsewhere in the house, that makes an unholy racket when the organic solvent levels in the shop get high. That way, the rest of the family will know immediately if your shop is toxic. If you don't share my love of monkeys, this circuit can control anything that can be switched on from a transistor Figure 7-3 shows the completed elements of the project. Figure 7-4 shows this project's network.

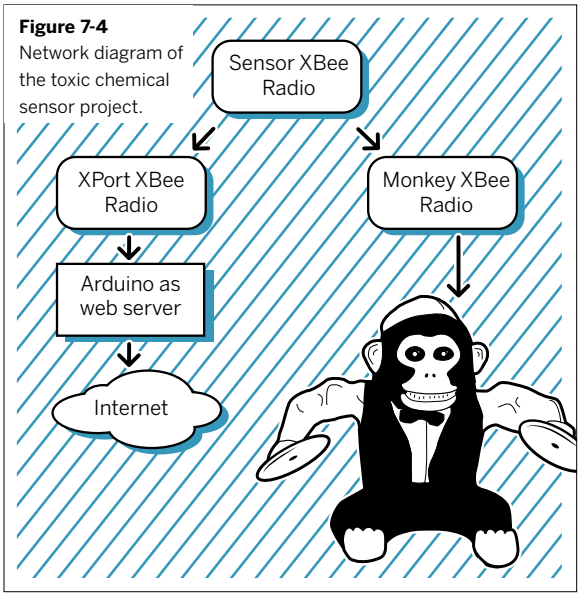


Figure 7-4
Network diagram of the toxic chemical sensor project.

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

MATERIALS

» **1 USB-to-XBee serial adapter**

Sensor Circuit

» **1 solderless breadboard**
» **1 Digi XBee OEM RF module**
» **1 5V regulator**
» **1 3.3V regulator**
» **1 9–12V DC power supply** Either a 9V battery or plug-in supply will do.
» **1 XBee breakout board**
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows**
» **1 1µF capacitor**
» **1 10µF capacitor**
» **1 Hanwei gas sensor**
» **1 gas sensor breakout board**
» **2 LEDs**
» **2 220-ohm resistors**
» **1 10-kilohm resistor**

Internet Connection Circuit

» **1 Arduino microcontroller module**
» **1 Arduino Ethernet shield**
» **1 Digi XBee OEM RF module**
» **1 Arduino wireless shield** If you use this, you won't need the rest of the parts listed below. They are all for connecting the XBee to your Arduino.
» **1 solderless breadboard**
» **1 3.3V regulator**
» **1 1µF capacitor**
» **1 10µF capacitor**
» **1 XBee breakout board**
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows**
» **2 LEDs**
» **2 220-ohm resistors**

Cymbal Monkey Circuit

» **1 solderless breadboard**
» **1 Digi XBee OEM RF module**
» **1 cymbal monkey**

NOTE: If your monkey uses a 3V power supply (such as two D batteries), you won't need the 3.3V regulator. Make sure that there's adequate amperage supplied for the radios. If you connect the circuit as shown and the radios behave erratically, the monkey's motor may be drawing all the power. If so, use a separate power supply for the radio circuit.

» **1 3.3V regulator**
» **1 XBee breakout board**
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows**
» **2 LEDs**
» **2 220-ohm resistors**
» **1 10K trimmer potentiometer**
» **1 TIP120 Darlington NPN transistor**.
» **1 1N4004 power diode**
» **1 1-kilohm resistor**
» **1 100µF capacitor**

You'll be building three separate circuits for this project, so the parts list is broken down for each one. Most of these items are available at retailers other than the ones listed here,.

Radio Settings

Connect one of the radios to the USB-to-XBee serial adapter. You'll use this for configuring the radios only. You've got three radios: the sensor's radio, the monkey's radio, and the Arduino's radio. In Chapter 6 you saw how to configure the radios' addresses, destination addresses, and Personal Area Network (PAN) IDs. In this project, you'll see how to configure some of their I/O pins' behavior. For example, you can configure the digital and analog I/O pins to operate as inputs, outputs, or to turn off. You can also set them to be digital or analog inputs, or digital or pulse-width modulation (PWM) outputs. You can even link an output pin's behavior to the signals it receives from another radio. (If you don't remember how to configure the XBee, see Step 1 in Project 10).

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

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

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

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

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

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

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

Here's a summary of all the settings:

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

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

```
+++
```

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

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

For the monkey radio, the configuration is:

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

And for the Arduino's radio, it's:

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

The Circuits

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

The sensor circuit

The gas sensor takes a 5V supply voltage, so you need a 5V regulator for it, a 3.3V regulator for the XBee, and a power supply that's at least 9V to supply voltage to the circuit. A 9V battery will do, or a 9–12V DC power adapter. Figure 7-5 shows the circuit. The gas sensor's output voltage should stay below 3.3V under the most likely shop conditions, but test it before connecting it to an XBee. Connect and power the circuit, but leave out the wire connecting the sensor's output to the XBee's analog input. Power up the circuit, and let it heat for a minute or two. The circuit takes time to warm up because there's a heater element in the sensor. Measure the voltage between the sensor's output and ground. You should get about 0.1 volt if the air is free of volatile organic compounds (VOCs). While still measuring the voltage, take a bottle of something that has an organic solvent (I used hand sanitizer, which has a lot of alcohol in it), and gently waft the fumes over the sensor. Be careful not to breathe it in yourself. You should get something considerably higher—up to 3 volts. If the voltage exceeds 3.3V, change the fixed resistor until you get results in a range below 3.3V, even when the solvent's fumes are high. Once you've got the sensor reading in an acceptable range, connect its output to the XBee's analog input pin, which is pin 20. Make sure to connect the XBee's voltage reference pin (pin 14) to 3.3 volts as well.

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

To test whether the XBee is reading the sensor correctly, connect its TX pin to the USB-to-XBee serial adapter's TX pin, its RX pin to the adapter's RX pin, and connect their ground lines together. Make sure there's no XBee in the socket when you do this. You're just using the adapter to connect to this circuit. Then plug the adapter into your computer and open a serial connection to it. Type `+++` and wait for the OK. Then type `ATIS\r`. This command forces the XBee to read the analog inputs and return a series of values. You'll get a reply like this:

```
1
200
3FF
```

Don't worry about what the values yet; all that matters is that you're getting something. You'll see the actual values as the project develops later.

The monkey circuit

To control the monkey, disconnect the monkey's motor from its switch and connect the motor directly to the circuit shown in Figure 7-6. The monkey's battery pack supplies 3V, which is enough for the XBee radio, so you can power the whole radio circuit from the monkey. Connect leads from the battery pack's power and ground to the board. If your monkey runs on a different voltage, make sure to adapt the circuit accordingly so that your radio circuit is getting at least 3V. Figure 7-7 shows the modifications in the monkey's innards. I used an old telephone cord to wire the monkey to the board.

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

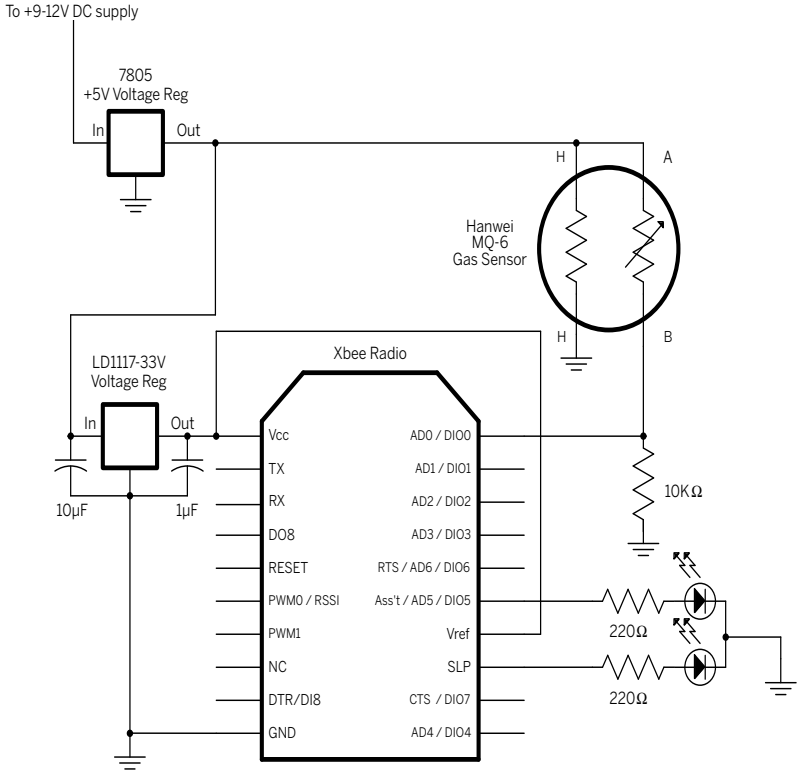
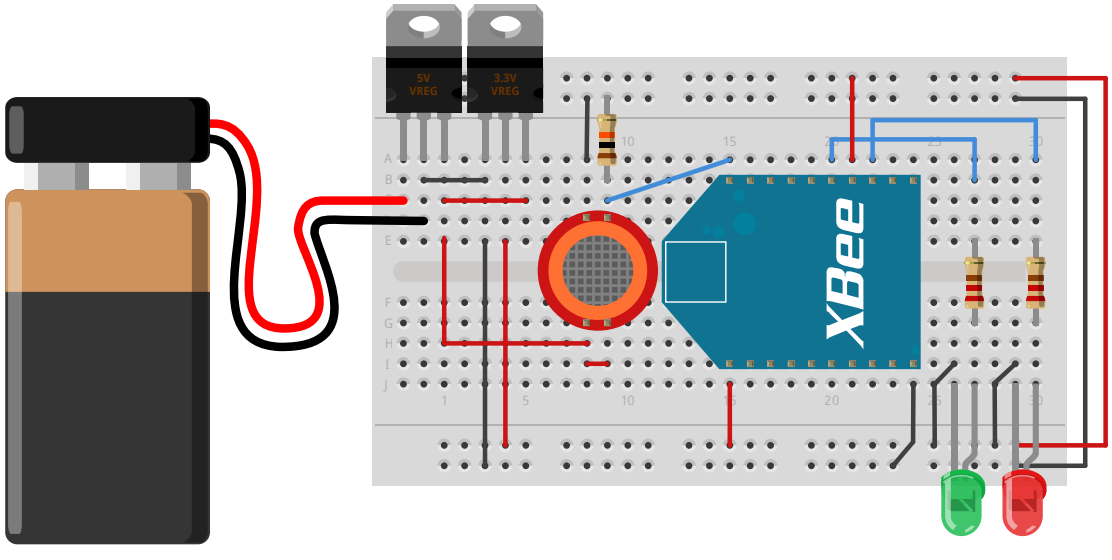


Figure 7-5
XBee radio connected to a gas sensor. The MQ-6 is shown here, but the same circuit will work for many of the Hanwei sensors. You may have to change the 10K resistor to adjust the output voltage range, however.

The XBee shown on the breadboard below is mounted on a breakout board. The XBee's pins will not fit into a breadboard without the breakout board.

Many of the projects in this chapter are made using the LD1117-33V 3.3V voltage regulator. The pins on this regulator are configured differently from the pins on the other regulators used in this book. Check the data sheet for your regulator to be sure you have the pins correct. You should make a habit of checking this because, otherwise, you will damage your regulator or other parts of your circuit.



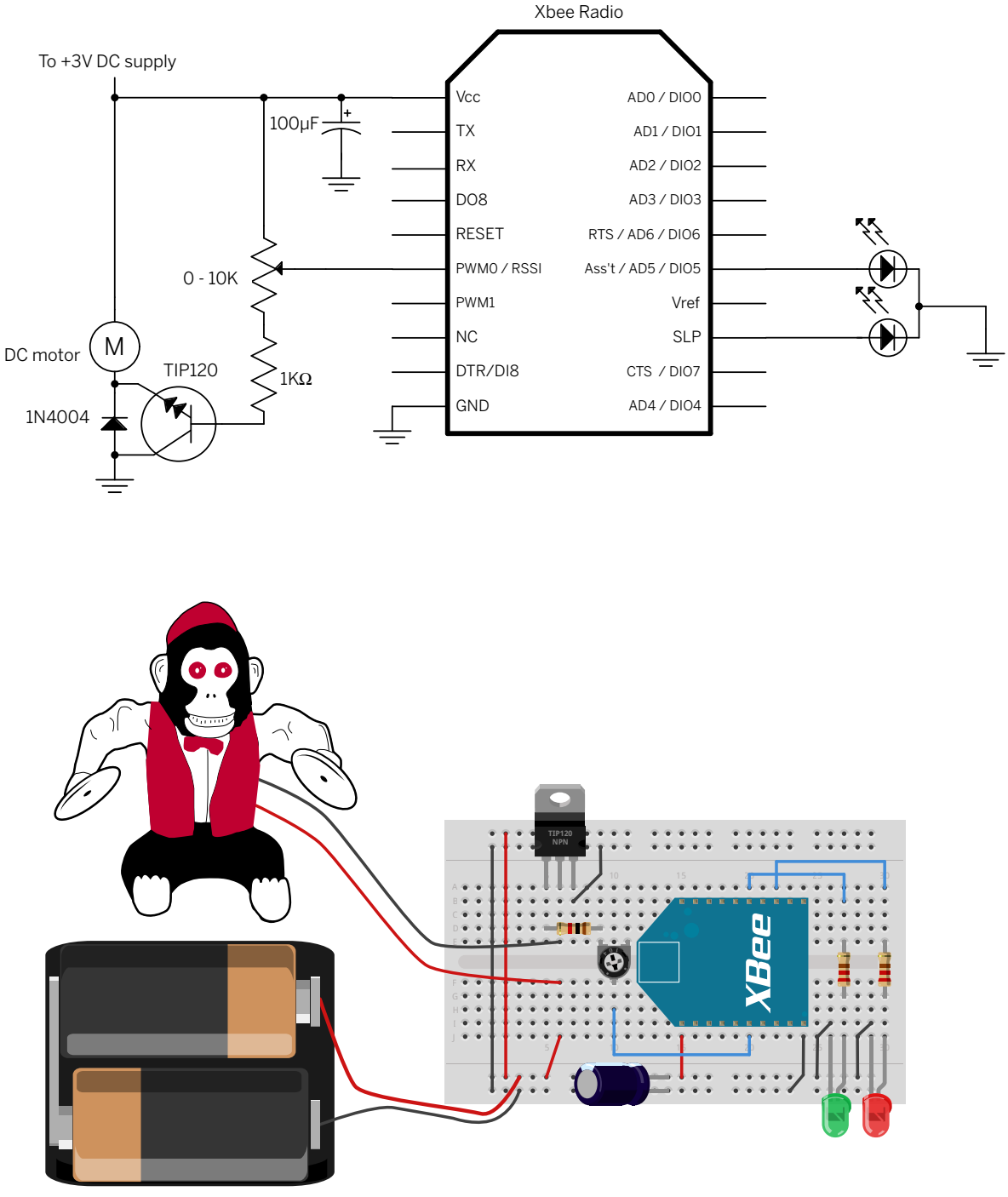
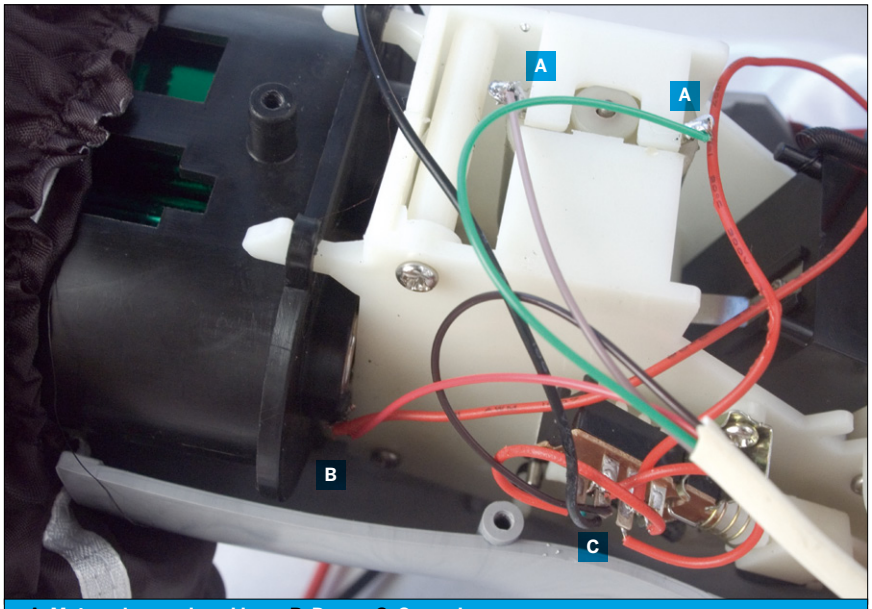


Figure 7-6
XBee radio connected to a cymbal monkey.



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

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

To test this circuit, make sure that the sensor radio is working, and turn it on. When the sensor's value is low, the motor should be turned off; when the sensor reads a high concentration of VOCs, the motor will turn on and the monkey will warn you by playing his cymbals. Use the potentiometer to affect the motor's activation threshold. Start with the potentiometer set very high, then slowly turn it down until the motor turns off. At that point, expose the sensor to some alcohol—the motor should turn on again. It should go off when the air around the sensor is clear. If you're unsure whether the motor circuit is working correctly, connect an LED and 220-ohm resistor in series from 3V to the collector of the transistor instead of the motor. The LED should grow brighter when the sensor reading is higher, and dimmer when the sensor reading is lower. The LED has no physical inertia like the motor does, and it consumes less current, so it turns on at a much lower duty cycle.

The Arduino circuit

If you're using an Arduino wireless shield, all you need to do for this project is stack that on top of an Ethernet shield or Arduino Ethernet board, and add them both to your microcontroller. If you're building the circuit yourself, it's shown in Figure 7-8.

The Arduino Ethernet and XBee circuit look a lot like the circuit you used for the duplex radio transmission project in Chapter 6. All you're doing differently is adding an Ethernet shield. You don't need to connect the XBee's receive line to the Arduino's transmit line because the Arduino's never going to talk to the XBee—it will just listen. You also won't need the LED, or the potentiometer from that project.

When the Arduino receives the message from the gas sensor XBee, it will parse the message, extract the sensor value, and convert it to a voltage. It will act as a web server, so anyone who wants to know the sensor level can check on the Web. In order to make that happen, it's best to break the task into two parts: first, establish that you can read and parse the XBee's messages; then, add the web server component to your code.

Reading the XBee Protocol

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

- Byte 1: 0x7E, the start byte value.
- Byte 2–3: Packet size, a 2-byte value. This depends on your other settings.
- Byte 4: API identifier value, a code that says what this response is.

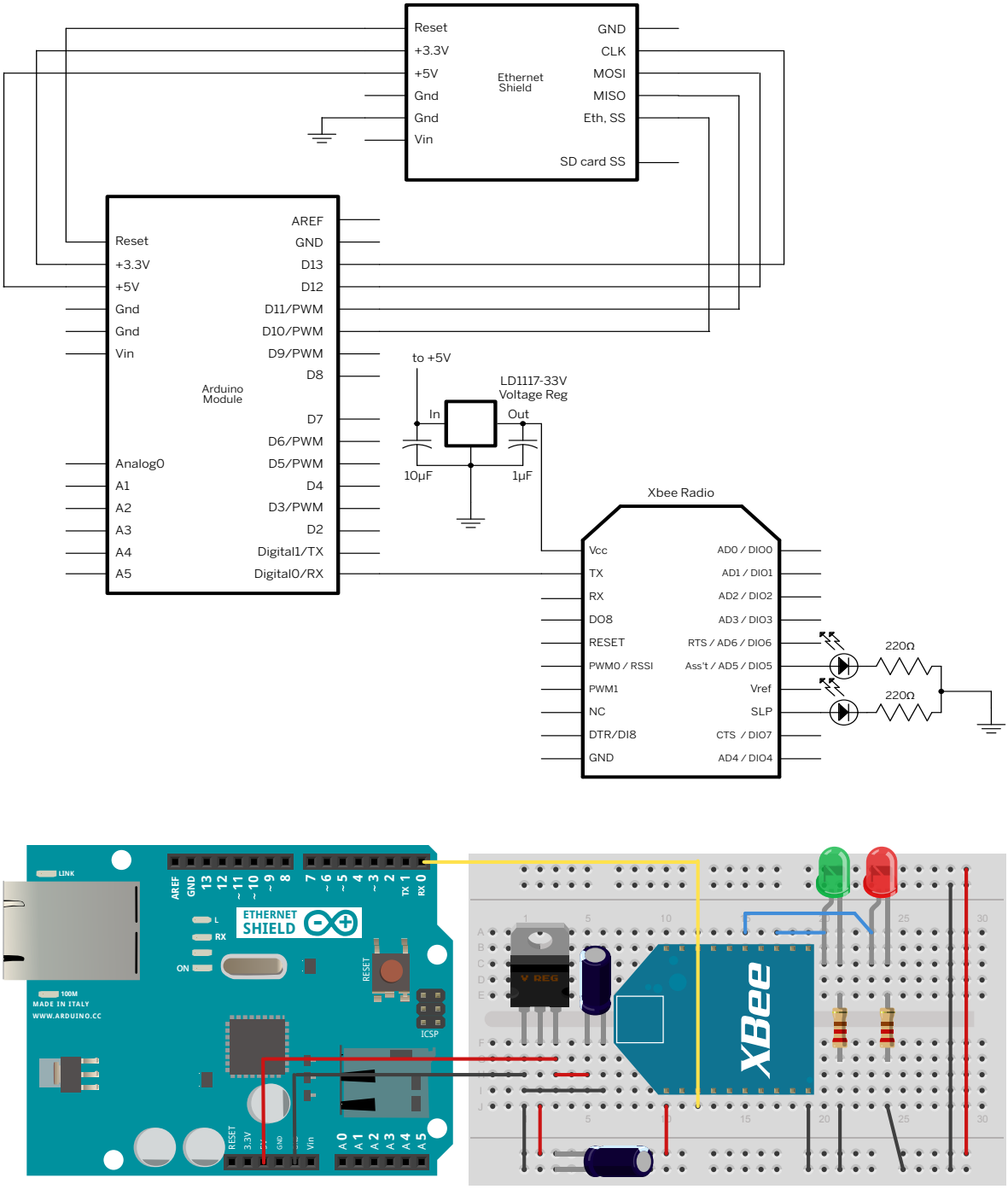


Figure 7-8
XBee radio connected to an Arduino with Ethernet shield.

- Byte 5–6: XBee sender's address.
- Byte 7: Received Signal Strength Indicator (RSSI).
- Byte 8: Broadcast options (not used here).
- Byte 9: Number of samples in the packet (you set it to 5 using the IT command shown earlier).
- Byte 10–11: Which I/O channels are currently being used. This example assumes only one analog channel, AD0, and no digital channels are in use.
- Byte 12–21: 10-bit values, each ADC sample from the sender. Each 10-bit sample represents a voltage from 0 to the XBee's maximum of 3.3V, so a value of 0 means 0 volts, and a value of 1023 means 3.3 volts. Each point of the sensor's value is then 3.3/1024, or 0.003 volts.

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

X

Read It

The following sketch reads bytes and puts them in an array until it sees the value 0x7E. Then, it parses the array to find the sensor's value. To start, you need the message length (or packet length), an array, and a counter to know where you are in the array. Then, you need to set up serial communications.

```
/*
  XBee message reader
 Context: Arduino
 */
const int packetLength = 22;    // XBee data messages will be 22 bytes long
int dataPacket[packetLength];   // array to hold the XBee data
int byteCounter = 0;            // counter for bytes received from the XBee

void setup() {
  // start serial communications:
  Serial.begin(9600);
}
```

The main loop doesn't do much in this sketch; it just calls a method, listenToSerial(), that goes through the bytes and does the work.

```
void loop() {
  // listen for incoming serial data:
  if (Serial.available() > 0) {
    listenToSerial();
  }
}
```

To start with, just read the bytes in and look for the value 0x7E. When you see it, print a linefeed. No matter what, print the byte values separated by a space.

```
void listenToSerial() {
  // read incoming byte:
  int inByte = Serial.read();
  // beginning of a new packet is 0x7E:
  if (inByte == 0x7E ) {
     Serial.println();
  }
  // print the bytes:
  Serial.print(inByte, DEC);
  Serial.print(" ");
}
```

NOTE: In this circuit, you didn't connect the Arduino's transmit pin to the XBee's receive pin on purpose. This way, you can use the serial connection both to receive from the XBee and to send debugging information to the Serial Monitor.

When you run this sketch, you'll get a result like what's shown to the right.

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

Each line that starts with 126 (0x7E in hexadecimal) is a single XBee message. Most of the lines have 22 bytes, corresponding to the packet format described earlier. You may wonder why the first line of the output shown here didn't have a full complement of bytes. It's simply because there's no way to know what byte the XBee is sending when the Arduino first starts listening. That's OK, because the code below will filter out the incomplete packets.

Refine It

It would be handy to have each packet in its own array. That's how you'll use the global variables you set up before.

Next, change the listenToSerial() method, adding code to parse the dataPacket array; then, empty the array to prepare to receive new data. Replace everything in the if statement with the code shown in blue, and remove the two lines that printed the byte and the space.

This function calls another function, parseData(), that extracts and averages the sensor readings in bytes 12 to 21 of the packet.

```
void listenToSerial() {
  // read incoming byte:
  int inByte = Serial.read();
  // beginning of a new packet is 0x7E:
  if (inByte == 0x7E ) {
    // parse the last packet and get a reading:
    int thisReading = parseData();
    // print the reading:
    Serial.println(thisReading);
    // empty the data array:
    for (int thisByte = 0; thisByte < packetLength; thisByte++) {
      dataPacket[thisByte] = 0;
    }
    // reset the incoming byte counter:
    byteCounter = 0;
  }
  // if the byte counter is less than the data packet length,
  // add this byte to the data array:
  if (byteCounter < packetLength) {
    dataPacket[byteCounter] = inByte;
    // increment the byte counter:
    byteCounter++;
  }
}
```

Parse It

This function goes through each complete message packet byte-by-byte, assembling the relevant values from the various bytes. When it's done, it returns the average of the five samples in the packet.

```
int parseData() {
  int adcStart = 11;                    // ADC reading starts at byte 12
  int numSamples = dataPacket[8];       // number of samples in packet
  int total = 0;                        // sum of all the ADC readings

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

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

When you run this sketch, you should get a value between 0 and 1023 every time you get a message from the sensor XBee radio. Make sure you're getting good values before you move on. If you're not, use the first version of the sketch that prints out the value of each byte to help diagnose the problem. Seeing every byte that you actually receive before you do anything with those bytes is the best way to get at the problem. There are a few common problems to look for:

- Are you getting anything at all? If not, is the sensor XBee transmitting?
- Does each radio have the correct settings? Connect them to a USB-to-XBee serial adapter and check in a serial terminal application.
- Is the receiving radio getting adequate power? Check the voltage between pins 1 and 10 of the receiving XBee. If you're using an older Arduino, and you're relying on the 3.3V output without a regulator, you may not be giving the radio enough power. The earlier Arduinos (before the Uno) didn't supply much amperage from the 3.3V output.

Also consider the advice in Figure 7-9 to make troubleshooting easier. Once you know you're receiving properly, move on to adding the web server code.

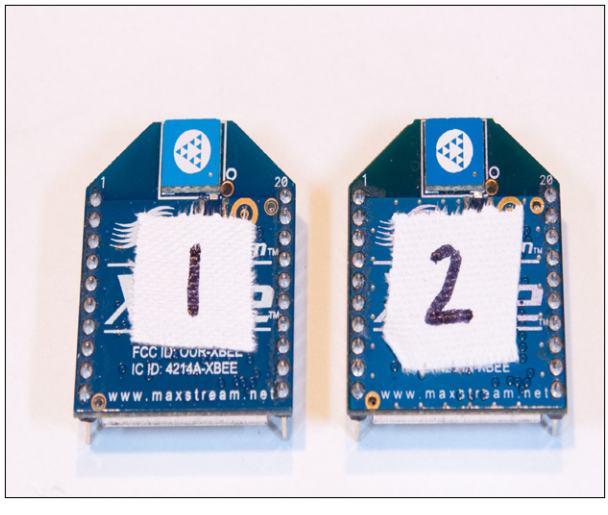


Figure 7-9
Until you're finished troubleshooting, it's a good idea to label your XBee radios so you can tell them apart without having to check their configurations serially.

X

Serve It

Now that you can read the XBee messages, add code to make a web server. The rest of this sketch is much like the RGB server from Chapter 4.

First, include the Ethernet and SPI libraries, and add a few more global variables at the top of your sketch to manage them. New lines are shown in blue.

```
// include SPI and Ethernet libraries:
#include <SPI.h>
#include <Ethernet.h>

// initialize a server instance:
EthernetServer server(80);

// Ethernet MAC address and IP address for server:
byte mac[] = {
  0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress ip(192,168,1,20);

String requestLine = "";        // incoming HTTP request from client

const int packetLength = 22;    // XBee data messages will be 22 bytes long
int dataPacket[packetLength];   // array to hold the XBee data
int byteCounter = 0;            // counter for bytes received from the XBee

// variables for calculating a sensor average:
const int averageInterval = 10 * 1000;  // time between averages, in seconds
long sensorTotal = 0;                   // used for averaging sensor values
float averageVoltage = 0.0;             // the average value, in volts
long lastAverageTime = 0;               // when you last took an average
int readingCount = 0;                   // readings since last average
```

» Change these to match your own device and network.

» Add some code to the main loop to listen for new clients and deal with them, and to average the values from the incoming XBee packets every 10 seconds. New lines are shown in blue.

```
void loop() {

  // listen for incoming serial data:
  if (Serial.available() > 0) {
    listenToSerial();
  }

  // listen for incoming clients:
  EthernetClient client = server.available();
  if (client) {
    listenToClient(client);
  }

  // calculate an average of readings after <averageInterval> seconds
  long now = millis();
  if (now - lastAverageTime > averageInterval) {
    averageVoltage = getAverageReading();
    Serial.println(averageVoltage);
  }
}
```

Just as you had a method earlier to listen to serial data, now you're adding one to listen to incoming HTTP client requests. This method reads the bytes and saves them in a string until it gets a linefeed. It's not actually parsing the lines, except to look for linefeeds or carriage returns. Whenever it gets two linefeeds in a row, it knows the HTTP request is over, and it should respond by calling the makeResponse() routine.

```
// this method listens to requests from an HTTP client:
void listenToClient( EthernetClient thisClient) {
  while (thisClient.connected()) {
    if (thisClient.available()) {
      // read in a byte:
      char thisChar = thisClient.read();
      // for any other character, increment the line length:
      requestLine = requestLine + thisChar;
      // if you get a linefeed and the request line
      // has only a newline in it, then the request is over:
      if (thisChar == '\n' && requestLine.length() < 2) {
        // send an HTTP response:
        makeResponse(thisClient);
        break;
      }
      //if you get a newline or carriage return,
      // you're at the end of a line:
      if (thisChar == '\n' || thisChar == '\r') {
        // clear the last line:
        requestLine = "";
      }
    }
  }
  // give the web browser time to receive the data
  delay(1);
  // close the connection:
  thisClient.stop();
}
```

The makeResponse() method sends an HTTP header and HTML response to the client, then closes the connection. It calls a method, getAverageReading(), which gets the average of all messages in the last 10 seconds and converts them to a voltage:

```
void makeResponse(EthernetClient thisClient) {
  // read the current average voltage:
  float thisVoltage = getAverageReading();

  // print the HTTP header:
  thisClient.print("HTTP/1.1 200 OK\n");
  thisClient.print("Content-Type: text/html\n\n");
  // print the HTML document:
  thisClient.print("<html><head><meta http-equiv=\"refresh\"
        content=\"3\">");
  thisClient.println("</head>");
  // if the reading is good, print it:
  if (thisVoltage > 0.0) {
    thisClient.print("<h1>reading = ");
    thisClient.print(thisVoltage);
    thisClient.print(" volts</h1>");
  }  // if the reading is not good, print that:
  else {
    thisClient.print("<h1>No reading</h1>");
  }
  thisClient.println("</html>\n\n");
}
```

The getAverageReading() is called every 10 seconds from the main loop, as well as every time a client makes a request. It takes the readings from a number of XBee messages and averages them all, as shown at right.

That's the whole sketch. Upload it to your Arduino, then open the Arduino's address in a browser, as shown in Figure 7-10.

Now when you're working in your shop, your family or friends have two ways to know whether the air is getting too foul for you to breathe. They can check the website or listen for the monkey to clang his cymbals.

NOTE: This code doesn't actually translate output voltage into concentration of gases. To do that calculation, you'll need to read the datasheet for the sensor you chose.

```
float getAverageReading() {
  int thisAverage = sensorTotal / readingCount;
  // convert to a voltage:
  float voltage = 3.3 * (thisAverage / 1024.0);
  // reset the reading count and totals for next time:
  readingCount = 0;
  sensorTotal = 0;
  // save the time you took this average:
  lastAverageTime = millis();
  // return the result:
  return voltage;
}
```

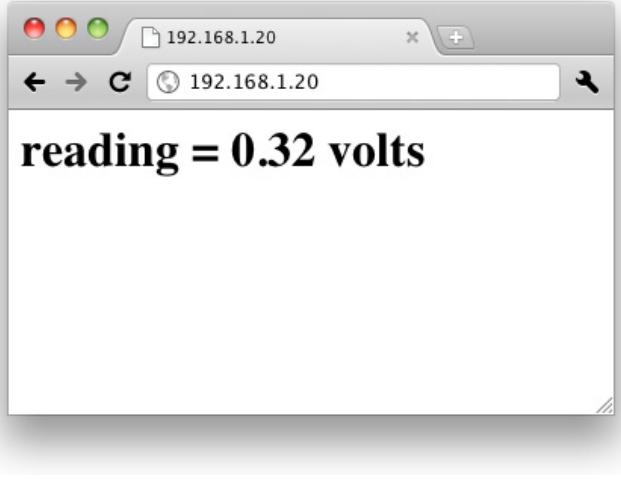


Figure 7-10
The results of the XBee gas server.

What Happens in the Subnet Stays in the Subnet

You're probably thinking to yourself, "That's great, but how come I'm not sending these messages over the Internet via UDP? Isn't that the message-based protocol that's parallel to the message-based protocol of the XBee radios?" Yes, it is. However, it's wise to be careful with the use of UDP outside your own subnet. If you were to broadcast beyond your own subnet to the rest of the Net, you'd flood the whole network with unnecessary traffic. UDP and message-based protocols are used on the wider Net but, in general, they are used in a directed manner rather than as broadcast messages. For example, video and audio transmissions on the Internet are generally directed UDP streams. The client contacts the server first using an HTTP request, or a similar TCP-based request. The server tells the client to open a port on which to recieve UDP messages, the client opens the port, and the server starts a stream of UDP messages. That's a more polite use of UDP on the wider Internet.

In the next project, you'll use UDP-directed messages from one device to another, as well as XBee-directed messages. X

Directed Messages

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

Sending Directed UDP Datagrams

In the example at the beginning of this chapter, you sent UDP datagrams between Processing and Arduino. The Processing sketch sent a broadcast packet, and the Arduinos sent a directed packet in response, by extracting the sender's address from the packet. As long as you know the address to which you want to send a datagram , it's really that simple.

If you've got more than one Ethernet-connected device (besides your personal computer), you can try this out with the following Processing sketch. It's an expansion on the broadcast sketch, allowing you to send both broadcast and directed messages. Using the same Arduino UDP query responder from before, try it on all the receiving Arduino Ethernet-connected devices. When you type `a`, all the receiving devices will respond. When you type `b` or `c`, only the one with that address will respond.

Send It This sketch, along with the Arduino UDP query responder sketch, is a useful way to test whether your network devices are all working. It's a good idea to do something simple like this to make sure the network connections are good before you try more complex sketches.

```
/*
  UDP directed or broadcast query sender/receiver
 Context: Processing
 */

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

UDP udp;      // initialize the UDP object

void setup() {
  udp = new UDP( this, 43770 );  // open a UDP port
  udp.listen( true );            //  listen for incoming messages
}

void draw()
{
}
```

»

Continued from previous page.

```
void keyPressed() {
  int port = 43770;                          // the destination port
  String message = ", I'm talking to YOU!";  // the message to send
  String ip = "255.255.255.255";             // the remote IP address

  // send to different addresses depending on which key is pressed:
  switch (key) {
  case 'a':                      // broadcast query
    message = "Calling all ducks!\n";
    break;
  case 'b':                      // directed query
    ip = "192.168.1.20";         // the remote IP address
    message =  ip + message;     // the message to send
    break;
  case 'c':                      // directed query
    ip = "192.168.1.30";         // the remote IP address
    message =  ip + message;     // the message to send
    break;
  }

  // send the message to the chosen address:
  udp.send(message, ip, port );
}


void receive( byte[] data ) {
  // print the incoming data bytes as ASCII characters:
  for(int thisChar=0; thisChar < data.length; thisChar++) {
    print(char(data[thisChar]));
  }
  println();
}
```

You'll need to change these numbers.

Project 14

Relaying Solar Cell Data Wirelessly

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

This project comes from Gilad Lotan and Angela Pablo (as shown in Figure 7-11), former students at the Interactive Telecommunications Program (ITP) at New York University. The ITP is on the fourth floor of a 12-story building in Manhattan, and it maintains an 80-watt solar panel on the roof. The students wanted to watch how much useful energy the cell receives each day. Because it's used to charge a 12-volt battery, it's useful only when the output voltage is higher than 12V. In order to monitor the cell's output voltage on the fourth floor, Gilad and Angela (advised by a third student, Robert Faludi, who later wrote the book on XBee radios: **Building Wireless Sensor Networks** [O'Reilly]) arranged three XBee radios to relay the signal down the building's stairwell from the roof to the fourth floor. From there, the data went over the local network via an Ethernet processor and onto a SQL database. This example, based on their work, uses a smaller solar cell from Spark Fun, and a Processing program to graph the data instead of a SQL database.

There are three radios in this project: one attached to the solar cell, one relay radio standing on its own, and one attached to the Arduino. Figure 7-12 shows the network.

Radio Settings

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

Sensor radio	Relay radio	XPort radio
MY = 01 DL = 02 ID = 1111 D0 = 2 IR = 0x64 IT = 5	MY = 02 DL = 03 ID = 1111	MY = 03 DL = 01 ID = 1111

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

For the relay radio:

```
ATMY2, DL03, ID1111, WR\r
```

And for the Arduino radio:

```
ATMY3, DL01, ID1111, WR\r
```

The Circuits

The solar cell circuit runs off the solar cell itself, because the cell can produce the voltage and amperage in daylight needed to power the radio. The LD1117-33V regulator can take up to 15V input, and the solar panel's maximum output is 12V, so you're safe there. The MAX8212 IC is a voltage trigger. When the input voltage on its threshold pin goes above a level determined by the resistors attached to the circuit, the output pin switches from high to low. This change turns on the 2N3906 transistor. The transistor then allows voltage and current from the solar cell to power the regulator. When the solar cell isn't producing enough voltage, the radio will simply turn off. It's OK if the radio doesn't transmit when the cell is dark because there's nothing worth reporting then anyhow. The two resistors attached to the XBee's AD0 pin form a voltage divider that drops the voltage from the solar cell proportionally to something within the 3.3V range of the radio's analog-to-digital converter. The 4700µF capacitors store the charge from the solar cell like batteries, keeping the radio's supply constant. Figure 7-13 shows the circuit.



Figure 7-11
ITP students Gilad Lotan and Angela Pablo with the solar battery pack and XBee monitor radio.

Mesh Networking

The XBee radios can be configured as a mesh network using the ZigBee protocol. In a mesh network, some radios function as routers, similar to how the relay radio works in this project. Routers can not only relay messages, they can also store and forward them when the radios at the end node are not on. This provides the whole network with net power saving, as the end nodes can be turned off most of the time. To do this, though, you need different radios: Digi's XBee or XBee-PRO ZigBee OEM modules. For details on how to use these, see Robert Faludi's book Building Wireless Sensor Networks (O'Reilly).

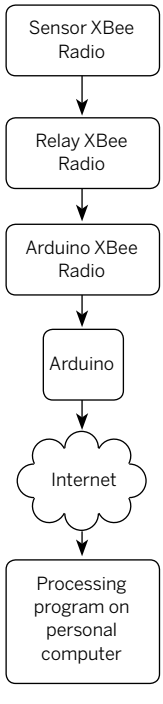


Figure 7-12
Network diagram for the solar project.

The Arduino radio circuit is identical to the one used in the previous project. Build it as shown in Figure 7-8.

The relay radio circuit is very simple. It's just a radio on a battery with its transmit pin and receive pin connected. This is how it will relay the messages. Any incoming messages will get sent out the serial transmit pin, then

MATERIALS

» **1 USB-to-XBee serial adapter**

Solar Cell Circuit

» **1 solderless breadboard**
» **1 Digi XBee OEM RF module**
» **1 3.3V regulator**
» **1 XBee breakout board**
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows**
» **1 1µF capacitor**
» **1 10µF capacitor**
» **3 4700µF electrolytic capacitors**
» **1 MAX8212 voltage monitor**
» **1 10-kilhom resistor**
» **3 100-kilhom resistors**
» **1 4.7-kilhom resistor**
» **1 1-kilhom resistor**
» **1 2N3906 PNP-type transistor**
» **2 LEDs**
» **2 220-ohm resistors**
» **1 solar cell**

Arduino Radio Circuit

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

» **1 Arduino microcontroller module**
» **1 Arduino Ethernet shield**
» **1 Digi XBee OEM RF module**
» **1 Arduino wireless shield** If you use this, you won't need the rest of the parts listed below. They are all for connecting the XBee to your Arduino.
» **1 solderless breadboard**
» **1 3.3V regulator**
» **1 1µF capacitor**
» **1 10µF capacitor**
» **1 XBee breakout board**
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows**
» **2 LEDs**
» **2 220Ω Resistors**

Relay Radio Circuit

This circuit is just an XBee radio by itself, powered by a battery. If you prefer, you can use the USB-to-XBee serial adapter instead of any of the three options below.

» **1 Digi XBee OEM RF module**
» **1 9V battery clip**
» **1 9V battery**

NOTE: If you use either of these, you won't need anything else but the XBee and the battery and a jumper wire to connect TX to RX.

Option 1:

» **1 XBee Explorer Regulated**

Option 2:

» **XBee LilyPad**

Option 3:

» **1 solderless breadboard**
» **1 3.3V regulator**
» **1 XBee breakout board**
» **2 rows of 0.1-inch header pins**
» **2 2mm female header rows**
» **2 LEDs**
» **2 220-ohm resistors**
» **1 10µF capacitor**
» **1 100µF capacitor**

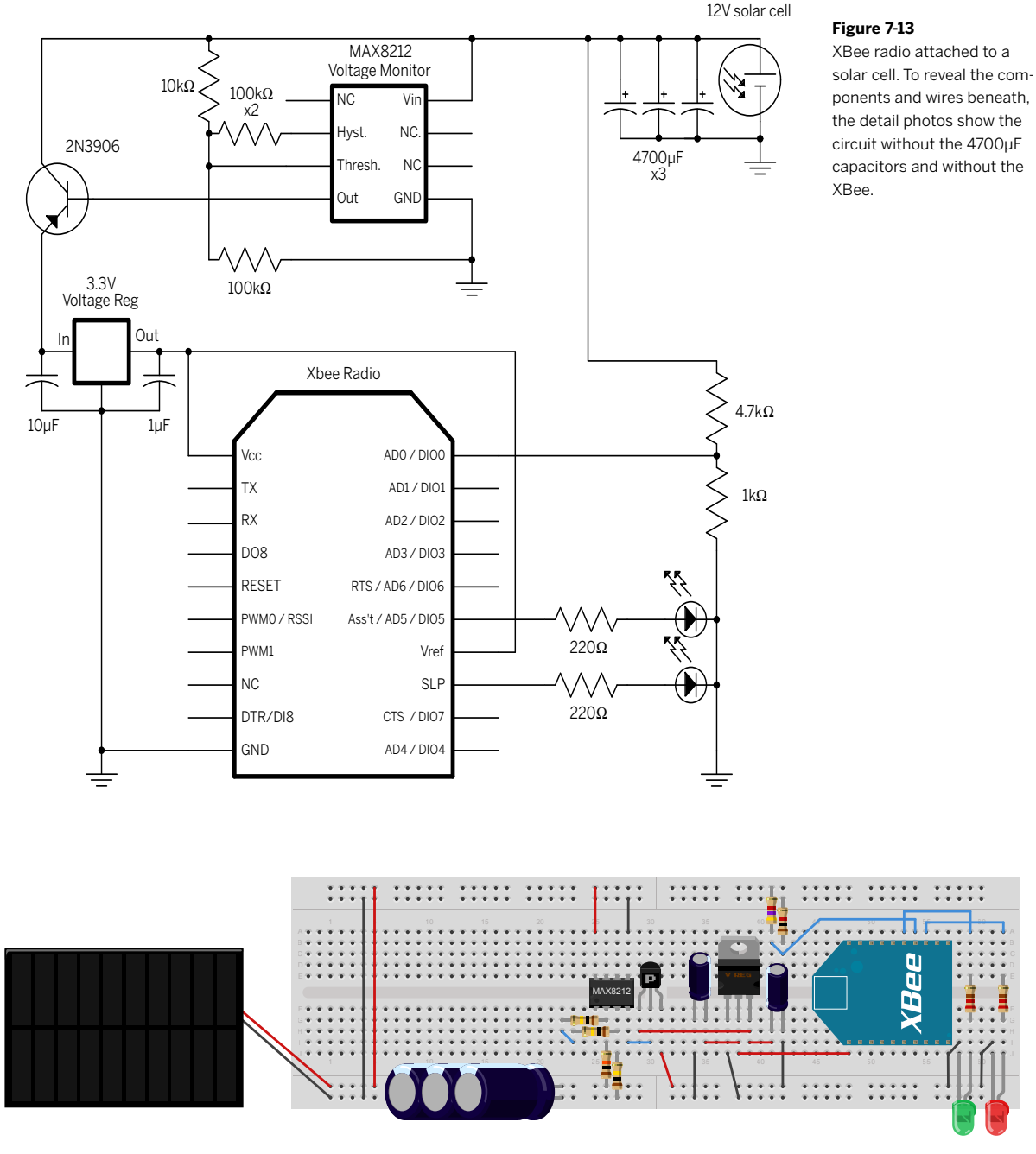


Figure 7-13
XBee radio attached to a solar cell. To reveal the components and wires beneath, the detail photos show the circuit without the 4700µF capacitors and without the XBee.

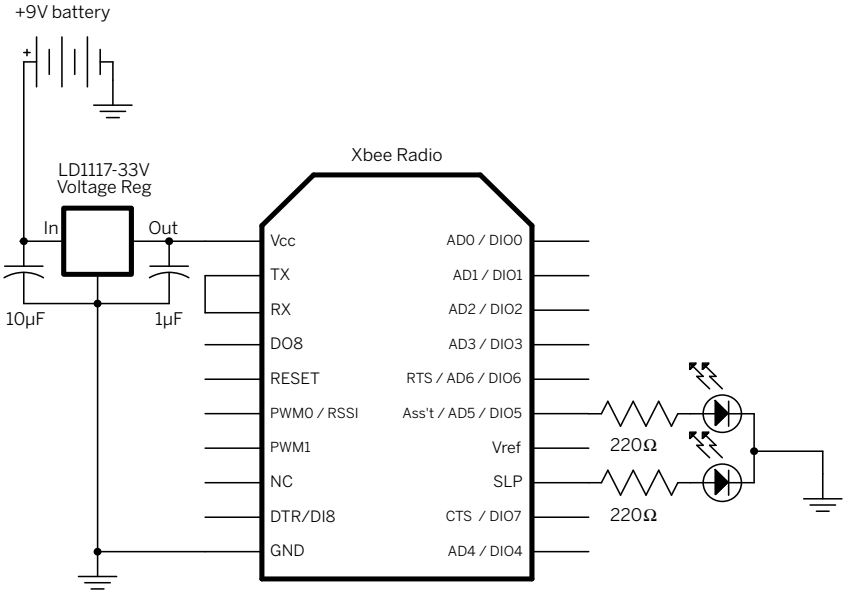
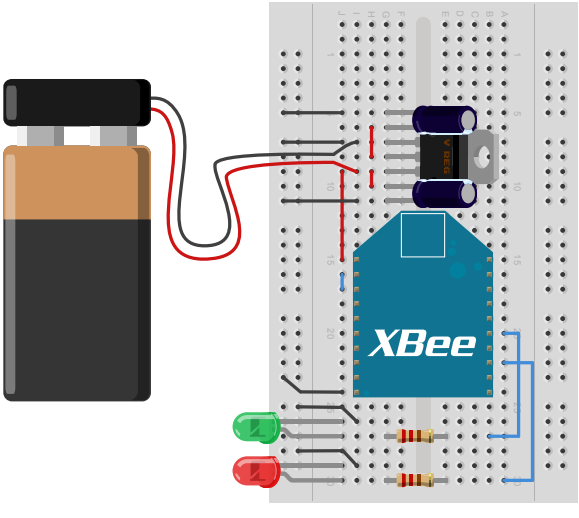


Figure 7-14
The XBee radio relay circuit.



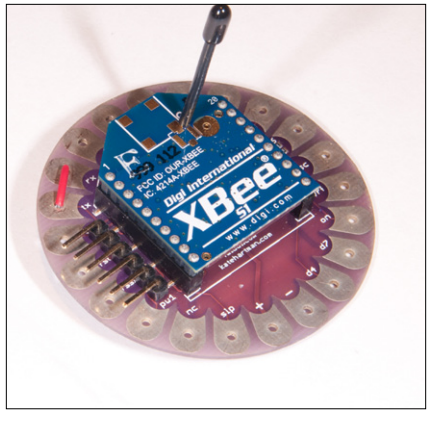


Figure 7-15
Alternative to the relay circuit: XBee LilyPad with transmit pin attached to receive pin.

back into the receive pin, where they will be sent out again as transmissions. Figure 7-14 shows the circuit. You could also use a LilyPad XBee, XBee Explorer Regulated, or any USB-to-XBee serial adapter. Just tie the transmit and receive pins together, and attach a battery to the voltage input of the regulator.

Once you've got the radios configured and working, you need to program the Arduino to read the incoming XBee packets and relay them to Processing. To do this, you'll use directed UDP datagrams.

Send It The Arduino sketch to read and send directed UDP datagrams is pretty simple. First, you need to include the relevant libraries and set up some global variables as usual. Notice that you're setting up a specific remote IP address and port—it should be the computer's address on which the Processing sketch will run.

```
/*
  XBee to UDP
 Context: Arduino
 */

#include <SPI.h>
#include <Ethernet.h>
#include <Udp.h>

// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = {
  0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x02 };
IPAddress myIp(192,168,1,20);
IPAddress yourIp(192,168,1,21);

unsigned int myPort = 43770;      // local port to listen on
unsigned int yourPort = 43770;    // remote port to send to

// A UDP instance to send and receive packets over UDP
UDP udp;
```

» You'll need to change these numbers.

» setup() starts the Ethernet connection, initializes serial communications, opens a UDP port, and starts an initial UDP packet for sending.

loop() just listens for serial input When it receives a byte of value 0x7E—indicating the beginning of a new XBee message—it sends the previous datagram using endPacket(), and begins a new packet. Finally, it adds any new byte to the current packet. That's it!

```
void setup() {
  // start the serial library:
  Serial.begin(9600);
  // start the Ethernet connection:
  Ethernet.begin(mac, myIp);
  // start UDP:
  udp.begin(myPort);
  // give the Ethernet shield a second to initialize:
  delay(1000);
  // set up a packet to send:
  udp.beginPacket(yourIp, yourPort);
}

void loop() {
  if (Serial.available()) {
    int serialByte = Serial.read();
    // if you get a 0x7E,
    // send the packet and begin a new one:
    if (serialByte == 0x7E) {
      udp.endPacket();
      // set up a packet to send:
      udp.beginPacket(yourIp, yourPort);
    }
    // send the byte:
    udp.write(serialByte);
  }
}
```

Graphing the Results

Now that all the hardware is ready, it's time to write a Processing sketch to graph the data. This sketch will receive UDP packets from the Arduino, parse the XBee packets contained therein, and graph the results. You'll notice that the parsing routine looks very similar to the parsing routine you wrote for the Arduino in the previous project. It's useful to compare them to see how the same algorithm is implemented slightly differently in the two programming languages.

X

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

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

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

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

UDP udp;                          // define the UDP object
int queryPort = 43770;            // the port number for the device query

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

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

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

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

Next, add a method to interpret the protocol. Not surprisingly, this looks a lot like the parsePacket() function from the Arduino sketch in the previous project. Add this method to the end of your program.

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

```
parseData(inString);
```

```
void parseData(int[] thisPacket) {
  // make sure the packet is 22 bytes long first:
  if (thisPacket.length >= 22) {
    int adcStart = 11;                    // ADC reading starts at byte 12
    int numSamples = thisPacket[8];       // number of samples in packet
    int[] adcValues = new int[numSamples]; // array to hold the 5 readings
    int total = 0;                        // sum of all the ADC readings

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

    // read the received signal strength:
    int signalStrength = thisPacket[6];

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

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

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

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

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

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

Call this from the parseData() method, replacing the println() statement that prints out the average, like so:

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

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

```
void drawGraph(int thisValue) {
  // draw the line:
  stroke(#4F9FE1);
  // map the given value to the height of the window:
  float graphValue = map(thisValue, 0, 1023, 0, height);
  // determine the line height for the graph:
  float graphLineHeight = height - (graphValue);
  // draw the line:
  line(hPos, height, hPos, graphLineHeight);
  // at the edge of the screen, go back to the beginning:
  if (hPos >= width) {
    hPos = 0;
    //wipe the screen:
    background(0);
  }
  else {
    // increment the horizontal position to draw the next line:
    hPos++;
  }
}
```

Next, add some code to add a time stamp. First, add a global variable to set the text line height.

```
int lineHeight = 14;          // a variable to set the line height
```

Add a method to the end of the program, drawReadings() . This will display the date, time, voltage reading, and received signal strength.

Call this method from a few different places in the program; first, at the end of the setup() method:

```
// show the readings text:
drawReadings(0,0);
```

Next, to draw the latest readings, call it at the end of the parseData() method:

```
// draw a line on the graph,
// and the readings:
drawGraph(average);
drawReadings(average, signalStrength);
```

```
void drawReadings(int thisReading, int thisSignalStrength) {
  // set up an array to get the names of the months
  // from their numeric values:
  String[] months = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
    "Sep", "Oct", "Nov", "Dec"
  };

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

  // format the time string
  // all digits are number-formatted as two digits:
  String time = nf(hour(), 2) + ":" + nf(minute(), 2)  + ":" + nf(second(),
2);

  // calculate the voltage from the reading:
  float voltage = thisReading * 3.3 / 1024;

  // choose a position for the text:
  int xPos = 20;
  int yPos = 20;
```

That's the whole program. When it's running, it should look like Figure 7-16.

Continued from previous page.

```
  // erase the previous readings:
  noStroke();
  fill(0);
  rect(xPos,yPos, 180, 80);
  // change the fill color for the text:
  fill(#4F9FE1);
  // print the readings:
  text(date, xPos, yPos + lineHeight);
  text(time, xPos, yPos + (2 * lineHeight));
  text("Voltage: " + voltage + "V", xPos, yPos + (3 * lineHeight));
  text("Signal Strength: -" + thisSignalStrength + " dBm", xPos,
          yPos + (4 * lineHeight));
}
```

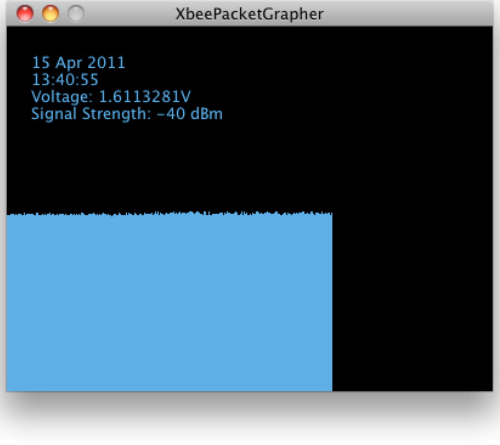


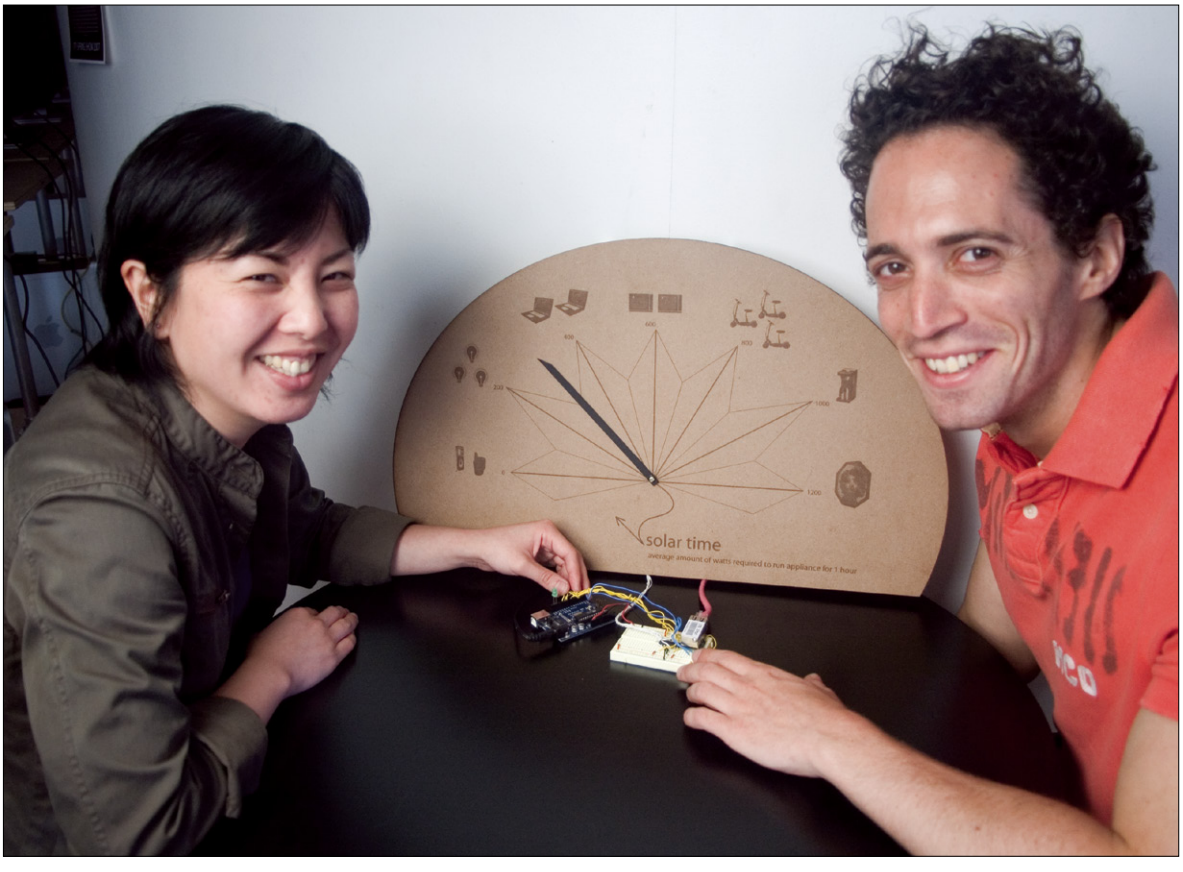
Figure 7-16
The output of the solar graph program. These sensor values were faked with a flashlight! Your actual values may differ.

Conclusion

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

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

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



The solar energy display by Gilad Lotan and Angela Pablo

The solar panel powering the display on the roof of NYU's Tisch School of the Arts





8

MAKE: PROJECTS 

How to Locate (Almost) Anything

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

◂◂ **Address 2007 by Mouna Andraos and Sonali Sridhar**

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

Supplies for Chapter 8

This chapter is all about sensing location, so most of the new parts are sensors. You'll also reuse the radios from Chapters 6 and 7.

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww/)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (http://farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **P** Pololu (www.pololu.com)
- **PX** Parallax (www.parallax.com)
- **RS** RS (www.rs-online.com)
- **SF** Spark Fun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 15: INFRARED DISTANCE-RANGER

» **1 Arduino module** An Arduino Uno or something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards.
D 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 Sharp GP2Y0A21 infrared ranger** This distance-ranging sensor operates on 5 volts, and outputs a 0–5V signal when it detects objects from 10 to 80cm away. It's one of a family of infrared-ranging sensors from Sharp. Available from many retailers, They are relatively inexpensive and very easy to use.
D 425-2063-ND, **SF** SEN-00242, **RS** 666-6570, **P** 136

» **1 3-wire JST connector pigtail** This cable allows you to attach the sensor to your microcontroller easily.
SF SEN-08733

» **1 10µF capacitor** **J** 29891, **D** P11212-ND, **F** 1144605, **RS** 715-1638

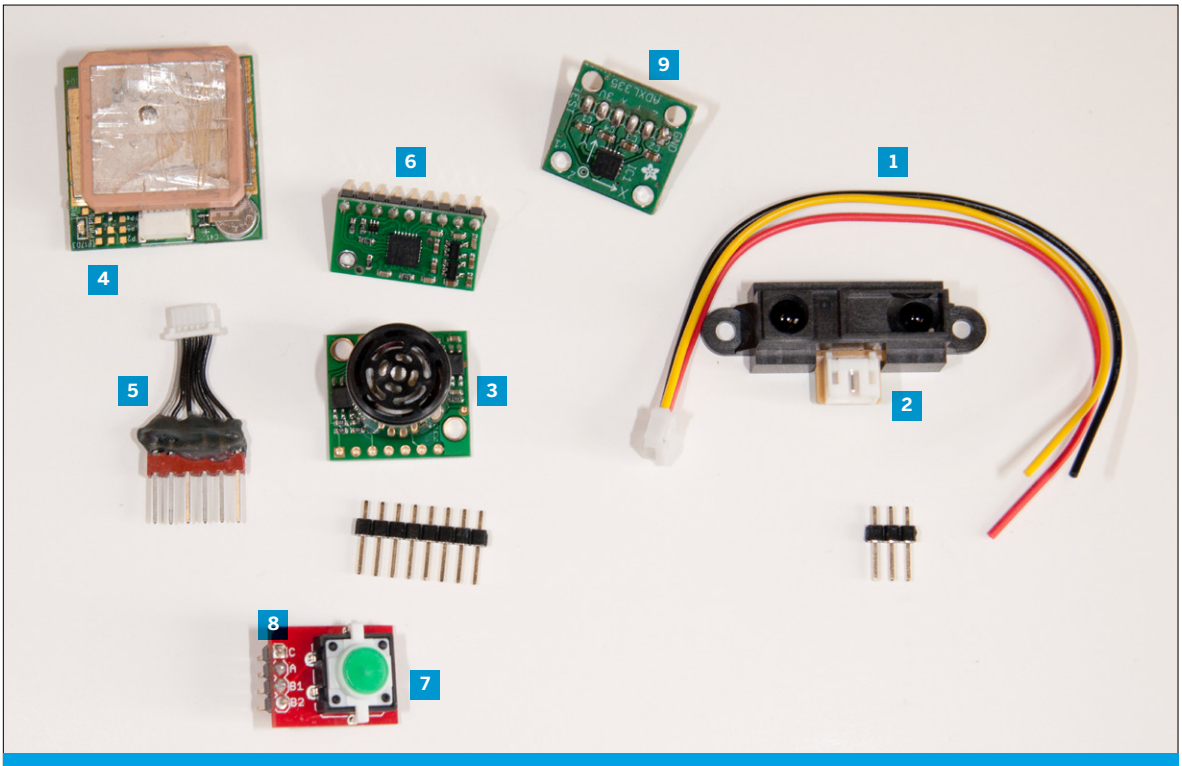


Figure 8-1. New parts for this chapter: **1.** Sharp GP20Y0A21 IR ranger **2.** JST pigtail for IR ranger **3.** Maxbotix LV-EZ1 ultrasonic ranger **4.** EM-406A GPS receiver **5.** Interface cable for EM-406A **6.** LMS303DLH digital compass **7.** LED tactile button **8.** Breakout board for button **9.** ADXL335 accelerometer. Don't forget plenty of male header pins for the breakout boards.

» **3 0.1-inch male header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411

PROJECT 16: ULTRASONIC DISTANCE-RANGER

» **1 Arduino module** See description in Project 15. **D** 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 MaxBotix LV-EZ1 ultrasonic ranger** Another distance-ranging sensor, this one uses sonar and has a longer range; 0 to 6.4m. **SF** SEN-00639, **AF** 172, **P** 726, **SS** SEN136B5B

» **3 0.1-inch male header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411

PROJECT 17: READING RECEIVED SIGNAL STRENGTH USING XBEE RADIOS

» **2 Digi XBee 802.15.4 RF modules** **J** 2113375, **SF** WRL-08664, **AF** 128, **F** 1546394, **SS** WLS113A4M, **MS** MKAD14

» **1 USB-XBee adapter** **J** 32400, **SF** WRL-08687, **AF** 247

» **XBee breakout board** Use the circuit you used for the XBee solar cell or gas monitor in Chapter 7. **J** 32403, **SF** BOB-08276, **AF** 127

PROJECT 18: READING RECEIVED STRENGTH USING BLUETOOTH RADIOS

» **1 Bluetooth Mate module** The same one used in Chapters 2 and 6. **SF** WRL-09358 or WRL-10393

» **1 USB-to-Serial adapter** **SF** DEV-09718 or DEV-09716, **AF** 70, **A** A000059, **M** MKAD22, **SS** PRO101D2P, **D** TTL-232R-3V3 or TTL-232R-5V

» **1 Bluetooth-enabled personal computer** If your laptop doesn't have a Bluetooth radio, use a USB Bluetooth adapter: **SF** WRL-09434, **F** 1814756

PROJECT 19: READING THE GPS SERIAL PROTOCOL

» **1 solderless breadboard** Just as you've used for previous projects. **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

» **1 EM-406A GPS receiver module** **S** GPS-00465, **PX** 28146, **AF** 99

» **1 interface cable for GPS module** **S** GPS-09123, **PX** 805-00013

» **1 Bluetooth Mate module** The same one used in Chapters 2 and 6. **SF** WRL-09358 or WRL-10393

» **12 0.1-inch male header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411

» **1 5V regulator** **J** 51262, **D** LM7805CT-ND, **SF** COM-00107, **F** 1860277, **RS** 298-8514

PROJECT 20: DETERMINING HEADING USING A DIGITAL COMPASS

» **1 solderless breadboard or prototyping shield** The same as used in previous projects. **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

» **1 Arduino module** See description in Project 15. **D** 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 ST Microelectronics LSM303DLH digital compass** Both Pololu and Spark Fun carry a module with this compass. As of this writing, the Pololu model is operable at 5V with no external components, but the Spark Fun one is not. **SF** SEN-09810, **RS** 717-3723, **P** 1250

» **1 LED tactile button** This example uses an LED tactile button and breakout board from Spark Fun, which has a built-in LED, but you can use any pushbutton and LED. **SF** COM-10443 and **SF** BOB-10467

» **1 220-ohm resistor** **D** 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842

» **1 10-kilohm resistor** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906

» **1 LED** Not needed if you are using the Spark Fun LED tactile button. **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590

» **13 0.1-inch male header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411

PROJECT 21: DETERMINING ATTITUDE USING AN ACCELEROMETER

» **1 solderless breadboard or prototyping shield** The same as used in previous projects. **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2

» **1 Arduino module** See description in Project 15. **D** 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 Analog Devices ADXL320 accelerometer** This is a three-axis analog accelerometer, the same one as you used in the balance board client project in Chapter 5. You can also use the accelerometer on your LSM303DLH digital compass from the previous project. **J** 28017, **SF** SEN-00692, **AF** 163, **RS** 726-3738, **P** 1247, **MS** MKPX7

Network Location and Physical Location

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

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

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

Step 1: Ask a Person

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

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

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

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

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

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

Step 2: Know the Environment

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

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

Street addresses are the most common coordinates that are mapped to latitude and longitude, but there are other systems for which it would be useful to have physical coordinates as well. For example, mobile phone cell transmitters all have physical locations. It would be handy to have a database of physical coordinates for those towers. However, cell towers are privately owned by mobile telephone carriers, so detailed data about the tower locations is proprietary, and the data is not in the public domain. Projects such as OpenCellID (www.opencellid.org) attempt to map cell towers by using GPS-equipped mobile phones running custom software. As there are many different mobile phone operating systems, just developing the software to do the job is a huge challenge.

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

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

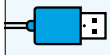
Step 3: Acquire and Refine

Once you know where you're going to look, there are two tasks that you have to do continually: acquire a new position, and refine the position's accuracy. Acquisition gives a rough position; it usually starts by identifying which device on a network is the center of activity. In the interactive installation example described earlier, you could acquire a new position by determining that the viewer tripped a sensor near one of the objects in the room. Once you know roughly where he is, you can refine the position by measuring his distance with the proximity sensor attached to the object.

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

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

X



35 Ways to Find Your Location

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

- **Assume: the Earth. Or a smaller domain, but assume that's the largest space you have to look in.**
- **Use the time.**
- **Ask someone.**
- **Association: who or what are you near?**
- **Proximity to phone boxes, public transport stops, and utility markings.**
- **Use a map.**
- **Which cell phone operators are available?**
- **Public phone operators?**
- **Phone number syntax?**
- **Newspapers available?**
- **Language being spoken?**
- **Post codes/ZIP codes.**
- **Street names.**
- **Street corners/intersections.**
- **Street numbers.**
- **Business names.**
- **Mobile phone location, through triangulation or trilateration.**
- **Triangulation and trilateration on other radio infrastructures, such as TV, radio, and public WiFi.**
- **GPS, assisted GPS, WAAS, and other GPS enhancements.**
- **Landmarks and "littlemarks."**
- **Dead reckoning.**

Determining Distance

Electronic locating systems—like GPS, mobile phone location, and sonar—seem magical at first because there's no visible evidence as to how they work. However, when you break the job down into its components, it becomes relatively straightforward. Most physical location systems are based on one of two methods: measuring the time of a signal's travel from a known location, or measuring its strength at the point of reception. Both methods combine measurements from multiple sources to determine a position in two or three dimensions using trilateration.

For example, a GPS receiver determines its position on the surface of the planet by measuring the time delay of received radio signals from several geosynchronous satellites. Mobile phone location systems function similarly, using the signal from nearby cell towers to determine the phone's position. Systems like Skyhook (www.skyhookwireless.com) use several different systems (WiFi, GPS, and cell tower location) to refine their positional accuracy. Sonar and infrared-ranging sensors work by sending out an acoustic signal (sonar) or an infrared signal (IR rangers), and then measuring the strength of that signal when it's reflected off the target.

Distance ranging techniques can be classified as active or passive. In active systems, the target has a radio, light, or acoustic source on it, and the receiver listens for the signal generated directly by the target. In passive systems, the target doesn't need to have any technology on board. The receiver emits a signal, then listens for the signal reflected back from the target. Mobile phone location is active because it relies on a two-way transmission between the phone and the cell tower. GPS is also active, even though the transmission is one-way, because the signal used to determine location is direct, not reflected. The target is a radio receiver. Sonar and infrared ranging are passive because the signal is reflected off the target, not generated by it.

Sometimes distance ranging is used for acquiring a position; other times, it's used for refining it. In the following examples, the passive distance rangers deliver a measurement of physical distance.

Passive Distance Ranging

Ultrasonic rangers like the MaxBotix LV-EZ1, and infrared rangers like the Sharp GP2Y0A21YK, shown in Figure 8-2, are examples of distance rangers. The MaxBotix sensor sends out an ultrasonic signal and listens for an echo. The Sharp sensor sends out an infrared light beam, and senses the reflection of that beam. These sensors work in a short range only. The Sharp sensor can read about 10cm to 80cm, and the MaxBotix sensor reads from about 0 to 7.5m. Passive sensors like these are handy when you want to measure the distance of a person in a limited space, and you don't want to put any hardware on the person. They're also useful when you're building moving objects that need to know their proximity to other objects in the same space as they move.

X

Project 15

Infrared Distance Ranger Example

The Sharp GP2*xx* series of infrared-ranging sensors give a decent measurement of short-range distance by bouncing an infrared light signal off the target, and then measuring the returned brightness. They're very simple to use. Figure 8-2 shows a circuit for a Sharp GP2Y0A21 IR ranger, which can detect an object in front of it within about 10cm to 80cm. The sensor requires 5V power, and it outputs an analog voltage from 0 to 5V, depending on the distance to the nearest object in its sensing area.

MATERIALS

- **1 Arduino module**
- **1 Sharp GP2Y0A21 IR ranger**
- **1 10µF capacitor**
- **3 male header pins**

The Sharp sensors' outputs are not linear, so if you want to get a linear range, you need to make a graph of the voltage over distance, and do some math. Fortunately, the good folks at Acroname Robotics have done the math for you. For the details, see www.acroname.com/robotics/info/articles/irlinear/irlinear.html. The sensor's datasheet at http://www.sharpsma.com/webfm_send/1208 includes a graph of voltage over the inverse of the distance. It shows a pretty linear relationship between the two from about 10 to 80cm, and the slope of that line is about 27V*cm. You can get a decent approximation of the distance using that.

» This sketch reads the sensor and converts the results to a voltage. Then, it uses the result explained above to convert the voltage to a distance measured in centimeters.

The conversion formula gives only an approximation, but it's accurate enough for general purposes.

For many applications, though, you don't need the absolute distance, but the relative distance. Is the person nearer or farther away? Has she passed a threshold that should trigger some other interaction? For such applications, you won't need this conversion. You can just use the output of the analogRead() command and choose a value for your threshold by experimentation.

```
/*
  Sharp GP2xx IR ranger reader
  Context: Arduino
*/
void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  int sensorValue = analogRead(A0);
  // convert to a voltage:
  float voltage = map(sensorValue, 0, 5, 0, 1023);

  // the sensor actually gives results that aren't linear.
  // This formula is derived from the datasheet's graph
  // of voltage over 1/distance. The slope of that line
  // is approximately 27:
  float distance = 27.0 /voltage;

  // print the sensor value
  Serial.print(distance);
  Serial.println(" cm");
  // wait 10 milliseconds before the next reading
  delay(10);
}
```

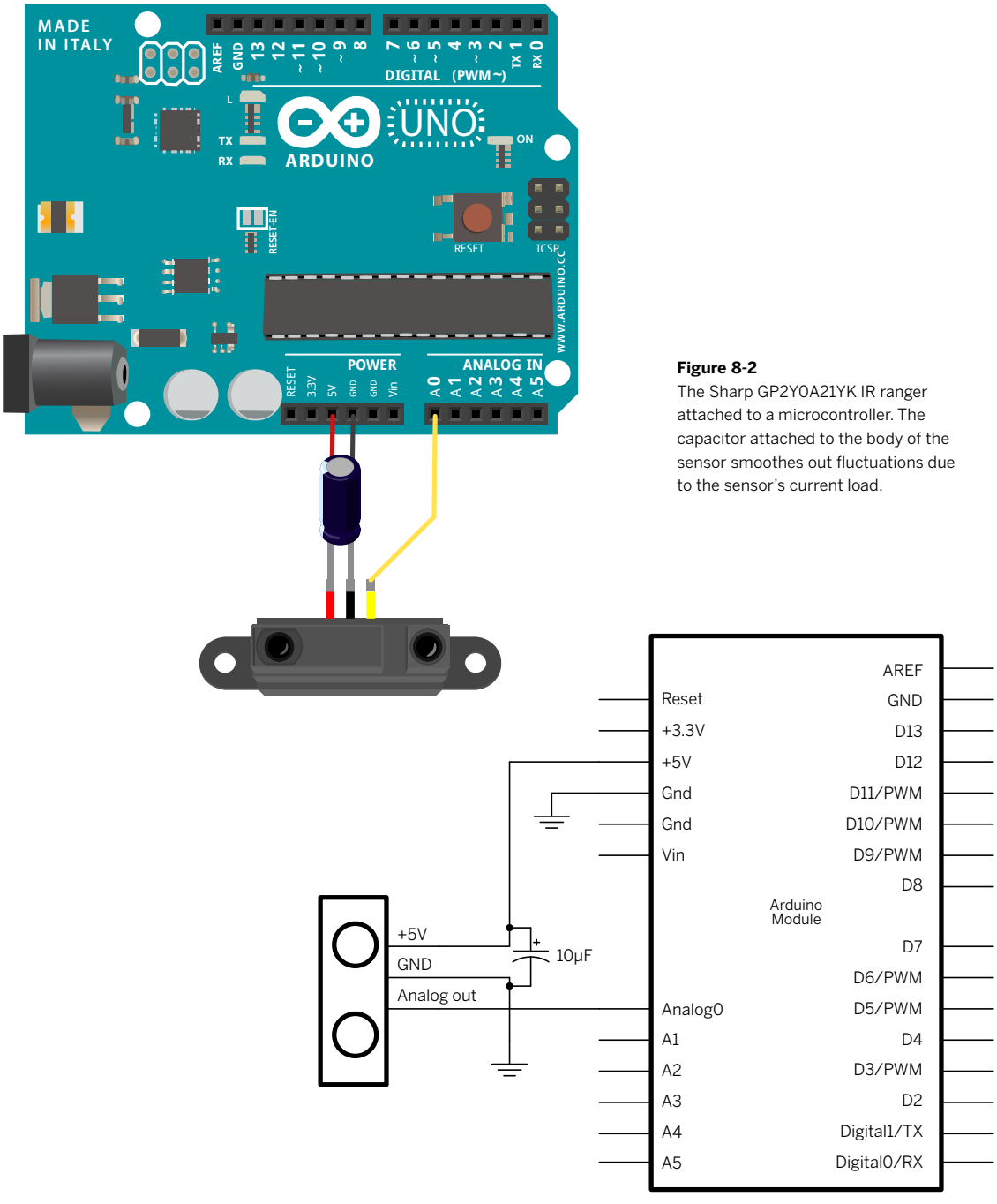


Figure 8-2
The Sharp GP2Y0A21YK IR ranger attached to a microcontroller. The capacitor attached to the body of the sensor smoothes out fluctuations due to the sensor's current load.

Project 16

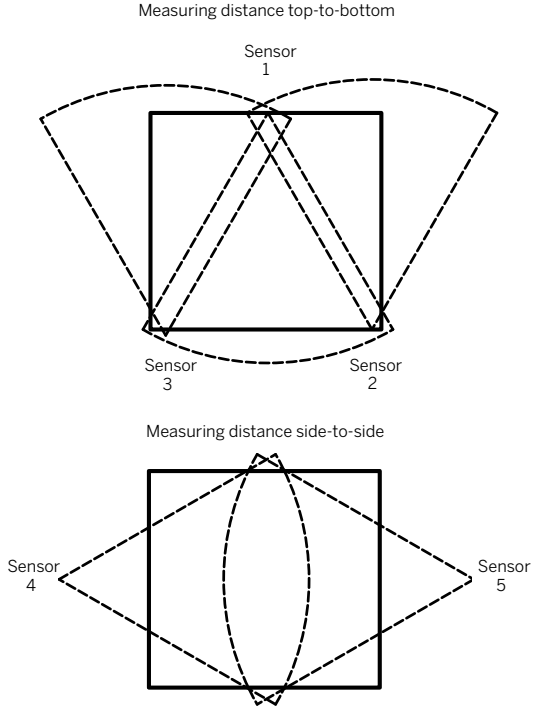
Ultrasonic Distance Ranger Example

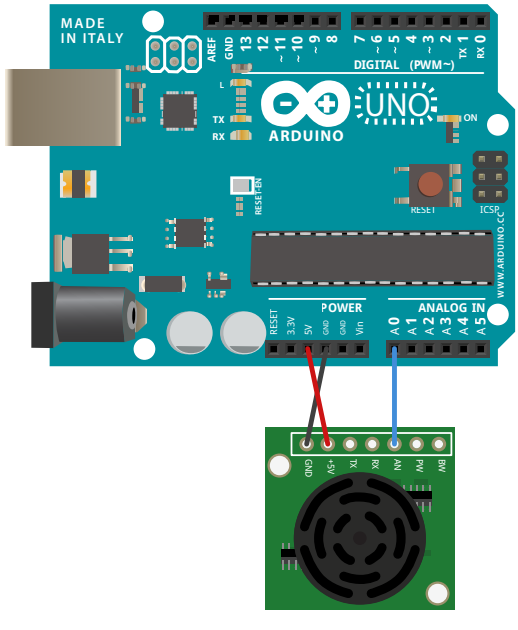
The MaxBotix (www.maxbotix.com) ultrasonic sensors measure distance using a similar method to the Sharp sensors, but theirs have a greater sensing range. Instead of infrared, they send out an ultrasonic signal and wait for the echo. Then they measure the distance based on the time required for the echo to return. These sensors require 5V power (the LV series can operate on 2.5–5V), and return their results via analog, pulse width, or asynchronous serial interface. They're available from MaxBotix, Spark Fun, Adafruit, Pololu, and many of the other retailers listed in this book.

MATERIALS

- **1 Arduino module**
- **1 MaxBotix LV-EZ1 ultrasonic ranger**
- **3 male header pins or 3 jumper wires**

Distance rangers are great for measuring linear distance, but they have a limited conical field of sensitivity, so they're not great for determining location over a large two-dimensional area. The MaxBotix LV-EZ1 sensor, for example, has a cone-shaped field of sensitivity that's about 80-degrees wide (though the sensitivity drops off at the edges) and 6.4 meters from the sensor to the edge of the range. In order to use it to cover a room, you'd need to use several of them and arrange them creatively. Figure 8-3 shows one way to cover a 4m x 4m space using five of the rangers. In this case, you'd need to make sure that no two of the sensors were operating at the same instant, because their signals would interfere with each other. The sensors would have to be activated one after another in sequence. Because each one takes up to 50 milliseconds to return a result, you'd need up to 250 milliseconds to make a complete scan of the space.





This sketch is similar to the infrared ranging sketch in Project 15. It reads the sensor and converts the results to a voltage, then converts that to a distance measured in centimeters. The conversion formula again gives only an approximation.

The MaxBotix sensors can read only every 50 milliseconds, so you need a delay after each read to give the sensor time to stabilize before the next read.

```
/*
 MaxBotix LV-EZ1 ultrasonic ranger reader
 Context: Arduino
*/
void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // read the sensor value and convert to a voltage:
  int sensorValue = analogRead(A0);
  float voltage = map(sensorValue, 0, 5, 0, 1023);

  // the sensor's output is about 9.8mV per inch,
  // so multiply by 2.54 to get it in centimeters:
  float distance = voltage * 2.54  / 0.0098;

  // print the sensor value
  Serial.print(distance);
  Serial.println(" cm");
  // wait 50 milliseconds before the next reading
  // so the sensor can stabilize:
  delay(50);
}
```

Figure 8-3
at left, opposite page
Measuring distance in two dimensions using ultrasonic distance rangers. The square in each drawing is a 4m × 4m floor plan of a room. In order to cover the whole of a rectangular space, you need several sensors placed around the sides of the room.

Figure 8-4
at right, opposite page
MaxBotix LV-EZ1 ultrasonic sensor connected to an Arduino module.

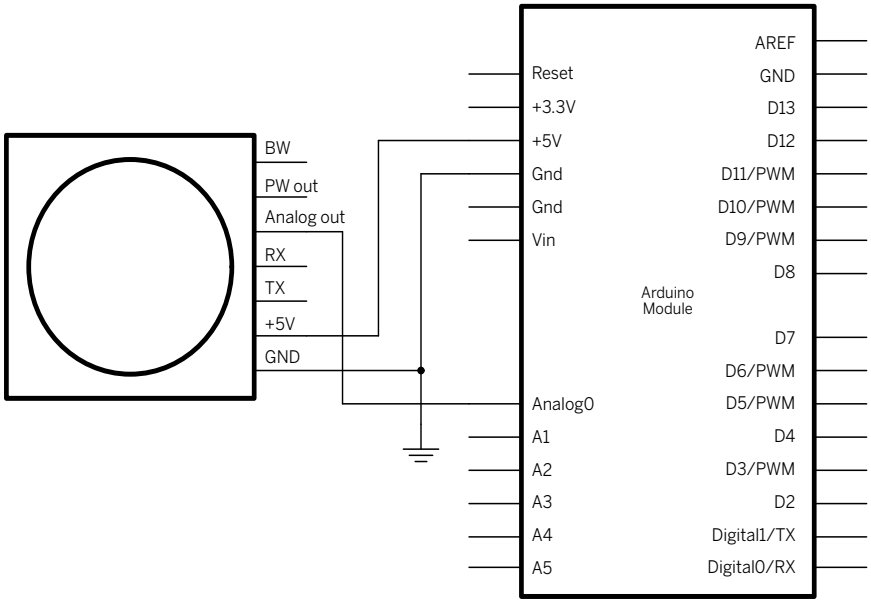


Figure 8-5
Schematic for MaxBotix LV-EZ1 ultrasonic sensor connected to an Arduino module.

Active Distance Ranging

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

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

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

X

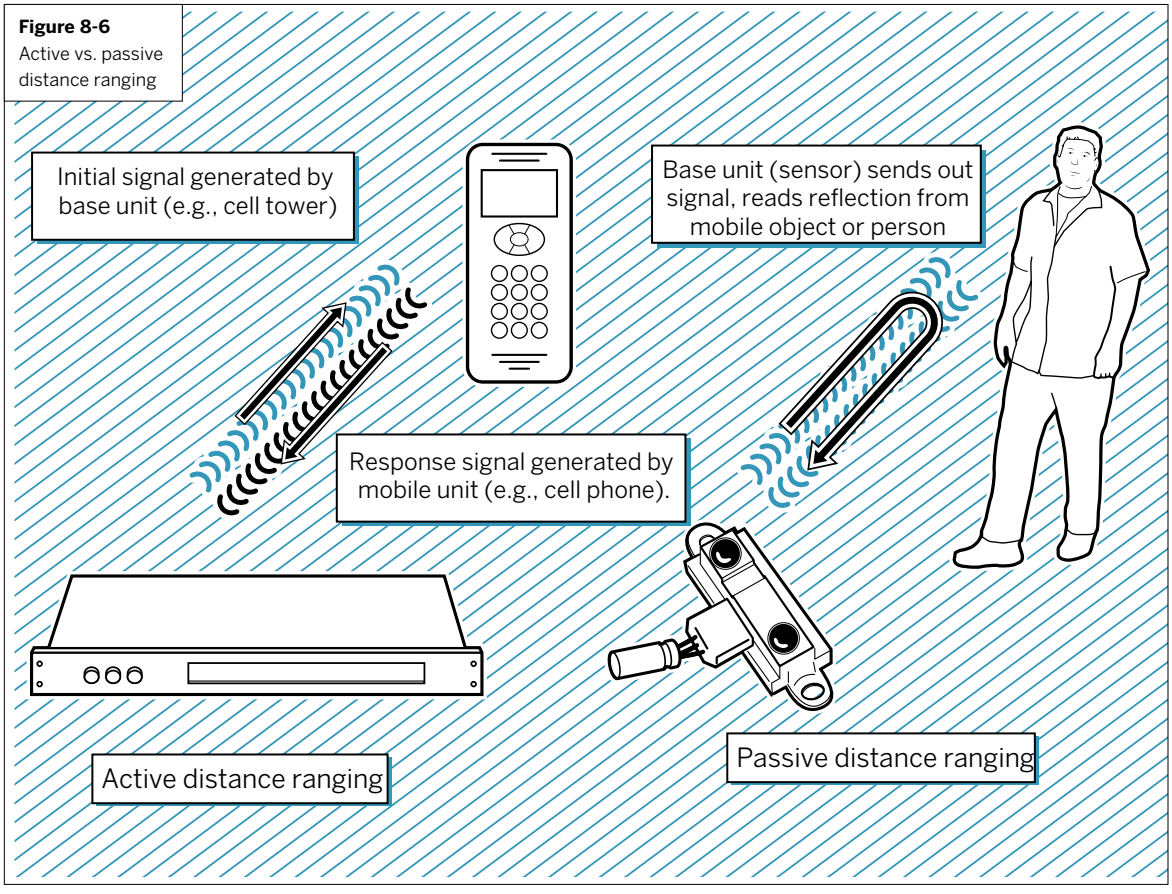


Figure 8-6
Active vs. passive distance ranging

Project 17

Reading Received Signal Strength Using XBee Radios

In the previous chapter, you saw the received signal strength, but you didn't do anything with it. The Processing code that read the solar cell's voltage output parsed the XBee packet for the received signal strength (RSSI). Here's a simpler variation on it that just reads the signal strength. To test it, you can use the same radio settings from Project 14, Relaying Solar Cell Data Wirelessly. Use a USB-to-XBee serial adapter for the receiving circuit, and see Figure 7-5 (the gas sensor circuit) or Figure 7-13 (the solar cell circuit) for circuits that work well as transmitters.

» Run this Processing sketch to connect to the receiver radio via the USB-to-XBee serial adapter. When you run this program, you'll get a graphing bar like that shown in Figure 8-7.

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

 Reads a packet from an XBee radio and parses it.  The packet
 should be 22 bytes long. It should be made up of the following:
 byte 1:     0x7E, the start byte value
 byte 2-3:   packet size, a 2-byte value  (not used here)
 byte 4:     API identifier value, a code that says what this response
             is (not used here)
 byte 5-6:   Sender's address
 byte 7:     RSSI, Received Signal Strength Indicator (not used here)
 byte 8:     Broadcast options (not used here)
 byte 9:     Number of samples to follow
 byte 10-11: Active channels indicator (not used here)
 byte 12-21: 5 10-bit values, each ADC samples from the sender
 */
import processing.serial.*;
Serial XBee ;                   // input serial port from the XBee Radio
int[] packet = new int[22];     // with 5 samples, the XBee packet is
                                // 22 bytes long
int byteCounter;                // keeps track of where you are in
                                // the packet
int rssi = 0;                   // received signal strength
int address = 0;                // the sending XBee's address
int lastReading = 0;            // value of the previous incoming byte

void setup () {
  size(320, 480);        // window size

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

»

Continued from previous page.

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

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

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

/*
 Once you've got a packet, you need to extract the useful data.
 This method gets the address of the sender and RSSI.
 */
int parseData(int[] thisPacket) {
  int result = -1;    // if you get no result, -1 will indicate that.

  // make sure you've got enough of a packet to read the data:
  if (thisPacket.length > 6) {
```



Continued from opposite page.

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

Radio signal strength is measured in decibel-milliwatts (dBm). You might wonder why the signal reads -65dBm. How can the signal strength be negative? The relationship between milliwatts of power and dBm is logarithmic. To get the dBm, take the log of the milliwatts. So, for example, if you receive 1 milliwatt of signal strength, you've got log 1 dBm. Log 1 = 0, so 1 mW = 0 dBm. When the power drops below 1 mW, the dBm drops below 0, like so: 0.5 mW = (log 0.0005) dBm or -3.01 dBm. 0.25mW = (log 0.00025) dBm, or -6.02 dBm.

If logarithms confuse you, just remember that 0 dBm is the maximum transmission power, which means that signal strength is going to start at 0 dBm and go down from there. The minimum signal that the XBee radios you're using here can receive is -92 dBm. Bluetooth radios and WiFi radios typically have a similar range of sensitivity. In a perfect world, with no obstructions to create errors, the relationship between signal strength and distance would be a logarithmic curve.

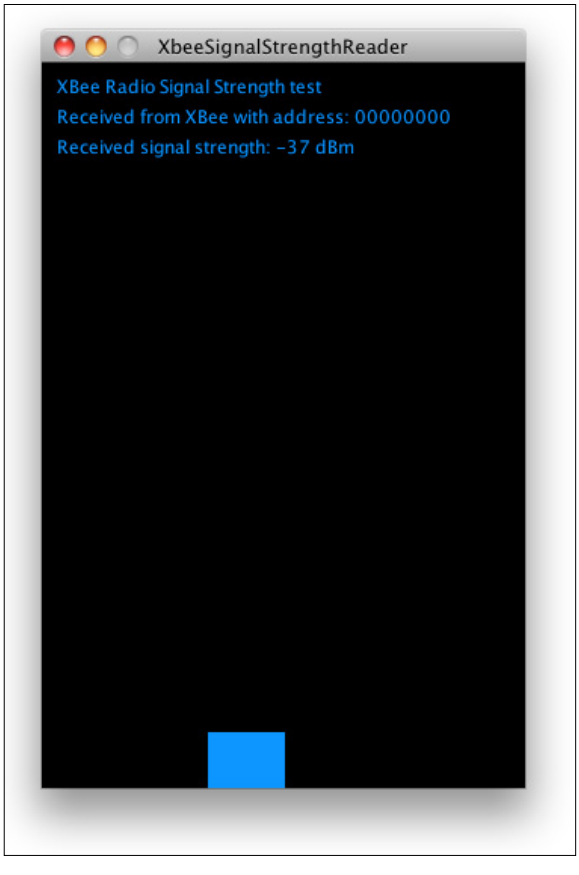


Figure 8-7
Output of the XBee RSSI test program.

Project 18

Reading Received Signal Strength Using Bluetooth Radios

The Bluetooth modules used in Chapters 2 and 6 can also give you an RSSI reading. To see this, the radio needs to be connected to another Bluetooth radio. The simplest way is to pair your radio with your computer, as shown in Project #4.

Once you've done this, open a serial connection to the radio via Bluetooth. Once you're connected, drop out of data mode into command mode by typing the following:

```
$$$
```

You'll get a CMD prompt from the radio. Next, type L and hit Return. The radio will respond like so:

```
RSSI=ff,ff
RSSI=ff,ff
```

These are the signal strength values of the link, in hexadecimal. FF is the strongest possible value, and 00 is the weakest. The first of the two values is the current link quality; the second is the lowest recorded value so far. As you move the radio closer to or farther from your computer, the values will change just as it did in the XBee example in the preceding project. To turn this off and get back to sending data, type L and hit Return again. Then, type `---` and press Return to leave command mode.

The Multipath Effect

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

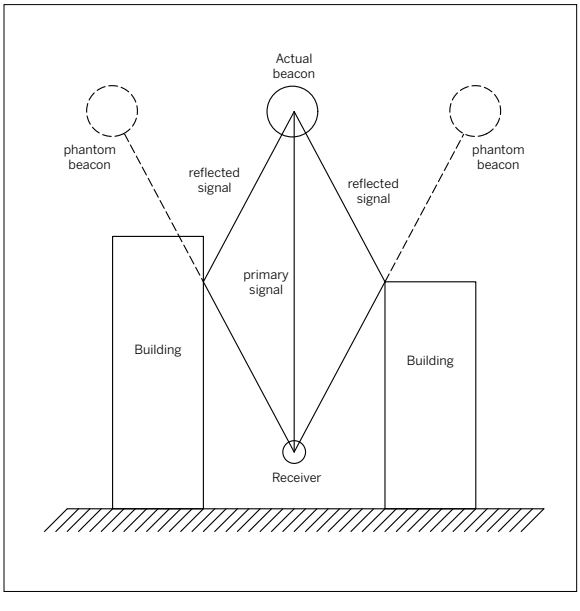


Figure 8-8
The multipath effect. Reflected radio waves create phantom beacons that the receiver can't tell from the real beacon, causing errors in calculating the distance based on signal strength.

Determining Position Through Trilateration

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

In order to locate it within a two- or three-dimensional space, though, you need to know more than distance. The most common way to do this is by measuring the distance from at least three points. This method is called trilateration. If you measure the object's distance from two points, you get two possible places it could be on a plane, as shown in Figure 8-9. When you add a third circle, you have one distinct point on the plane where your object could be. A similar method, triangulation, uses two known points and calculates the position using the distance between these points; it then uses the angles of the triangle formed by those points and the position you want to know.

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

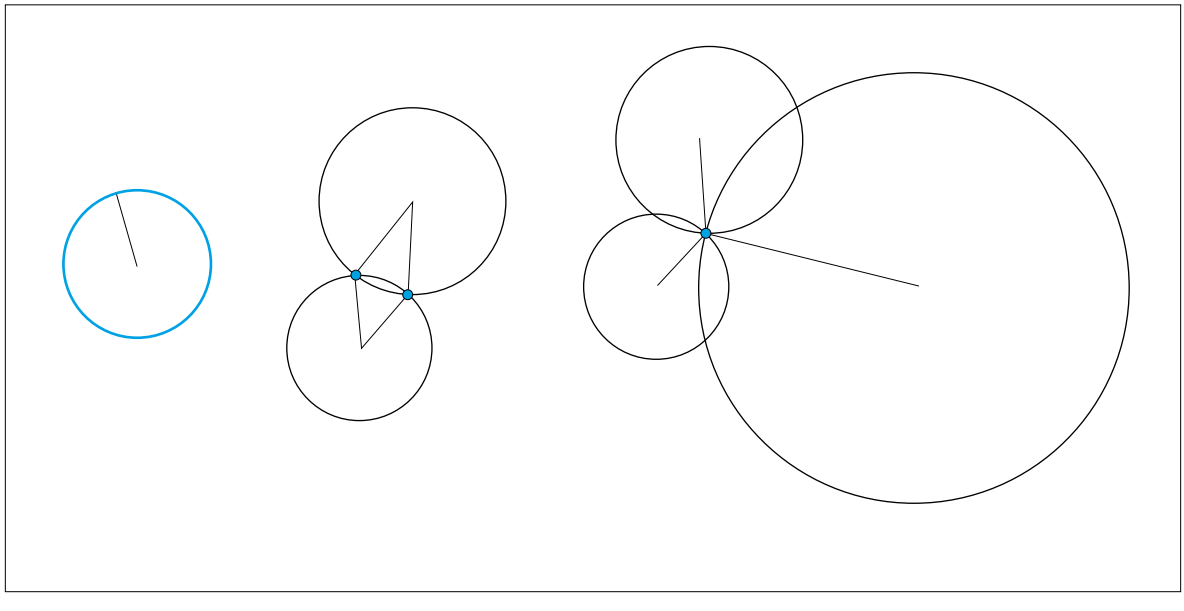


Figure 8-9
Trilateration on a two-dimensional plane. Knowing the distance from one point defines a circle of possible locations. Knowing the distance from two points narrows it to two possible points on the plane. Knowing the distance from three points determines a single point on the plane.

Project 19

Reading the GPS Serial Protocol

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

MATERIALS

- **1 solderless breadboard**
- **1 EM-406A GPS receiver**
- **1 interface cable for GPS receiver**
- **12 male header pins**
- **1 Bluetooth Mate**
- **1 5V voltage regulator**

NMEA 0183 is a serial protocol that operates at 4800 bits per second, 8 data bits, no parity, and 1 stop bit (4800-8-N-1). Most receivers send this data using either RS-232 or TTL serial levels. The receiver used for this example, a US GlobalSat EM-406a receiver, sends NMEA data at 5V TTL levels.

You're going to connect the GPS receiver to a Bluetooth Mate for this project. Before you do, however, you need to match their data rates by resetting the Bluetooth Mate to 4800bps. To do this, connect your Mate to a USB-to-Serial adapter and open a connection to it in a serial terminal application (like CoolTerm or PuTTY) at 115200 bits per second (see Project 4 in Chapter 2 for more details). First, type: `$$$`. This command takes the radio out of data mode and puts it in command mode. The radio will respond: `CMD`

Next, type:

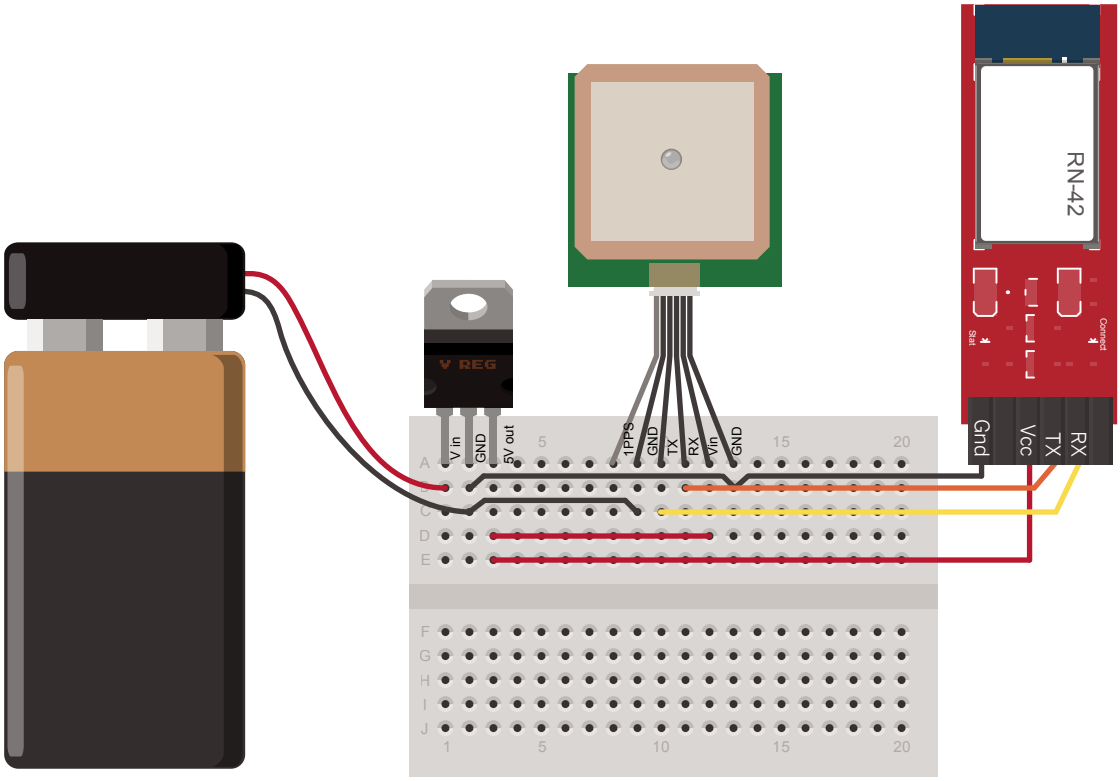
`SU,48\r` ▸▸ This sets the radio to 4800 bps.

`R,1\r` ▸▸ This reboots the radio.

Your radio will save the new setting at 4800bps and reboot. Now you can connect it to the GPS module.

Figure 8-10 shows the module connected to a Bluetooth Mate radio. The GPS data will be sent over Bluetooth to a personal computer running Processing. If it's not already paired, you can pair the Bluetooth Mate to your personal computer using the instructions from Project 3 in Chapter 2. That will give you a Bluetooth serial port in your list of ports, as in that project. Open a connection to that port in a serial terminal application, and you should see data in the NMEA protocol, like what you see below.

```
$GPGGA,180226.000,4040.6559,N,07358.1789,W,1,04,6.6,75.4,M,-34.3,M,,0000*5B
$GPGSA,A,3,12,25,09,18,,,,,,,,,6.7,6.6,1.0*36
$GPGSV,3,1,10,22,72,171,,14,67,338,,25,39,126,39,18,39,146,35*70
$GPGSV,3,2,10,31,35,228,20,12,35,073,37,09,15,047,29,11,09,302,20*7D
$GPGSV,3,3,10,32,04,314,17,27,02,049,15*73
$GPRMC,180226.000,A,4040.6559,N,07358.1789,W,0.29,290.90,220411,,*12
$GPGGA,180227.000,4040.6559,N,07358.1789,W,1,04,6.6,75.4,M,-34.3,M,,0000*5A
$GPGSA,A,3,12,25,09,18,,,,,,,,,6.7,6.6,1.0*36
$GPRMC,180227.000,A,4040.6559,N,07358.1789,W,0.30,289.06,220411,,*1C
```



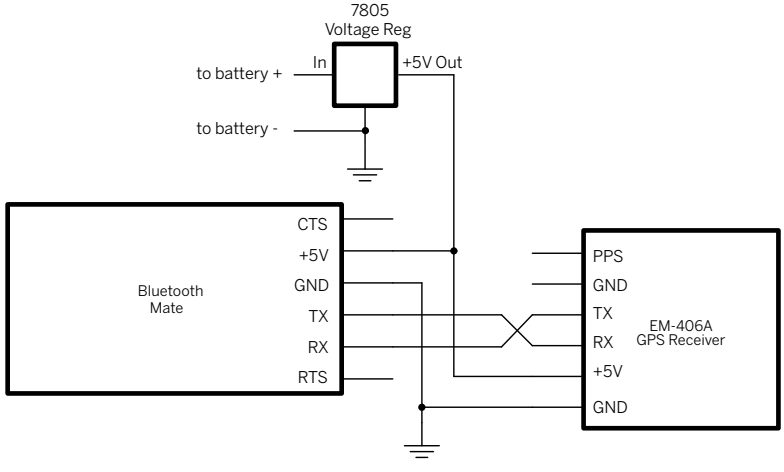


Figure 8-10
EM-406a GPS receiver attached to a Bluetooth radio. In order to get a real GPS signal, you'll have to go outside, so wireless data and a battery power source are handy.

NOTE: If your power is inconsistent, add in the decoupling capacitors on the input and output sides of the regulator, as seen in other projects using this regulator.

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

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

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

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

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

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

The Processing sketch shown next reads NMEA serial data in and parses out the time, date, latitude, longitude, and heading. It uses the $GPRMC sentence to get basic info, the $GPGSV sentence to get the satellites in view, and the $GPGGA sentence to get the number of satellites used to obtain a fix. It draws an arrow on the screen to indicate heading. The output looks like Figure 8-11. (Be sure to use the Bluetooth serial port when opening the serial port!)

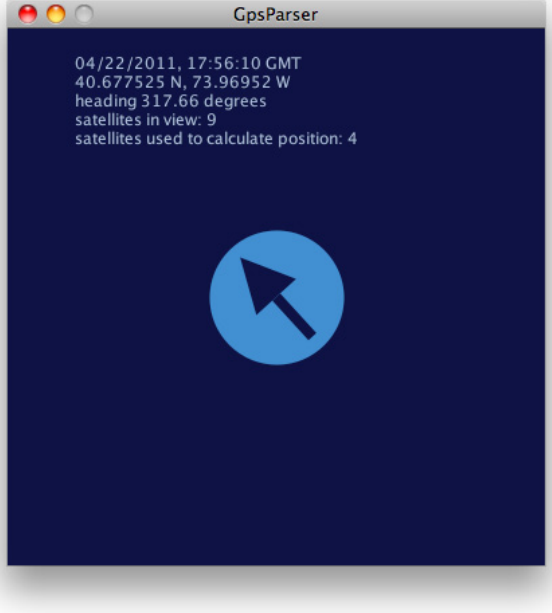


Figure 8-11
The output of the Processing GPS parser.

NOTE: Extra credit. Figure out where I was when I wrote this chapter. Thanks to reader Derrick O'Brien who figured out from the first edition that there was an error in my calculations, and introduced me to Glen Murphy's excellent GPS processing example at http://bodytag.org/p5gps/p5gps.pde**. With the changes in this edition, it will be easier to find me.**

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

```
/*
 GPS parser
 Context: Processing

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

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

int hrs, mins, secs;      // time units
int currentDay, currentMonth, currentYear;

int satellitesInView = 0;  // satellites in view
int satellitesToFix = 0;   // satellites used to calculate fix

float textX = 50;          // position of the text on the screen
float textY = 30;
```

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

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

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

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

  // Open whatever port is the one you're using.
  // for a Bluetooth device, this may be further down your
  // serial port list:
  String portName = Serial.list()[6];
  myPort = new Serial(this, portName, 9600);

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

» You will probably need to look at the output of Serial.list() and change this number to match the serial port that corresponds to your Bluetooth device.

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

```
void draw() {
  // deep blue background:
  background(#0D1133);

  // pale slightly blue text:
  fill(#A3B5CF);

  // put all the text together in one big string:

  // display the date and time from the GPS sentence
  // as MM/DD/YYYY, HH:MM:SS GMT
  // all numbers are formatted using nf() to make them 2- or 4-digit:
  String displayString = nf(currentMonth, 2)+ "/"+ nf(currentDay, 2)
    + "/"+ nf(currentYear, 4) + ", " + nf(hrs, 2)+ ":"
    + nf(mins, 2)+ ":"+ nf(secs, 2) + " GMT\n";


  // display the position from the GPS sentence:
  displayString = displayString + latitude + " " + northSouth + ", "
    + longitude + " " + eastWest + "\n";

  // display the heading:
  displayString = displayString + "heading " + heading + " degrees\n";

  // show some info about the satellites used:
  displayString = displayString + "satellites in view: "
    + satellitesInView + "\n";
  displayString = displayString + "satellites used to calculate position: "
    + satellitesToFix;
  text(displayString, textX, textY);

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

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

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

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

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

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

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

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

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

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

  if (items[0].equals("$GPGGA")) {
    // $GPGGA gives time, date, position, satellites used
    getGGA(items);
  }

  if (items[0].equals("$GPGSV")) {
    // $GPGSV gives satellites in view
    satellitesInView = getGSV(items);
  }
}
```

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

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

  // if you have a valid reading, parse the rest of it:
  if (data[2].equals("A")) {
    latitude = minutesToDegrees(float(data[3]));

    northSouth = data[4];
    longitude = minutesToDegrees(float(data[5]));
    eastWest = data[6];
    heading = float(data[8]);
    int date = int(data[9]);
```

Continued from previous page.

```
    // last two digits of the date are year.  Add the century too:
    currentYear = date % 100 + 2000;
    // second two digits of the date are month:
    currentMonth =  (date % 10000)/100;
    // first two digits of the date are day:
    currentDay = date/10000;
  }
}
```

The getGGA() method parses out the parts of the $GPGGA sentence. Some of it is redundant with the $GPRMC sentence, but the number of satellites to fix a position is new.

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

  // if you have a valid reading, parse the rest of it:
  if (data[6].equals("1")) {
    latitude = minutesToDegrees(float(data[2]));
    northSouth = data[3];
    longitude = minutesToDegrees(float(data[4]));
    eastWest = data[5];
    satellitesToFix = int(data[7]);
  }
}
```

The getGSV() method parses out the parts of the $GPGSV sentence. It returns a single integer—the number of satellites in view.

```
int getGSV(String[] data) {
  int satellites = int(data[3]);
  return satellites;
}
```

Finally, the minutesToDegrees() method is used by both the getRMC() and getGGA() methods. The NMEA protocol sends latitude and longitude like this:

```
ddmm.mmmm
```

where dd is degrees, and mm.mmm is minutes. This method converts the minutes to a decimal fraction of the degrees.

```
float minutesToDegrees(float thisValue) {
  // get the integer portion of the degree measurement:
  int wholeNumber = (int)(thisValue / 100);
  // get the fraction portion, and convert minutes to a decimal fraction:
  float fraction = (thisValue - ( wholeNumber ) * 100) / 60;
  // combine the two and return it:
  float result = wholeNumber + fraction;
  return result;
}
```

When you run this sketch, it may take several minutes before you get a position. Different receivers take varying times to acquire an initial position when first started, or when moved to a new place on the planet. So, pick a spot with a clear view of the skies and be patient. Pay attention to the number of satellites in view as well. If that number is less than four, your receiver's not going to acquire a position too well. Satellites move, though, so if you don't get anything, wait a half-hour or so and try again.

Many mobile phones on the market these days feature a GPS receiver, and seem to have a signal most of the time, which may make you wonder why your receiver can't get a signal as fast. Remember, mobile phones tend to use both GPS and cellular location tracking together. So, even if they don't have enough satellite signals, they can generally determine their position relative to the nearest cell towers, and—using the known locations of those towers—approximate a fix.

NMEA 0183 is just one of many protocols used in mapping GPS data. You'll learn about a few more, as well as some tools for using them to map routes and locations, in Chapter 11. The great thing about NMEA, though, is that it's nearly ubiquitous among GPS receivers. No matter what other protocols they use, they all seem to have NMEA as an option. So it's a good one to know about no matter what GPS tools you're working with.
X

Choosing Which GPS Accessories to Buy

There are many GPS receiver modules on the market, and it can get confusing choosing the right tools for this job. Here are a few things to consider.

Most common GPS receivers communicate with the microcontroller via TTL serial, using the NMEA 0183 protocol. So when it comes to connecting them to your microcontroller, and reading their data, they're largely interchangeable. The big difference between them is how well they receive a GPS signal, which is influenced by how many channels they receive (generally more is better), what kind of antenna they have (generally larger is better), and how much power they consume. I prefer the EM-406a receiver mentioned in this project because it gets good reception, acquires a fix fast, and has been the most reliable compared to others I've tested. There are other good ones, though. Spark Fun has a nice GPS receiver buying guide on their site.

To connect a GPS receiver, all you need is a serial transmit connection, power, and ground. There are some shields available that allow you to mount a receiver to your Arduino, but they're optional. You need only three wires.

Images courtesy of Spark Fun. Thanks to the students of 2011's Wildlife Tracking class for confirming my tests.
X



EM-406a GPS receiver. 20 channels, good reception, less expensive than other alternatives.



D2523T GPS receiver. 50 channels, great reception, but more expensive.



LS20126 receiver. Has a very small antenna, but works decently in open areas.



GPS MiniMod with GR10/MN1010 receiver. Smallest I could find. Takes a long time to acquire a signal.

Determining Orientation

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

Project 20

Determining Heading Using a Digital Compass

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

MATERIALS

- **1 solderless breadboard or prototyping shield**
- **1 Arduino module**
- **1 digital compass, ST Microelectronics model LSM303DLH**
- **1 LED tactile pushbutton** This example uses an LED tactile button from Spark Fun, which has a built-in LED, but you can use any pushbutton and LED
- **1 220-ohm resistor**
- **1 10-kilohm resistor**
- **13 male header pins**

This example uses a digital compass from ST Microelectronics, model LSM303DLH. Both Spark Fun and Pololu carry breakout boards for this compass. The Pololu version was used for this example because it has built-in voltage level shifters and works better at 5V. It measures magnetic field strength along three axes, and it has a three-axis accelerometer built in as well to help compensate for tilt. It reports the results via synchronous serial data sent over an I2C connection via the Wire library.

Figure 8-12 shows the compass connected to an Arduino. The compass requires calibration, so this project features a pushbutton to toggle between calibration mode and normal mode, and an LED to indicate when you're calibrating.

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

1. 1V8: 1.8V output. You won't use this pin.
2. 3V: 3-volt output. You won't use this pin.
3. Vin: 5-volt input. Connect to the microcontroller's 5V.
4. GND: ground. Connect to the microcontroller's ground.
5. SCL: Serial clock. Connect to microcontroller's SCL pin (analog pin 5).
6. SDA: Serial data. Connect to microcontroller's SDA pin (analog pin 4).
7. DRDY: Data ready indicator: outputs 1.8V when the compass is ready to be read. You won't use this pin.
8. INT1: interrupt 1. You won't use this pin.
9. INT2: interrupt 2. You won't use this pin.

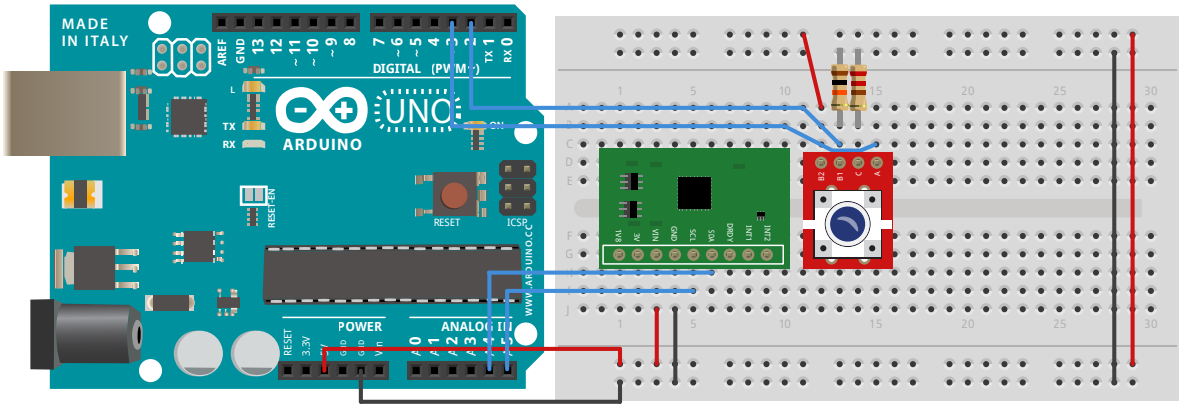
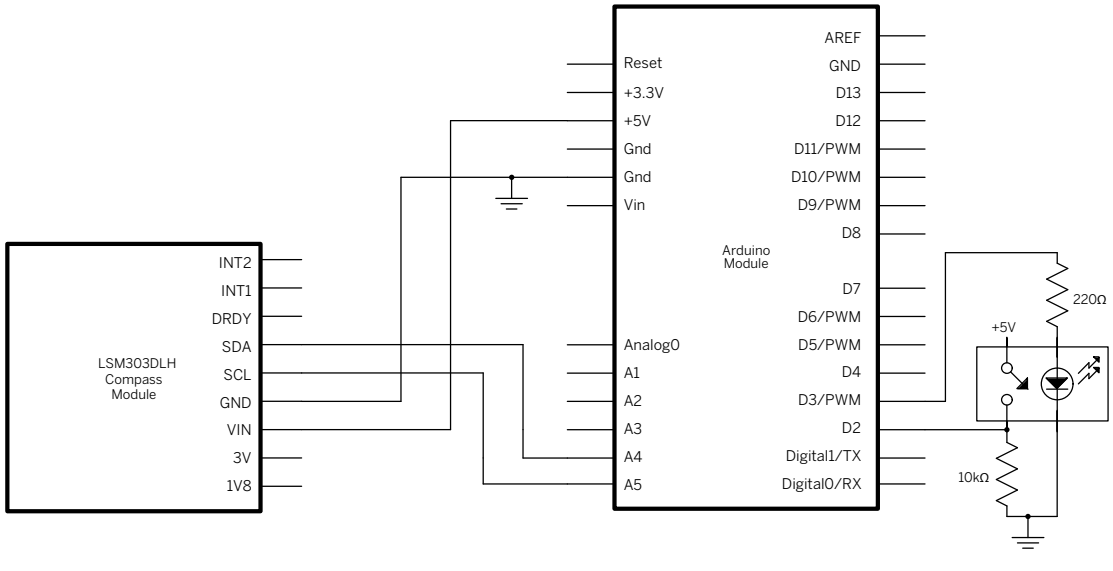


Figure 8-12
ST Microelectronics LSM303DLH compass connected to an Arduino.

NOTE: To calibrate the compass, rotate it slowly through 360 degrees on a flat, level surface while in calibration mode (see code). When calibrating the compass, you need to know the cardinal directions precisely. Get a magnetic needle compass and check properly. You should calibrate away from lots of electronic equipment and sources of magnetic energy (except the earth). For example, in my office, needle compasses tend to point west-southwest, so I calibrate outside, powering the whole Arduino circuit from a battery. To calibrate the accelerometer, rotate the module through every possible axis.



This sketch uses the Wire library to communicate via I2C with the compass. The Wire library is encapsulated in another library, the LSM303DLH library, originally written by Ryan Mulligan of Pololu. I've made a variation on it with a few extra functions, available at http://github.com/tigoe/LSM303DLH. Download version 1.1.0 (the latest as of this writing) and copy the **LSM303DLH** folder into the **libraries** directory of your Arduino sketch directory. Note that there is a folder called **LSM303DLH** *inside* the folder you download. It's the inner folder that you want.

This sketch also uses Alexander Brevig's Button library; the current Wiring version is at http://wiring.uniandes.edu.co/source/trunk/wiring/firmware/libraries/Button; the current Arduino version is at http://github.com/tigoe/Button. Download it to your **libraries** directory as well. Then, restart Arduino and you're ready to begin.

X

Try It There are no global variables, but before the setup() method, you have to include the library and define a couple of constants.

```
// include the necessary libraries:
#include <Wire.h>
#include <LSM303DLH.h>
#include <Button.h>

const int modeButton = 2;          // pushbutton for calibration mode
const int buttonLed = 3;           // LED for the button

// initialize the compass library
LSM303DLH compass;
// initialize a button on pin 2 :
Button button = Button(modeButton,BUTTON_PULLDOWN);
boolean calibrating = false;      // keep track of calibration state
```

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

```
void setup() {
  // initialize serial:
  Serial.begin(9600);
    // set up the button LED:
  pinMode(buttonLed,OUTPUT);
  // start the Wire library and enable the compass:
  Wire.begin();
  compass.enable();
}
```

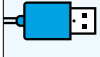
The main loop starts by checking the button. If the button is currently pressed and its state has changed since the last check, the sketch toggles between normal mode and calibrating mode. It also changes the LED state: on means you're calibrating, off means normal mode.

```
void loop() {
  // if the button changes state, change the calibration state
  // and the state of the LED:
  if(button.isPressed() && button.stateChanged()){
    calibrating = !calibrating;
    digitalWrite(buttonLed, calibrating);
  }
```

If the sketch is in calibrating mode, it calls the compass calibrate() method continuously. If not, it reads the compass and reports the heading. Then it waits 100 milliseconds to let the compass stabilize before reading again.

When you run this sketch, open the Serial Monitor, and then put the compass in calibrating mode by pressing the button. Turn it around 360 degrees on a level surface for a second or two, then rotate it through all three axes for a few seconds. Next, press the button to put it in normal mode, and you'll see the heading values. Zero degrees should be due north, 180 degrees is due south, 90 degrees is east, and 270 degrees is west.

```
  // if you're in calibration mode, calibrate:
  if (calibrating) {
    compass.calibrate();
  }
  else {    // if in normal mode, read the heading:
    compass.read();
    int heading = compass.heading();
    Serial.println("Heading: " + String(heading) + " degrees");
  }
  delay(100);
}
```



Introducing the I2C Interface

The LSM303DLH compass uses a form of synchronous serial communication called Inter-Integrated Circuit, or I2C. Sometimes called Two-Wire Interface, or TWI, it's the other common synchronous serial protocol besides SPI, which you learned about in Chapter 4.

I2C is comparable to SPI, in that it uses a single clock on the master device to coordinate the devices that are communicating. Every I2C device uses two wires to send and receive data: a serial clock pin, called the SCL pin, that the microcontroller pulses at a regular interval; and a serial data pin, called the SDA pin, over which data is transmitted. For each serial clock pulse, a bit of data is sent or received. When the clock changes from low to high (known as the rising edge of the clock), a bit of data is transferred from the microcontroller to the I2C device. When the clock changes from high to low (known as the falling edge of the clock), a bit of data is transferred from the I2C device to the microcontroller.

Unlike SPI, I2C devices don't need a chip select pin. Each has a unique address, and the master device starts each exchange by sending the address of the device with which it wants to communicate.

I2C connections have just two connections between the controlling device (or master device) and the peripheral device (or slave), as follows:

Clock (SCK): The pin that the master pulses regularly.
Data (SDA): The pin that data is sent on, in both directions.

All devices on an I2C bus can share the same two lines.

The Arduino Wire library is the interface for I2C. On most Arduino boards, the SDA pin is analog input pin 4, and the SCL pin is on analog input pin 5. On the Arduino Mega, SDA is digital pin 20 and SCL is 21.

Project 21

Determining Attitude Using an Accelerometer

Compass heading is an excellent way to determine orientation if you're level with the earth. And if you've ever used an analog compass, you know how important it is to keep the compass level in order to get an accurate reading. In navigational terms, your tilt relative to the earth is called your attitude, and there are two major aspects to it: roll and pitch. Roll refers to how you're tilted side-to-side. Pitch refers to how you're tilted front-to-back.

Pitch and roll are only two of six navigational terms used to refer to movement. Pitch, roll, and yaw refer to angular motion around the X, Y, and Z axes. These are called rotations. Surge, sway, and heave refer to linear motion along those same axes. These are called translations. Figure 8-14 illustrates these six motions.

Measuring roll and pitch is relatively easy to do using an accelerometer. You used one of these already in Chapter 5, in the balance board ping-pong client. Accelerometers measure changing acceleration. At the center of an accelerometer is a tiny mass that's free to swing in one, two, or three dimensions. As the accelerometer tilts relative to the earth, the gravitational force exerted on the mass changes. Because force equals mass times acceleration, and because the mass of the accelerometer is constant, the change is read as a changing acceleration. In this project, you'll use an accelerometer to control the pitch and roll of a disk onscreen in Processing. The numeric values from the sensor are written on the disk as it tilts.

MATERIALS

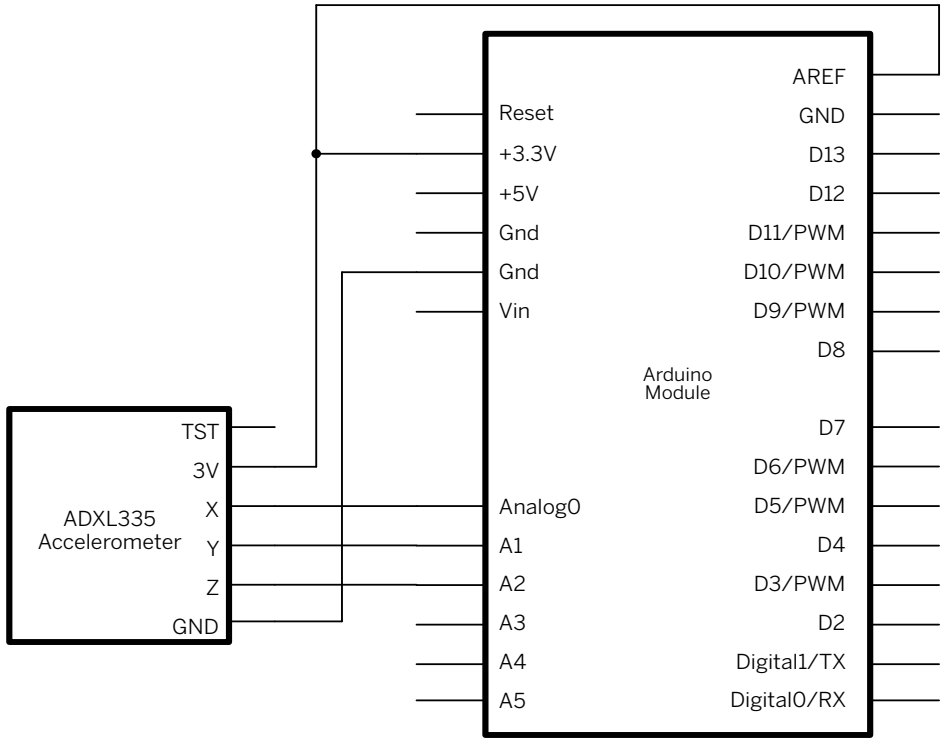
- **1 solderless breadboard or prototyping shield**
- **1 Arduino module**
- **1 Analog Devices ADXL320 accelerometer**
  You can also use the accelerometer on your LSM303DLH digital compass. An alternate sketch to do so is shown below.
- **6 male header pins**

You can use the ADXL335 accelerometer module (the Adafruit version is shown in Figure 8-13) (or another analog accelerometer), or the accelerometer in the LSM303DLH compass module from the previous project.

Accelerometers come with various interfaces. The one on the compass module shares an I2C interface with the magnetometer. Others use a pulse-width interface. Many simply have analog outputs for each axis, like the ADXL accelerometers shown below. Accelerometers also come in a variety of resolutions. For many human activities, around 3–6g, or six times the acceleration due to gravity (9.8 meters per second, per second) will do. Most commercial products using accelerometers, like the Nintendo Wii or the majority of mobile phones, use an accelerometer in this range. The ADXL335 is a 3g accelerometer. Other activities require a much higher range. For example, a boxer's fist can decelerate at up to 100g when he hits!

If you're using the accelerometer on the LSM303DLH compass for this project, use the circuit as shown back in Figure 8-12, but without the LED and pushbutton. The first sketch below will work with any two- or three-axis analog accelerometer, and the second will work with the compass module's accelerometer. Both will communicate with the Processing sketch that follows them.

X



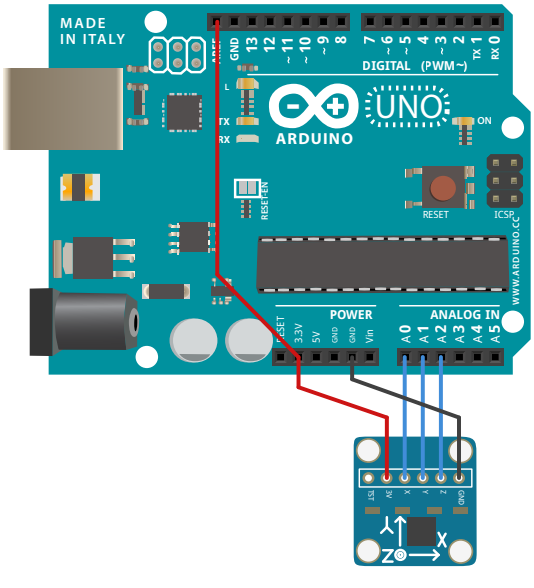


Figure 8-13
ADXL335 accelerometer connected to an Arduino. Shown here is an Adafruit breakout board for the ADXL335. This accelerometer operates on 3.3V, so its output range is also 0 to 3.3V. The microcontroller's analog reference pin is connected to 3.3V as well, so it knows that the maximum range of the analog inputs is from 0 to 3.3V.

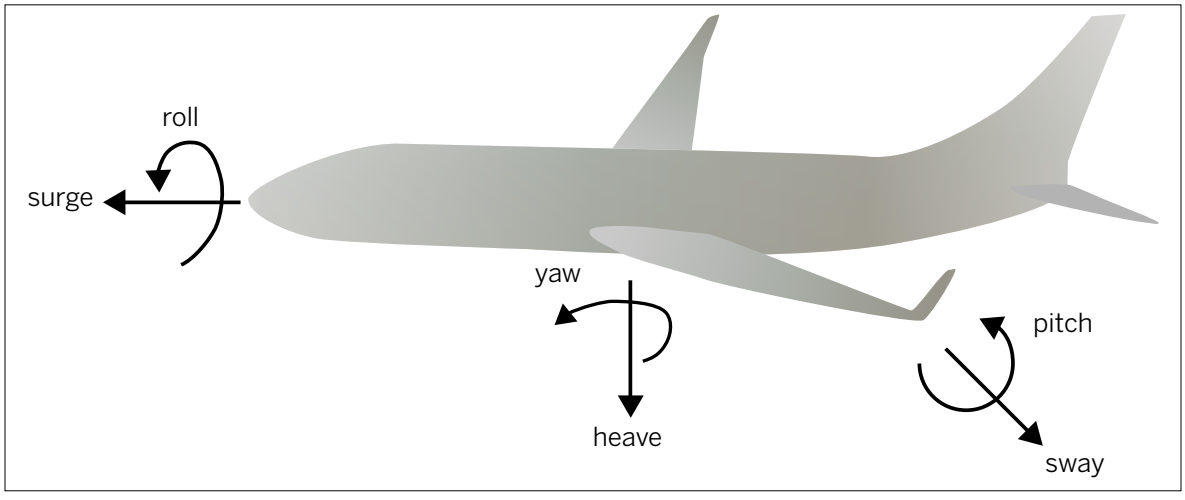


Figure 8-14
Rotations and translations of a body in three dimensions.

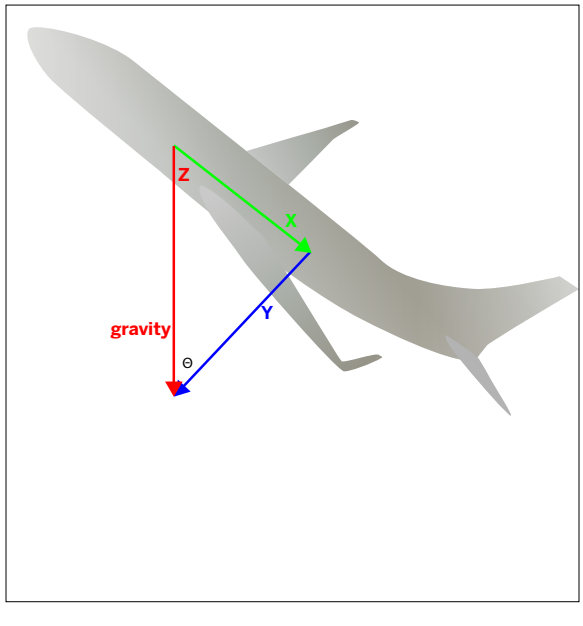
Determining Pitch and Roll from an Accelerometer

Three-axis accelerometers like the one you're using measure the linear acceleration of a body on each axis—in other words, the surge, sway, or heave of a body. They don't give you the roll, pitch, or yaw. However, you can calculate the roll and pitch when you know the acceleration along each axis. That calculation takes some tricky trigonometry. For a full explanation, see Freescale Semiconductor's application note on accelerometers at http://cache.freescale.com/files/sensors/doc/app_note/AN3461.pdf. Here are the highlights:

The force of gravity always acts perpendicular to the earth's surface. So when an object is tilted at an angle (called theta, or θ), part of that force acts along the X axis of the object, and part acts along the Y axis (see Figure 8-15). The X-axis acceleration and the Y-axis acceleration add up to the total force of gravity using the Pythagorean Theorem: $x^2 + y^2 = z^2$.

Since you know that, you can calculate the portions of the acceleration using sines and cosines. The X-axis portion of the acceleration is gravity * sinθ, and the Y-axis portion is gravity * cosθ (remember, sine = opposite/hypotenuse, and cosine = adjacent/hypotenuse).

Figure 8-15
Calculating the portion of the force of gravity based on the angle of tilt.



From there, you have to calculate the portions along all three axes at once. It turns out that:

$$roll = \arctan(x\text{-}axis / \sqrt{(y\text{-}axis^2 + z\text{-}axis^2)})$$

and:

$$pitch = \arctan(y\text{-}axis / \sqrt{(x\text{-}axis^2 + z\text{-}axis^2)})$$

In order to know the acceleration on each axis, though, you have to convert the reading you get from the analogRead() command. It's not too difficult. You already know that the accelerometer's range is from 0 volts to 3.3 volts (because the accelerometer operates on 3.3 volts), and that it gets converted to a range from 0 to 1023. So, you can say that:

```
voltage = analogRead(axis) * 3.3 / 1024;
```

Each axis has a zero acceleration point at half its range, or 1.65 volts. So subtract that from your reading, and any negative reading means tilt in the opposite direction.

From the accelerometer's data sheet, you can find out that the sensitivity of the accelerometer is 300 millivolts per g, where 1g is the amount of the acceleration due to gravity. When any axis is perpendicular to the ground, it experiences 1g of force, and should read about 300 mV, or 0.3V. So, the acceleration on each axis is the voltage reading divided by the sensitivity, or:

```
acceleration = voltage / 0.3;
```

Once you've got the acceleration for each axis, you can apply the earlier trigonometric formulas to get the pitch and roll.

The following sketch reads an analog accelerometer's X and Y axes, calculates pitch and roll as an angle from -90 degrees to 90 degrees, and sends the results out serially. If you want to use this sketch with an analog accelerometer other than the Adafruit ADXL335 module, rearrange the pins to match your accelerometer. If you're using a 5V accelerometer, change the voltage calculation as well.
X

Listen to It (Analog)

The setup() method initializes serial communications, and configures the analog-to-digital converter to take its reference from the external analog reference pin.

NOTE: Use this sketch with the Analog Devices ADXL320 accelerometer.

```
/*
 Accelerometer reader
 Context: Arduino
 Reads 2 axes of an accelerometer, calculates pitch and roll,
 and sends the values out the serial port
 */

void setup() {
  // initialize serial communication:
  Serial.begin(9600);
  // tell the microcontroller to read the external
  // analog reference voltage:
  analogReference(EXTERNAL);
}
```

» The loop() calls a method to convert the analog readings to acceleration values. Then it plugs those results into the trigonometric calculations described above, and prints the results.

```
void loop() {
  // read the accelerometer axes, and convert
  // the results to acceleration values:
  float xAxis = readAcceleration(analogRead(A0));
  delay(10);
  float yAxis= readAcceleration(analogRead(A1));
  delay(10);
  float zAxis = readAcceleration(analogRead(A2));

  // apply trigonometry to get the pitch and roll:
  float pitch = atan(xAxis/sqrt(pow(yAxis,2) + pow(zAxis,2)));
  float roll = atan(yAxis/sqrt(pow(xAxis,2) + pow(zAxis,2)));
  pitch = pitch * (180.0/PI);
  roll = roll * (180.0/PI) ;

  // print the results:
  Serial.print(pitch);
  Serial.print(",");
  Serial.println(roll);
}
```

» The readAcceleration() method takes an analog reading and converts it to an acceleration value from 0 to 1g.

```
float readAcceleration(int thisAxis) {
  // the accelerometer's zero reading is at half
  // its voltage range:
  float zeroPoint = 1.65;
  // convert the reading into a voltage:
  float voltage = (thisAxis * 3.3 / 1024.0) - zeroPoint;
  // divide by the accelerometer's sensitivity:
  float acceleration = voltage / 0.3;
  // return the acceleration in g's:
  return acceleration;
}
```

Regardless of the accelerometer you're using, you'll find that the angle readings aren't always accurate, and that they can be quite noisy. The calculations explained above assume there are no forces acting on the accelerometer besides gravity, but that is seldom the case. As you move the accelerometer through space, your movement accelerates and decelerates, adding more force along all three axes to the calculation. Generally, accelerometer data is combined with data from gyrometers to help adjust for these forces.

If you're using the accelerometer on the LSM303DLH compass from the earlier project, you're in luck. The Arduino library for that accelerometer does the pitch and roll calculations for you, and simply returns the results as pitch and roll. The sketch below reads them and returns the values as angles from -90 degrees to 90 degrees, just like the analog accelerometer sketch above.

X

Listen to It (I2C)

This sketch reads the LSM303DLH accelerometer's X and Y axes and sends the results out serially. Like the previous project, it uses the LSM303DLH library, so make sure you have it installed.

NOTE: This sketch uses the accelerometer on the LSM303DLH digital compass.

```
/*
 I2C accelerometer
 Context: Arduino
 Reads an ST Microelectronics LSM303DLH compass and prints
 the X and Y axes accelerometer output.
  */

// include the necessary libraries:
#include <LSM303DLH.h>
#include <Wire.h>

// initialize the compass:
LSM303DLH compass;

void setup() {
  // initialize serial and Wire, and enable the compass:
  Serial.begin(9600);
  Wire.begin();
  compass.enable();

  // calibrate for the first five seconds after startup:
  while (millis() < 5000) {
    compass.calibrate();
  }
}

void loop() {
  // read the compass and print the accelerometer
  // X and Y readings:
  compass.read();
  Serial.print(compass.pitch());     // X axis angle
  Serial.print(",");
  Serial.println(compass.roll());    // Y axis angle
  delay(100);
}
```

Connect It This Processing sketch reads the incoming data from the microcontroller and uses it to change the attitude of a disc onscreen in three dimensions. It will work with either of the accelerometer sketches above, because they both output the same data in the same format. Make sure the serial port opened by the sketch matches the one to which your microcontroller is connected.

```
/*
 Accelerometer Tilt
 Context: Processing

 Takes the values in serially from an accelerometer
 attached to a microcontroller and uses them to set the
 attitude of a disk on the screen.
 */
import processing.serial.*;      // import the serial lib

float pitch, roll;               // pitch and roll
float position;                  // position to translate to

Serial myPort;                   // the serial port
```

The setup() method initializes the window, the serial connection, and sets the graphics smoothing.

```
void setup() {
  // draw the window:
  size(400, 400, P3D);
  // calculate translate position for disc:
  position = width/2;

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

  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[2], 9600);
  // only generate a serial event when you get a newline:
  myPort.bufferUntil('\n');
  // enable smoothing for 3D:
  hint(ENABLE_OPENGL_4X_SMOOTH);
}
```

You will probably need to look at the output of Serial.list() and change this number to match the serial port that corresponds to your microcontroller.

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

```
void draw () {
  // colors inspired by the Amazon rainforest:
  background(#20542E);
  fill(#79BF3D);
  // draw the disc:
  tilt();
}
```

The 3D system in Processing works on rotations from zero to 2*PI. tilt() maps the accelerometer angles into that range. It uses Processing's translate() and rotate() methods to move and rotate the plane of the disc to correspond with the accelerometer's movement.

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

  // X is front-to-back:
  rotateX(radians(roll + 90));
  // Y is left-to-right:
  rotateY(radians(pitch) );

  // set the disc fill color:
  fill(#79BF3D);
  // draw the disc:
  ellipse(0, 0, width/4, width/4);
  // set the text fill color:
  fill(#20542E);
  // Draw some text so you can tell front from back:
  text(pitch + "," + roll, -40, 10, 1);
}
```

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

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

  // if you got any bytes other than the linefeed:
  if (myString != null) {
    myString = trim(myString);
    // split the string at the commas
    String items[] = split(myString, ',');
    if (items.length > 1) {
      pitch = float(items[0]);
      roll = float(items[1]);
    }
  }
}
```

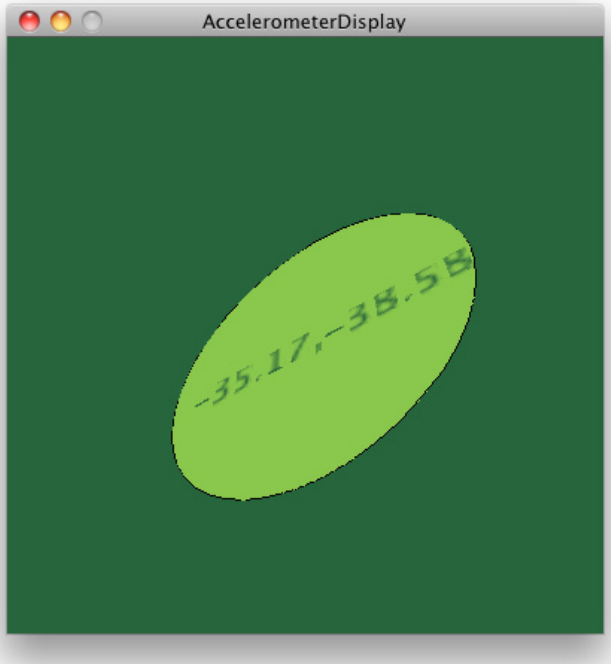


Figure 8-16
The output of the Processing accelerometer sketch.

Though it may seem like a lot of work to go from the raw output of an accelerometer to the visualization shown in Figure 8-16, it's useful to understand the process. You went from the translation of acclerations along three axes into analog voltages, then converted those voltages to digital values in the microcontroller's memory using analogRead(). From there, you converted the digital values into voltage readings, and then converted those to acceleration measurements relative to the acceleration due to gravity. Then, you used some trigonometry to convert the results to angles in degrees.

The advantage of having the results in degrees is that it's a known standard measurement, so you didn't have to do a lot of mapping when you sent the values to Processing. Instead, Processing could take the output from an accelerometer that gave it pitch and roll in degrees.

You don't always need this level of standardization. For many applications, all you care about is that the accelerometer readings are changing. However, if you want to convert those readings into a measurement of attitude relative to the ground, the process you went through is the process you'll use.

X

▸▸ Address 2007 by Mouna Andraos and Sonali Sridhar
Address shows that location technologies don't have to be purely utilitarian. *Photo by J. Nordberg.*

Conclusion

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

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

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

X





9

MAKE: PROJECTS

Identification

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

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

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

Supplies for Chapter 9

Cameras and RFID readers are your main new components in this chapter. You'll be using them to identify colors, faces, tags, and tokens.

Distributor KEY

- **A** Arduino Store (store.arduino.com)
- **AF** Adafruit (http://adafruit.com)
- **CR** CoreRFID (www.rfidshop.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **RSH** Radio Shack(www.radioshack.com)
- **RS** RS (www.rs-online.com)
- **SF** SparkFun (www.sparkfun.com)
- **SH** Smarthome (www.smarthome.com)
- **SS** Seeed Studio (www.seeedstudio.com)
- **ST** SamTec (www.samtec.com)

PROJECT 22: Color Recognition Using a Webcam

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

PROJECT 23: Face Detection Using a Webcam

» **Personal computer with USB or FireWire port**
» **USB or FireWire webcam**
» **Face**

PROJECT 24: 2D Barcode Recognition Using Webcam

» **Personal computer with USB or FireWire port**
» **USB or FireWire webcam**
» **Printer**

PROJECT 25: Reading RFID Tags in Processing

» **RFID reader** The ID Innovations ID-12 or ID-20 can work in this project. The ID-20 has a slightly longer range than the ID-12, but otherwise their operation is identical. **CR** IDI003 or IDI004, **SF** SEN-08419
» **RFID breakout board** Works for either of the ID Innovations readers. **SF** SEN-08423

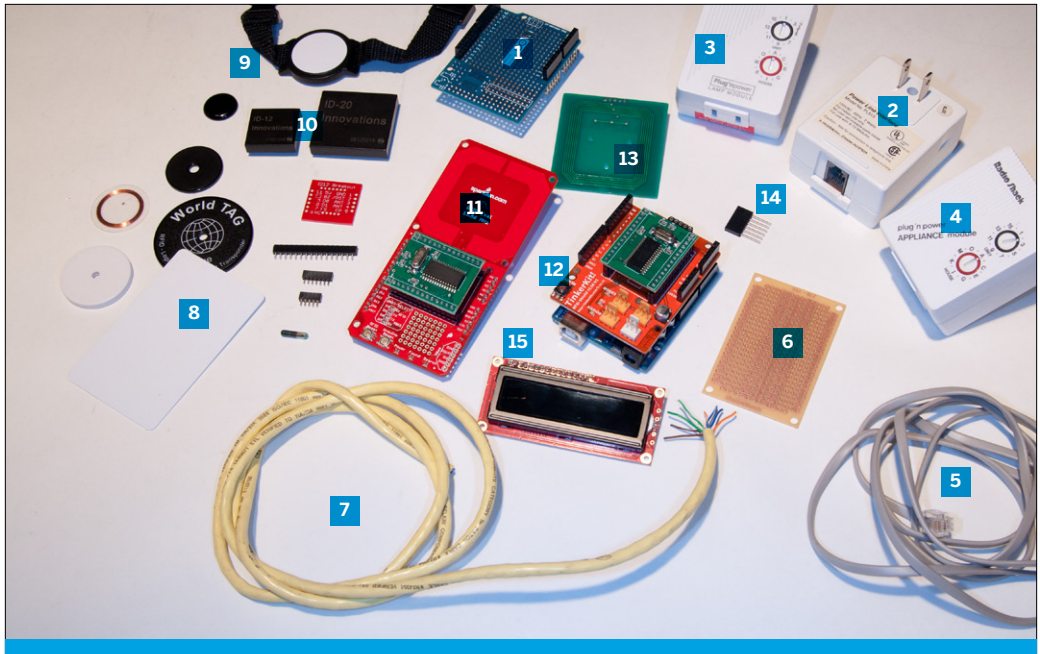


Figure 9-1. New parts for this chapter: **1.** Prototyping shield **2.** X10 interface module **3.** X10 lamp module or **4.** X10 appliance module **5.** Four-wire telephone (RJ-11) cable **6.** Perforated circuit board **7.** Eight-wire Ethernet cable **8.** EM4001 RFID tags **9.** Mifare RFID tags **10.** ID Innovations ID-12 or ID-20 125kHz RFID reader **11.** Spark Fun 13.56 MHz RFID board or **12.** TinkerKit RFID shield **13.** Antenna for SM130 RFID reader **14.** Extra-long female header pins **15.** 2x16 LCD screen. Don't forget plenty of male header pins for the breakout boards.

» **0.1-inch male header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411
» **RFID tags** Get the tags that match your reader. All the retailers listed sell tags that match their readers in a variety of physical packages, so choose the ones you like best. The examples use EM4001 tags, which are interchangeable with EM4102 tags.
CR WON002, **SF** COM-10169
» **1 FTDI USB-to-Serial adapter** You could use the one you've been using throughout this book in conjunction with the RFID breakout board mentioned above. If you want a breakout that's designed specifically for your reader, you can use the RFID USB reader from Spark Fun, part SEN-09963.
SF DEV-09718 or DEV-09716, **AF** 70, **A** A000059, **M** MKAD22, **SS** PRO101D2P, **D** TTL-232R-3V3 or TTL-232R-5V

PROJECT 26: RFID Meets Home Automation

» **1 Arduino module** An Arduino Uno or something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards.
D 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4
» **1 prototyping shield** **J** 2124314, **SF** DEV-07914, **AF** 51, **F** 1848696, **SS** STR104B2P, **M** MSMS01
» **RFID reader** The ID Innovations ID-12 or ID-20 can work in this project. The ID-20 has a slightly longer range than the ID-12, but otherwise their operation is identical.
CR IDI003 or IDI004, **SF** SEN-08419
» **RFID reader breakout** Or use the parts below.
SF SEN-08423
» **Two 2mm 10 pin female header rows** Not necessary if using the breakout board. Samtec, like many part makers, supplies free samples of this part in small quantities.
ST MMS-110-01-L-SV, **J** 164822, **SF** PRT-08272, **F** 1109732
» **2 rows of 20 0.1-inch male header pins** **J** 103377, **D** A26509-20ND, **SF** PRT-00116, **F** 1593411
» **2mm 5-pin socket** **SF** PRT-10519
» **2mm 7-pin socket** **SF** PRT-10518
» **RFID tags** Get the tags that match your reader. All the retailers listed sell tags that match their readers in a variety of physical packages, so choose the ones you like best. The examples use EM4001 tags, which are interchangeable with EM4102 tags.
CR WON002, **SF** COM-10169
» **1 LED** **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590
» **1 220-ohm resistor** **D** 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842
» **Interface module: X10 One-Way Interface Module** **SH** 1134B
» **2 X10 modules** *Either:* 2 appliance modules from Smarthome, part number 2002; or 2 Powerhouse X10 lamp modules from Smarthome, part number 2000. You'll need two modules total. Choose one of each, or two of one as you see fit. If you're going to control incandescent lamps only, get lamp modules. For anything else, get appliance modules.
» **4-wire phone cable with RJ-11 connector** You can take this from any discarded phone, or get one at your local electronics shop.
D A1642R-07-ND, **J** 115617, **F** 1335141

PROJECT 27: Tweets from RFID

» **1 SonMicro SM130 RFID read/write module** **SF** SEN-10126
» **3 Mifare RFID read/write tags** **SF** SEN-10127
» **1 Arduino Ethernet board** **A** A000050
Alternatively, an Uno-compatible board (see Chapter 2) with an Ethernet shield will work.
SF DEV-09026, **J** 2124242, **A** A000056, **AF** 201, **F** 1848680
» **1 RFID shield** **SF** DEV-10406, **A** T040030 or T040031
» **13.56MHz antenna** Unless your reader incorporates an antenna, **A** C000027
» **2 potentiometers** **J** 29082, **SF** COM-09939, **F** 350072, **RS** 522-0625
» **1 perforated printed circuit board** **RSH** 276-150, **D** V2018-ND, **J** 616673, **SS** STR125C2B, **F** 4903213, **RS** 159-5420
» **1/16-inch Mat board**
» **16 pin female header rows** **ST** MMS-110-01-L-SV, **J** 164822, **SF** PRT-08272, **F** 1109732
» **6 pin stackable header** **SF** PRT-09280, **AF** 85
» **8-conductor wire** Ribbon cable or Ethernet cable will do.
D AE08A-5-ND , **F** 1301013
» **16x2 character LCD** **SF** LCD-00709

The parts for building the reader circuit without a shield are listed below, for those who prefer that option:

» **2 4.7-kilohm resistors** **J** 691024, **D** CF14JT4K70CT-ND, **F** 735033, **RS** 707-8693
» **1 solderless breadboard** **D** 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **M** MKKN2

Physical Identification

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

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

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

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

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



Figure 9-2
Video color recognition in Processing, using the code in Project 22. This simple sketch works well with vibrantly pink monkeys.

Video Identification

All video identification relies on the same basic method: the computer reads a camera's image and stores it as a two-dimensional array of pixels. Each pixel has a characteristic brightness and color that can be measured using any one of a number of palettes: red-green-blue is a common scheme for video- and screen-based applications, as is hue-saturation-value. Cyan-magenta-yellow-black is common in print applications. The properties of the pixels, taken as a group, form patterns of color, brightness, and shape. When those patterns resemble other patterns in the computer's memory, it can identify those patterns as objects. Figure 9-2 shows an example, using a bright pink monkey.

In the three projects below, you'll use a computer-vision library called OpenCV to read an image from your personal computer's camera or webcam, and then analyze the image. The first, and simplest, will look for a color. The second will look for something resembling a face. The third will look for a 2D barcode called a QR (Quick Response) code.

OpenCV is a computer-vision library originally developed by Intel and released under an open source license. It's been adapted for many programming environments, including Processing. The Processing version can be found linked off the Processing site at http://processing.org/reference/libraries/. Follow the OpenCV link from that page, then follow the directions to install OpenCV on your platform. Download the Processing OpenCV library and copy it to the **libraries** directory of your Processing sketchbook directory (find it listed in Processing's preferences). Now you're all set to build the examples below.

NOTE: The current home of OpenCV for Processing as of this writing is at http://ubaa.net/shared/processing/opencv. A new version is in the works, so check the Processing site for the most up-to-date link.

Color Recognition

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

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

X

Project 22

Color Recognition Using a Webcam

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

MATERIALS

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

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

» Before the setup(), set up a few global variables, including an instance of the OpenCV library, an array to hold the pixel color values, and a color variable for the color you want to track.

The setup() sets the initial conditions as usual, in this case, initializing OpenCV using the first camera available to your computer, sizing the window, and initializing smooth, anti-alised graphics.

NOTE: Because the OpenCV application uses the first camera available, you may have problems if you have both a built-in camera on a laptop and an external webcam. One solution is to open the camera you don't want to use in another application, so only the one you want is available to OpenCV. It's a crude solution, but it works.

```
/*
 ColorTracking with openCV
 Context: Processing
 Based on an example by Daniel Shiffman
 */

// import the opencv library:
import hypermedia.video.*;

OpenCV opencv;              // opencv instance
int[] pixelArray;           // array to copy the pixel array to
color trackColor;           // the color you're looking for


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

  // initialize opencv
  opencv = new OpenCV( this );
  opencv.capture( width, height );
  // Start off tracking for red
  trackColor = color(255, 0, 0);
  // draw smooth edges:
  smooth();
}
```

The draw() method begins by establishing a value for the closest matching color, a threshold for similarity, and X and Y positions of the closest matching pixel. Then it reads the camera and saves the resulting image in a pixel array.

```
void draw() {
  float closestMatch = 500;   // value representing the closest color match
  float colorThreshold = 10;  // the threshold of color similarity
  int closestX = 0;           // horizontal position of the closest color
  int closestY = 0;           // vertical position of the closest color

  // read the camera:
  opencv.read();
  // draw the camera image to the window:
  image(opencv.image(), 0, 0);
  // copy the camera pixel array:
  pixelArray = opencv.pixels();
```

Next comes a pair of nested for loops that iterates over the rows and columns of pixels. This is a standard algorithm for examining all the pixels of an image. With each pixel, you determine its position in the array with the following formula (which you'll see frequently.)

```
arrayLocation = x + (y * width);
```

Once you have the pixel's position, you extract the red, green, and blue values, as well as the values for the color you wish to track.

```
 // Begin loop to walk through every pixel
  for (int x = 0; x < opencv.width; x++ ) {
    for (int y = 0; y < opencv.height; y++ ) {
      // calculate the pixel's position in the array
      // based on its width and height:
      int loc = x + y*opencv.width;
      // get the color of the current pixel:
      color currentColor = pixelArray[loc];
      float r1 = red(currentColor);
      float g1 = green(currentColor);
      float b1 = blue(currentColor);
      float r2 = red(trackColor);
      float g2 = green(trackColor);
      float b2 = blue(trackColor);
```

Next, calculate the difference between the two colors. By treating the red, green, and blue values of each color as positions in 3-dimensional space, you can find the difference by calculating the Euclidean distance between them. Processing's dist() function is handy for this.

Once you know the difference, compare that to the closest match so far (which was set arbitrarily high for the first match). If the current difference is less than the closest match, the current pixel is the new closest match.

```
      // use the dist() function to figure the aggregate color
      // of the current pixel. This method treats the red, green, and blue
      // of the current pixel's color  and of the target color as
      // coordinates
      // in 3D space and calculates the difference between them
      // as Euclidean distance.
      // In this formula, closer distance = closer color similarity:
      float d = dist(r1, g1, b1, r2, g2, b2);

      // If current color is more similar to tracked color than
      // closest color, save current location and current difference
      if (d < closestMatch) {
        closestMatch = d;
        closestX = x;
        closestY = y;
      }
    }
  }
```

» Once you have the closest match, the nested for loops are over. All that remains in the draw() method is to see whether the closest match is less than the threshold you set as an acceptable match. If it is, draw a circle there.

Continued from previous page.

```
  // Only consider the color found if its color distance is less than
  // the color threshold. For greater color accuracy, make this lower.
  // For more forgiving matching, make it higher:
  if (closestMatch < colorThreshold) {
    // Draw a circle at the tracked pixel
    fill(trackColor);
    strokeWeight(2.0);
    stroke(0);
    ellipse(closestX, closestY, 16, 16);
  }
}
```

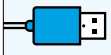
» Now, add a handler for when the mouse is pressed. This changes the tracked color to whatever color is at the mouse location.

```
void mousePressed() {
  // Save color where the mouse is clicked in trackColor variable
  int loc = mouseX + mouseY*opencv.width;
  trackColor = pixelArray[loc];
}
```

Lighting for Color Tracking

As you can see when you run this sketch, it's not the most robust color tracker! The closest match tends to jump around a lot. Changing the colorThreshold helps, but not a lot. You can get it to be more precise by controlling the image and the lighting very carefully. There are some lighting tricks you can use as well:

- DayGlo colors under ultraviolet fluorescent lighting tend to be the easiest to track, but they lock you into a very specific visual aesthetic.
- Objects that produce their own light are easier to track, especially if you put a filter on the camera to block out stray light. A black piece of 35mm film negative (if you can still find 35mm film!) works well as a visible light filter, blocking most everything but infrared light. Two polarizers, placed so that their polarizing axes are perpendicular, are also effective. Infrared LEDs track very well through this kind of filter, as do incandescent flashlight lamps.
- Regular LEDs don't work well as color-tracking objects unless they're relatively dim. Brighter LEDs tend to show up as white in a camera image because their brightness overwhelms the camera's sensor.
- Color recognition doesn't have to be done with a camera; color sensors can do the same job. Texas Advanced Optoelectronic Solutions (www.taosinc.com) makes a few different color sensors, including the TAOS TCS230. This sensor contains four photodiodes, three of which are covered with color filters; the fourth is not, so it can read red, green, blue, and white light. It outputs the intensity of all four channels as a changing pulse width. The cheaper TAOS TSL230R has no LEDs—it just detects ambient color. Other color sensors are available as well. Their shortcoming is that they are designed to detect color only relatively close (within a few centimeters), and they don't have the ability to see a coherent image. They are basically one-pixel camera sensors.



Challenges of Identifying Physical Tokens

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

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

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

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

Shape and Pattern Recognition

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

A computer can't actually "see" in three dimensions using a single camera. The view it has of any object is just a two-dimensional shadow. Furthermore, it has no way of distinguishing one object from another in the camera view without some visual reference. The computer has no concept of a physical object. It can only compare and match patterns. It can rotate the view, stretch it, and do all kinds of mathematical transformations on the pixel array, but the computer doesn't understand an object as a discrete entity the same way a human does.

Face Detection

If you've used a digital camera developed in the past five years or so, chances are it's got a face-detection algorithm built in. It'll put a rectangle around each human face and attempt to focus on it. Face detection is a good example of visual pattern recognition. The camera looks for a predescribed pattern that describes, generically, a face. It has a particular proportion of height to width: there are two darker spots about one-third of the way from the top, a second darker spot about two-thirds of the way down, and so forth. Facial detection is not facial recognition—a face-detection algorithm generally isn't looking specificially enough at an image to tell you who the person is, just that they have something that more or less resembles a face.

OpenCV has patterns for facial detection that are very simple to use. The following project shows you how to detect faces.

X

Project 23

Face Detection Using a Webcam

Now that you've got a basic understanding of how optical detection works from the color-tracking project, it's time to try some simple pattern detection. In this project, you'll use OpenCV's facial-detection methods to look for faces in a camera image.

MATERIALS

- **Personal computer with USB or FireWire port**
- **USB or FireWire webcam**
- **Faces**

OpenCV's pattern detection uses pattern description files called cascades to describe the characteristics of a particular patern. A cascade describes the subregions of a given pattern, including their relative sizes, shapes, and contrast ratios. The patterns are designed to be general enough to allow for some variation, but specific enough to tell it from other patterns. The Processing OpenCV library comes with patterns for the following human features:

Frontal face view (four variations)
Profile face view
Full-body view
Lower-body view
Upper-body view

The sketch attempts basic facial recognition; see if you can fool it.

Before the setup(), set up the global variables as usual. These are almost identical to the last project, but you also need the Java Rectangle object. This is because the OpenCV detection method returns an array of Rectangles that it thinks contain faces.

The setup() is similar, but this time, you're going to have OpenCV look for detection pattern using the library's cascade() method. The available patterns are described on the Processing OpenCV site at http://ubaa.net/shared/processing/opencv/opencv_cascade.html.

Finally, the setup() sets the drawing conditions for ellipses, so you can draw circles over the faces.

```
/*
  Face detection using openCV
  Context: Processing
*/
// import the opencv and Rectangle libraries:
import hypermedia.video.*;
import java.awt.Rectangle;

OpenCV opencv;       // new instance of the openCV library


void setup() {
  // initialize the window:
  size( 320,240 );
  // initialize  opencv:
  opencv = new OpenCV( this );
  opencv.capture( width, height );
  // choose a detection description to use:
  opencv.cascade( OpenCV.CASCADE_FRONTALFACE_DEFAULT );
 // draw smooth edges:
  smooth();
  // set ellipses to draw from the upper left corner:
  ellipseMode(CORNER);
}
```

The draw() method is much simpler than the last project because you're not looking at every pixel. It uses the OpenCV detect() method to look for rectangles matching the pattern you chose, and delivers them in an array. Then it iterates over the array and draws ellipses over each face.

When you run this, point your face at it, and you'll get a nice fuchsia mask, as shown in Figure 9-3.

Try the program on different versions of faces and different conditions. Also try the other face-detection cascades mentioned on the OpenCV for Processing site.

```
void draw() {
  // grab a new frame:
  opencv.read();

  // Look for faces:
  Rectangle[] faces = opencv.detect();

  // display the image:
  image( opencv.image(), 0, 0 );

  // draw circles around the faces:
  fill(0xFF, 0x00, 0x84, 0x3F);   // a nice shade of fuchsia
  noStroke();                     // no border

  for ( int thisFace=0; thisFace<faces.length; thisFace++ ) {
    ellipse( faces[thisFace].x, faces[thisFace].y,
             faces[thisFace].width, faces[thisFace].height );
  }
}
```









Figure 9-3
The face detection finds me fairly well.

Turning sideways, I disappear.

It does well with a photo of people, even the abstracted face on my shirt. Facial hair and other markings make a person harder to detect.

Animals are not detected by this algorithm. Noodles is not detected (and not happy either).

Barcode Recognition

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

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



Figure 9-4
A one-dimensional barcode. This is the ISBN bar code for this book.

Barcodes, such as the one shown in Figure 9-4, are called one-dimensional barcodes because the scanner or camera needs to read the image only along one axis. There are also two-dimensional barcodes that encode data in a two-dimensional matrix for more information density. As with one-dimensional barcodes, there are a variety of symbologies. Figure 9-5 shows a typical two-dimensional barcode. This type of code, the QR (Quick Response) code, was created in Japan and originally used for tracking vehicle parts, but it's since become popular for all kinds of product labeling. The inclusion of software to read these tags on many camera phones in Japan has made the tags more popular. The following project uses an open source Java library to read QR codes in Processing.



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

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

Project 24

2D Barcode Recognition Using a Webcam

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

MATERIALS

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

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

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

In the setup() for this sketch, you'll import the pqrcode and OpenCV libraries, and initialize a few global variables.

```
/*
    QRCode reader
    Context: Processing
*/

import hypermedia.video.*;
import pqrcode.*;

OpenCV opencv;      // instance of the opencv library
Decoder decoder;    // instance of the pqrcode library

// a string to return messages:
String statusMsg = "Waiting for an image";

void setup() {
  // initialize the window:
  size(400, 320);

  // initialize opencv:
  opencv = new OpenCV( this );
  opencv.capture( width, height );

  // initialize the decoder:
  decoder = new Decoder(this);
}
```

The draw() method draws the camera image and prints a status message to the screen. If the decoder library is in the middle of reading an image, it displays that image in the upper-lefthand corner, and changes the status message.

```
void draw() {
  // read the camera:
  opencv.read();
  // show the camera image:
  image( opencv.image(), 0, 0 );

  // Display status message:
  text(statusMsg, 10, height-4);

  // If you're currently decoding:
  if (decoder.decoding()) {
    // Display the image being decoded:
    PImage show = decoder.getImage();
    image(show, 0, 0, show.width/4, show.height/4);
    // update the status message:
    statusMsg = "Decoding image";
    // add a dot after every tenth frame:
    for (int dotCount = 0; dotCount < (frameCount) % 10; dotCount++) {
      statusMsg += ".";
    }
  }
}
```

The pqrcode library has a method called decodeImage(). To use it, pass it an image in the keyReleased() method. A switch statement checks to see which key has been pressed. If you type f, it passes the decoder a file called **qrcode.png** from the **data** subdirectory. If you press the space bar, it passes the camera image. If you type s, it brings up a camera settings dialog box.

```
void keyReleased() {
  String code = "";
  // Depending on which key is hit, do different things:
  switch (key) {
  case ' ':        // space bar takes a picture and tests it:
    // Decode the image:
    decoder.decodeImage(opencv.image());
    break;
  case 'f':    // f runs a test on a file
    PImage preservedFrame = loadImage("qrcode.png");
    // Decode the file
    decoder.decodeImage(preservedFrame);
    break;
  }
}
```

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

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

When you run this, notice how the **.jpg** or **.png** images scan much more reliably than the camera images. The distortion from the analog-to-digital conversion through the camera causes many errors. This error is made worse by poor optics or low-end camera imaging chips in mobile phones and webcams. Even with a good lens, if the code to be scanned isn't centered, the distortion at the edge of an image can throw off the pattern-recognition routine. You can improve the reliability of the scan by guiding the user to center the tag before taking an image. Even simple graphic hints like putting crop marks around the tag, as shown in Figure 9-5, can help. When you do this, users framing the image tend to frame to the crop marks, which ensures more space around the code and a better scan. Such methods help with any optical pattern recognition through a camera, whether it's one- or two-dimensional barcodes, or another type of pattern altogether.



Figure 9-5
A two-dimensional barcode (a QR code, to be specific) with crop marks around it. The image parsers won't read the crop marks, but they help users center the tag for image capture.

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

Radio Frequency Identification (RFID)

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

There are two types of RFID system: passive and active, just like distance-ranging systems. Passive RFID tags contain an integrated circuit that has a basic radio transceiver and a small amount of nonvolatile memory. They are powered by the current that the reader's signal induces in their antennas. The received energy is just enough to power the tag to transmit its data once, and the signal is relatively weak. Most passive readers can only read tags a few inches to a few feet away.

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

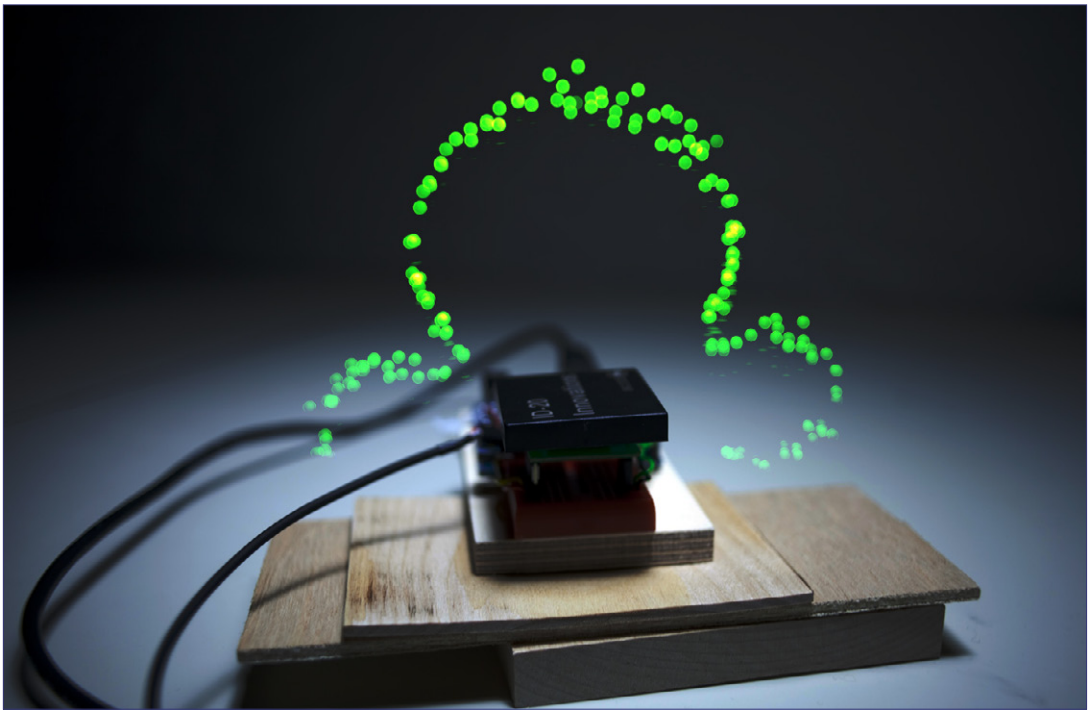


Figure 9-6
The field of an RFID reader, by Timo Arnall. This stop-motion photo shows the effective range and shape of the RFID reader's field. The reader shown is an ID Innovations ID-20, which you'll see in the next project.

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

RFID systems vary widely in cost. Active systems can cost tens of thousands of dollars to purchase and install. Commercial passive systems can also be expensive. A typical passive reader that can read a tag a meter away from the antenna typically costs a few thousand dollars. At the low end, short-range passive readers can come as cheap as $30 or less. As of this writing, $30 to $100 gets you a reader that can read a few inches. Anything that can read a longer distance will be more expensive.

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

There are a number of cheap and simple readers on the market now, covering the range of passive RFID frequencies and protocols. ID Innovations makes a range of small, inexpensive, and easy-to-use 125kHz readers with a serial output. The smallest of these is less than 1.5 inches on a side and is capable of reading the EM4001 protocol tags. Spark Fun and CoreRFID both sell these readers and matching tags. You'll use one of these in the next project.



Figure 9-7
RFID tags in all shapes and sizes. All of thes items have RFID tags in them. Photo by Timo Arnall. For more information on RFID design research by Arnall and his colleagues, see www.nearfield.org.

Parallax sells a 125kHz reader that can also read EM Microelectronic tags, such as EM4001. It has a built-in antenna, and the whole module is about 2.5" x 3.5" on a flat circuit board. The ID Innovations readers and the Parallax readers can read the same tags. The EM4001 protocol isn't as common in everyday applications as the Mifare protocol, a variation on the ISO 14443 standard in the 13.56MHz range. Mifare shows up in many transit systems, like the London Tube. SonMicro makes a module that can both read from and write to these tags, which you'll see in the project after next.

As shown in Figure 9-7, RFID tags come in a number of different forms: sticker tags, coin discs, key fobs, credit cards, playing cards, even capsules designed for injection under the skin. The last are used for pet tracking and are not designed for human use, though there are some adventurous hackers who have inserted these tags under their own skin. Like any radio signal, RFID can be read through a number of materials, but it is blocked by any kind of RF shielding, such as wire mesh, conductive fabric lamé, metal foil, or adamantium skeletons. This feature means that you can embed it in all kinds of projects, as long as your reader has the signal strength to penetrate the materials.

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

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

The readers mentioned here have TTL serial interfaces, so they can be connected to a microcontroller or a USB-to-Serial module very easily. The ID Innovations and Parallax readers have a similar serial behavior, so you could swap one for the other with only a few code changes to your program.
X

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

Project 25

Reading RFID Tags in Processing

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

MATERIALS

- **ID Innovations ID-12 or ID-20 RFID reader**
- **EM4001 RFID tags**
- **RFID breakout board**
- **Male header pins**
- **1 USB-to-TTL serial adapter**

The ID Innovations readers operate on 5 volts and have a TTL serial output, so the circuit is very simple.

There's also a buzzer pin, which goes high whenever a tag is read, so you know that the reader's working. An LED will work fine in place of the buzzer. Figure 9-8 shows the circuit for the ID Innovations reader connected to a USB-to-Serial adapter. The Parallax reader can work for this project too, though you'll have to modify the code to match its protocol.

All the ID Innovations readers use the same protocol. They operate at 9600bps. The serial sentence begins with a start-of-transmission (STX) byte (ASCII 02) and ends with an end-of-transmission (ETX) byte (ASCII 03). The STX is followed by the 10-byte tag ID. A checksum follows that, then a carriage return (ASCII 13) and linefeed (ASCII 10), then the ETX. The EM4001 tags format their tag IDs as ASCII-encoded hexadecimal values, so the string will never contain anything but the ASCII digits 0 through 9 and the letters A through F.

Try It The Processing sketch shown here reads from an ID Innovations reader. The setup() should look very familiar to you by now—it's just opening the serial port and establishing a string for incoming data. Since the ID Innovations serial sentence ends with the value 03, you'll buffer any incoming serial until you see a byte of value 03.

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

  Reads data serially from an ID Innovations ID-12 RFID reader.
*/
// import the serial library:
import processing.serial.*;

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

void setup() {
  size(150,150);
  // list all the serial ports:
  println(Serial.list());

  // change the number below to match your port:
  String portnum = Serial.list()[2];
  // initialize the serial port:
  myPort = new Serial(this, portnum, 9600);
  // incoming string will end with 0x03:
  myPort.bufferUntil(0x03);
}
```

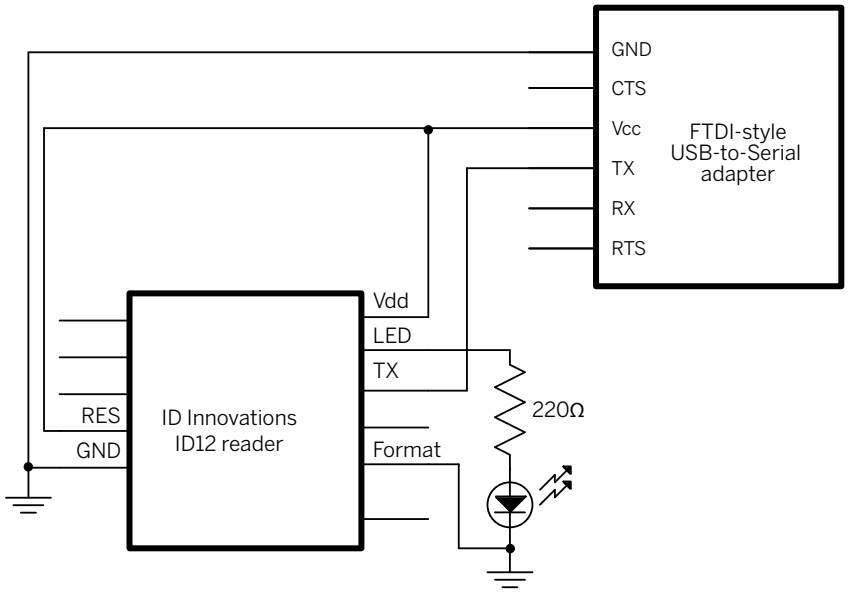
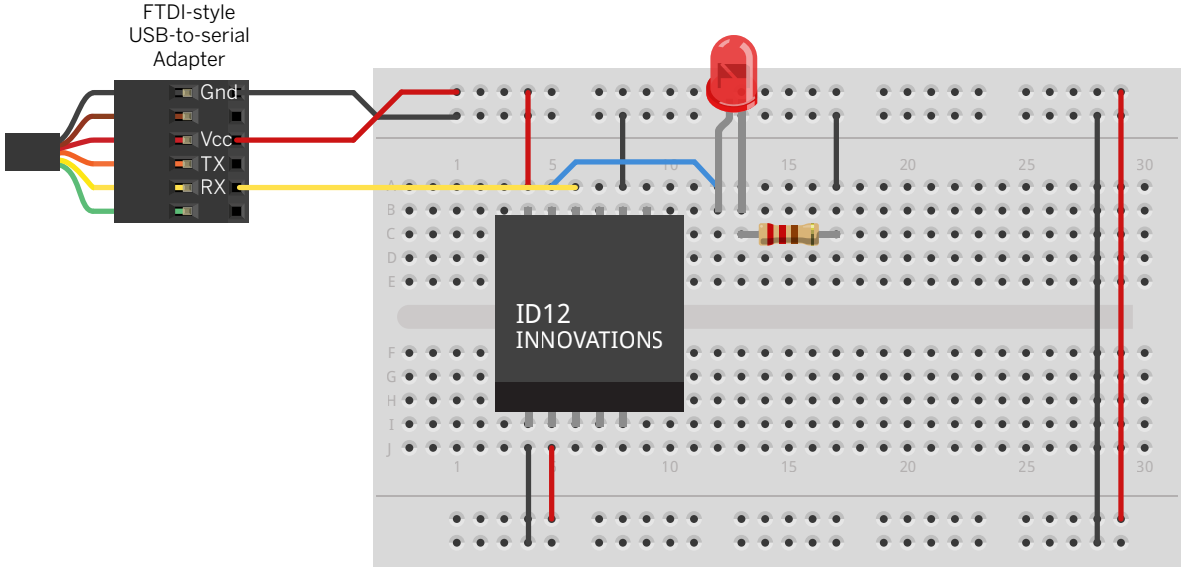


Figure 9-8
The ID Innovations ID-12 RFID reader attached to an FTDI USB-to-Serial adapter. The ID-12 has pins spaced 2mm apart. Spark Fun's breakout board breaks it out to standard breadboard spacing, however.

If you plan to use the ID Innovations readers a lot, you might consider replacing this circuit with Spark Fun's RFID USB reader (part no. SEN-09963), which combines the FTDI adapter and mount for the reader in one module.



» The draw() method draws the tag ID to the screen.

```
void draw() {
  // clear the screen and choose
  // pleasant colors inspired by a seascape:
  background(#022E61);
  fill(#D9EADD);
  // print the string to the screen:
  text(tagID, width/4, height/2);
}
```

» The real work happens in the serialEvent() method. It's called whenever a byte of value 03 comes in the serial port. It checks to see whether the string starts with a byte of value 02 and ends with 03. If so, it takes the first 10 bytes after the 02 as the tag ID.

Run this sketch with the ID Innovations reader attached, and wave some tags in front of it. With each tag, you should get a result like the screenshot in Figure 9-9.

```
void serialEvent(Serial myPort) {
  // get the serial input buffer in a string:
  String inputString = myPort.readString();

  // filter out the tag ID from the string:
  // first character of the input:
  char firstChar = inputString.charAt(0);
  // last character of the input:
  char lastChar = inputString.charAt(inputString.length() -1);

  // if the first char is STX (0x02) and the last char
  // is ETX (0x03), then put the next 10 bytes
  // into the tagID string:

  if ((firstChar == 0x02) && (lastChar == 0x03)) {
    tagID = inputString.substring(1, 11);
  }
}
```



Figure 9-9
Output of the ID Innovations Reader sketch.

When you have this sketch working, use it to test the range of your reader to see from how far away you can read a tag. You'll notice that the tag has to leave the reader's range before it can be read a second time. You may not be able to tell that from the screen output, but you'll notice that the LED goes off when the tag goes out of range, and it turns on again when it comes back in range. Many readers have a similar behavior. You should also see that the reader can't read more than one tag at a time—this is common among all the readers mentioned here.

X

Project 26

RFID Meets Home Automation

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

When we come into the office, all we have to do is touch our keys on a plate by the door, and the room turns on or off the devices we normally use. Each of us has a key ring with an RFID-tag key fob. The module behind the plate has an RFID reader in it, which reads the tags.

The reader is connected to a microcontroller module that communicates over the AC power lines using the X10 protocol. Each of the various power strips is plugged into an X10 appliance module. Depending on which tag is read, the microcontroller knows which modules to turn on or off. Figure 9-10 shows the system.

MATERIALS

- **1 solderless breadboard**
- **1 Arduino module**
- **1 ID Innovations ID-12 or ID-20 RFID reader**
- **2 EM4001 RFID tags**
- **1 RFID breakout board**
- **Male header pins**
- **Interface module: X10 One-Way Interface Module**
- **2 X10 modules**
- **4-wire phone cable with RJ-11 connector**

The Circuit

The X10 interface module connects to the microcontroller via a four-wire phone cable. Clip one end of the cable and solder headers onto the four wires. Then connect them to the microcontroller as shown in Figure 9-11. The schematic shows the phone jack (an RJ-11 jack) on the interface module as you're looking at it from the bottom. Make sure the wires at the header ends correspond with the pins on the jack from right to left.

NOTE: The colors for the four-wire cable shown in Figure 9-11 are typical for many four-wire phone cables in the U.S., but they do not correspond to the colors used in other diagrams in this book, so be careful to connect the circuit exactly as shown.

The RFID reader's reset pin is connected to the microcontroller's reset pin so that whenever the latter is reset, the former will be as well.

The RFID reader is connected to the microcontroller a little differently than you might expect. Its serial transmit pin is connected to pin 7. You're going to use the SoftwareSerial library to add an extra serial connection to the Arduino this time. The SoftwareSerial library is included in the standard download for Arduino, so you won't need to download anything in order to use it.

To send X10 commands, use the X10 library for Arduino. You can download it from https://github.com/tigoe/x10. Unzip it and place the resulting directory in the **libraries** subdirectory of your Arduino sketch directory. Then restart the Arduino environment.

This project doesn't receive any X10 data, it only sends from the microcontroller to the modules. However, the circuit should be compatible with two-way interface modules. Likewise, the library—though not written to send both ways—could be adapted to send as well as receive. You'll need to add your own sending methods to the library, as the current ones are just stubs. The library should be compatible with both 50Hz and 60Hz AC systems as well. For experienced programmers, the stubs of receive methods can be found in the library's source code—feel free to adapt it as you see fit. For readers seeking a more advanced X10 library, see http://load-8-1.blogspot.com/2010_06_01_archive.html. Thanks to reader Tom Crites for the link.

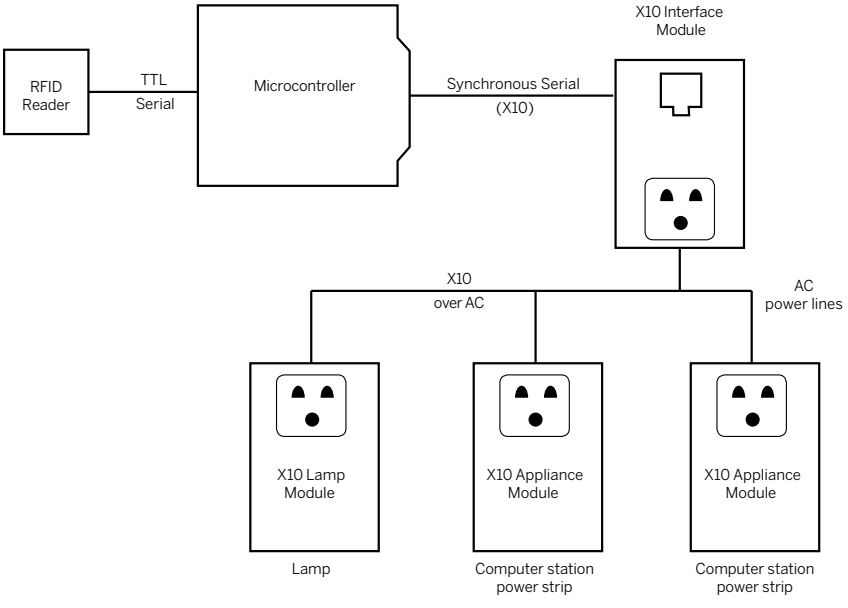
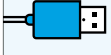


Figure 9-10
An RFID-controlled home (or office) automation system using X10.



What Is X10?

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

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

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

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

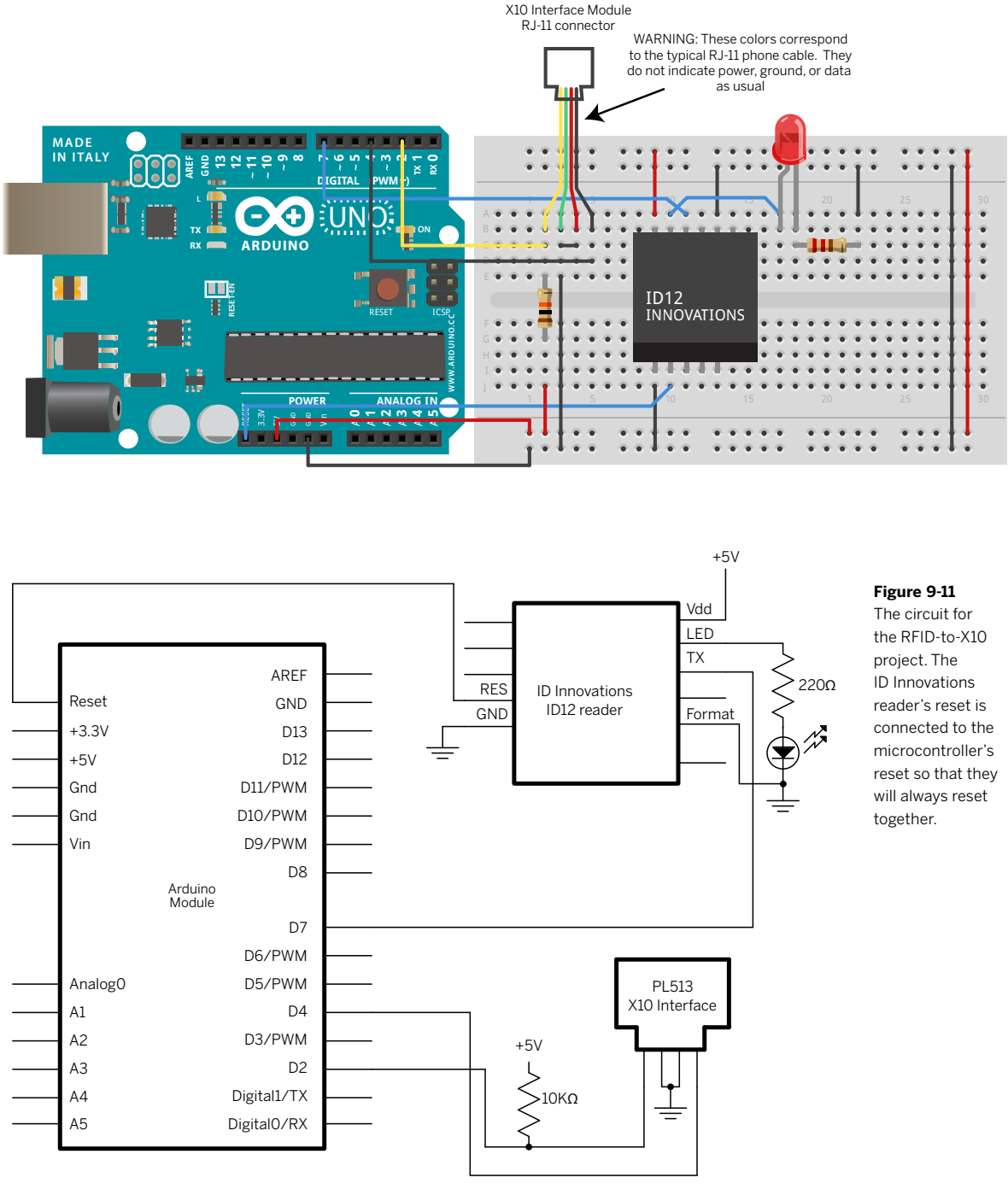


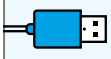
Figure 9-11
The circuit for the RFID-to-X10 project. The ID Innovations reader's reset is connected to the microcontroller's reset so that they will always reset together.

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

Construction

The box for this project is similar to the ones for the Pong clients in Chapter 5. In fact, it's made from the same template, with the side heights and widths changed slightly. There's only a single LED hole, and there's no hole for the Ethernet jack (since there isn't one!). Instead, there's a hole for the X10 cable to go through. Figure 9-12 shows the finished prototype, and a commercially available version of an RFID door lock as well.

X



More Serial Ports: Software Serial

The Arduino's hardware serial connection (digital pins 0 and 1, labeled RX and TX) allow it to send and receive serial data reliably, no matter what your code is doing, because the processor has a dedicated UART (Universal Asynchronous Receiver-Transmitter) module that listens for serial all the time. What do you do when you need to attach more than one asynchronous serial device to your Arduino? The Arduino Mega 2560 has four UARTs, but you may not need all that a Mega has to offer just to get another serial port. This is where the SoftwareSerial library comes in handy. SoftwareSerial allows you to use two digital pins as a "fake" UART. The library can listen for incoming serial on those pins, and transmit as well. Because it's not a dedicated hardware UART, SoftwareSerial isn't as reliable as hardware serial at very high or very low speeds, though it does well from 4800bps through 57.6kbps. It can be very useful when you need an extra port, or when you want to use the hardware serial port for diagnostic purposes, as in this project.

Test the RFID

The first thing you want to do is test both parts of your circuit. This sketch just reads from the RFID reader via a software serial port, and writes what it got to the hardware serial port. Open the Serial Monitor and wave a tag or two in front of the reader. You should see their tag numbers show up in the Serial Monitor. Write down your tags' IDs—you'll need them later.

```
/*
 RFID Reader
 Context: Arduino
*/
#include <SoftwareSerial.h>
SoftwareSerial rfid(7,8);       // 7 is rx, 8 is tx

void setup() {
  // begin serial:
  Serial.begin(9600);
  rfid.begin(9600);
}
void loop() {
  // read in from the reader and
  // write it out to the serial monitor:
  if (rfid.available()) {
    char thisChar = rfid.read();
    Serial.write(thisChar);
  }
}
```

Test the X10

Once you know the RFID reader works, it's time to test the X10 output. This sketch turns on Unit 1 in house A for half a second, then turns it off. To test it, plug in your X10 interface module to the wall, and plug in a lamp module with a lamp connected and turned on to the same circuit in your house. When the sketch runs, the lamp should blink.

```
/*
  X10 blink
  Context: Arduino
 */
#include <x10.h>

const int rxPin = 3;    // data receive pin
const int txPin = 4;    // data transmit pin
const int zcPin = 2;    // zero crossing pin

void setup() {
  // initialize serial and X10:
  Serial.begin(9600);
  x10.begin(rxPin, txPin, zcPin);
}

void loop() {
  // open transmission to house code A:
  x10.beginTransmission(A);
  Serial.println("Lights on:");
  // send a "lights on" command:
  x10.write(ON);

  delay(500);
  Serial.println("Lights off:");
  // send a "lights off" command:
  x10.write(OFF);
  x10.endTransmission();
  delay(500);
}
```

It's unlikely that this will work the first time. X10 is notorious for having synchronization problems while you're developing the hardware and firmware. For one thing, it doesn't work when the transmitter and receiver are on different circuits, so you need to know which circuits in your house's circuit-breaker panel control which outlets. Some surge protectors and power strips might filter out X10 as well, so make sure that the X10 units are plugged into the wall, not the surge protector. You can plug surge protectors into X10 appliance modules, but avoid plugging them into lamp modules unless everything plugged into the surge protector is an incandescent lamp.

X10 lamp modules allow you to dim incandescent lights, but they will control only resistive loads—that means no blow dryers, blenders, or anything with a motor. Compact fluorescent bulbs are generally not dimmable, and they are not designed for use on lamp modules either. If you're unsure, use an appliance module instead of a lamp module.

If your lights don't turn on correctly, first unplug everything, then set the addresses, then plug everything back in, then reset the Arduino. If that fails, make sure your units are on the same circuit, and eliminate surge protectors (if you're using them.) Try to turn the modules using a control panel module. Make sure the control panel isn't sending an ALL UNITS OFF signal at the same time as your Arduino. If needed, unplug the control panel once you know the lamp module is responding. Once you've got control over your modules, you can combine the RFID and X10 programs. **X**

Refine It This sketch reads incoming tags and checks them against a list of tags in memory. When it sees a tag it knows, it checks the status of an X10 lamp or appliance module that corresponds to that tag, and then changes the status.

The global variables are the X10 pin and SoftwareSerial pin numbers, the number of tags being used, and a bunch of arrays for the tag IDs, unit names, and unit states.

The setup() method initializes serial, software serial, and X10, then sends an ALL LIGHTS OFF code to reset all the remote X10 units.

```
/*
  RFID Tag checker
  Context: Arduino
*/
// include the X10  and softwareSerial library files:
#include <x10.h>
#include <SoftwareSerial.h>

const int x10ZeroCrossing = 2;  // x10 zero crossing pin
const int x10Tx = 3;            // x10 transmit pin
const int x10Rx = 4;            // x10 receive pin (not used)
const int rfidRx = 7;           // rfid receive pin
const int rfidTx = 8;           // rfid transmit pin (not used)
int numTags = 2;                // how many tags in your list

String currentTag;              // String to hold the tag you're reading
// lists of tags, unit names, and unit states:
String tag[] = {
  "10000CDFF7","0F00AD72B5"};
int unit[] = {
  UNIT_1, UNIT_2};
int unitState[] = {
  OFF, OFF};

SoftwareSerial rfid(rfidRx,rfidTx);

void setup() {
  // begin serial:
  Serial.begin(9600);
  rfid.begin(9600);
  // begin x10:
  x10.begin(x10Tx, x10Rx,x10ZeroCrossing);
  // Turn off all lights:
  x10.beginTransmission(A);
  x10.write(ALL_LIGHTS_OFF);
  x10.endTransmission();
}
```

» The loop() method just checks for new serial bytes from the RFID reader. The real work is left to a pair of other methods.

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

The readByte() method is called every time a new byte comes in from the RFID reader. If the incoming byte is 02, it starts a new current tag string. If the byte is 03, it checks the current tag against the list of known tags using a method called checkTags(). If it's any other byte, it checks to see whether the current tag string is long enough; if not, it adds the new byte to that string.

```
void readByte() {
  char thisChar = rfid.read();

  // depending on the byte's value,
  // take different actions:
  switch(thisChar) {
    // if the byte = 02, you're at the beginning
    // of a new tag:
  case 0x02:
    currentTag = "";
    break;
    // if the byte = 03, you're at the end of a tag:
  case 0x03:
    checkTags();
    break;
    // other bytes, if the current tag is less than
    // 10 bytes, you're still reading it:
  default:
    if (currentTag.length() < 10) {
      currentTag += thisChar;
    }
  }
}
```

The checkTags() method checks the current tag string against the known list. When it finds the tag, it checks the state of the corresponding X10 unit from a list of unit states. Then it sends that unit a message to change its state from ON to OFF, or vice versa.

```
void checkTags() {
  // iterate over the list of tags:
  for (int thisTag = 0; thisTag < numTags; thisTag++) {
    // if the current tag matches the tag you're on:

    if (currentTag.equals(tag[thisTag])) {
      // unit number starts at 1, but list position starts at 0:
      Serial.print("unit " + String(thisTag +1));
      // start transmission to unit:
      x10.beginTransmission(A);
      x10.write(unit[thisTag]);
      // change the status of the corresponding unit:
      if (unitState[thisTag] == ON) {
        unitState[thisTag] = OFF;
        Serial.println(" turning OFF");
      }
      else {
        unitState[thisTag] = ON;
        Serial.println(" turning ON");
      }
      // send the new status:
      x10.write(unitState[thisTag]);
      // end transmission to unit:
      x10.endTransmission();
    }
  }
}
```

When you run this code, you'll see that the RFID reader only reads when a new tag enters its field. The ID Innovations readers don't have the ability to read multiple tags if more than one tag is in the field. That's an important limitation. It means that you have to design the interaction so that the person using the system places only one tag at a time, and then removes it before placing the second one. In effect, it means that two people can't place their key tags on the reader at the same time. Users of the system need to take explicit action to make something happen. Presence isn't enough.

The simplest solution is to design the project physically so that the user has to remove the tag once he passes it by the reader, as shown in Figure 9-12. This is the most common commercial solution as well. For example, the front door of my apartment building uses an RFID lock. The reader is vertical so I can't leave my keys on it, and the inside locks are regular keys, so I have incentive to remove the keyring once I open the front door if I want to open the inner door.

X

Figure 9-12
The finished RFID reader box, left, and a commmercially available RFID door lock on the front of an apartment building, right. Mounted by the door, it ensures that the user has to remove the tag. This works around the reader's limitation of being able to read only one tag at a time, To avoid a second wire, the prototype on the left is battery-powered.





Project 27

Tweets from RFID

You've seen RFID readers in action, but there are some RFID tags that can be written to as well. In the first part of this project, you'll use the popular Mifare standard RFID read/write tags and a SonMicro high-frequency reader to write Twitter handles to tags. In the second part, you'll build a microcontroller-based reader to read the data from those tags and display it on a 2x16-character LCD display. This links the physical identity of an RFID tag to your network identity on Twitter.

MATERIALS

- **1 SonMicro SM130 RFID read/write module**
- **3 Mifare RFID read/write tags**
- **Arduino Ethernet module**
- **1 RFID shield**
- **13.56MHz antenna** Unless your reader incorporates an antenna

The parts for building the reader circuit without a shield are listed below, for those who prefer that option:

- **2 4.7-kilohm resistors**
- **1 solderless breadboard**

RFID tags are often used for more than just the serial number stored on them. In some mass-transit systems, the customer's available balance is read from, decremented, and written to the card with each transaction. Some conferences use the RFID tag to store business card information, so attendees can exchange data digitally by tapping their cards to a reader. Near field communications (NFC) enhances this kind of exchange even further. NFC involves both passive and active exchange, where two devices communicate with each other over short distances. NFC accommodates a number of communications standards, including some of the RFID standards like ISO14443A and B, which includes Mifare. Because of this, NFC devices are often compatible with Mifare RFID readers and tags. It is starting to gain popularity in mobile phones and other portable devices, so look for many NFC and RFID applications in the near future. Projects like this one here may become relatively commonplace.

The Circuit

The SonMicro SM130 RFID reader operates on 5 volts, and it can communicate with a microcontroller using either asynchronous serial communication or synchronous serial via I2C. You'll see both in practice in this project. For the first part, you'll communicate with Processing directly using asynchronous serial. In the second part, you'll communicate with the microcontroller directly using I2C.

There are two Arduino shields available to connect the SM130 to an Arduino: the TinkerKit RFID shield and the Spark Fun RFID Evaluation shield 13.56MHz. Both allow you to connect to either the asynchronous serial or the I2C connections. The Spark Fun board uses solder jumpers, which let you choose whether to connect to digital pins 7 and 8 for a software serial connection. The TinkerKit shield has a switch that turns the serial connection hardware on or off. The Spark Fun comes with a built-in antenna. The TinkerKit shield has connections for an external antenna so you can place it where you wish, relative to the board.

To communicate with Processing, you could use a USB-to-Serial adapter, like you've done with many other projects, or you could use the Arduino as a USB-to-Serial adapter. For the Spark Fun board, you would need to include a sketch to pass the software serial data to the hardware serial, and vice versa. For the TinkerKit board, you would need to switch the serial switch to ON and put a blank sketch on the board. Both sketches are shown below.

There's only a difference between these two shields when you're communicating with the RFID reader via asynchronous serial, because they use differing pins for that. When you're communicating with the reader using I2C, as you will later in the project, they're functionally identical.

This project was built using the TinkerKit RFID shield. It's been tested on both, however.

If you're not using a shield, connect the SonMicro reader to a USB-to-Serial adapter, as shown in Figure 9-13. Then load the appropriate Arduino sketch below, and you're ready to try the Processing sketch on the following pages. X

Bypass It (TinkerKit)

This sketch passes data from a SonMicro RFID reader on a TinkerKit RFID shield to the Arduino's USB-to-Serial adapter, bypassing the microcontroller. When your board is programmed this way, the Arduino acts as a USB-to-Serial adapter for the SonMicro reader.

```
/*
  TinkerKit RFID shield serial pass through
  Context: Arduino
*/

void setup() {
}

void loop() {
}
```

Bypass It (Spark Fun)

This sketch passes data from a SonMicro RFID reader on a Spark Fun RFID shield to the Arduino's hardware serial port, and vice versa. When your board is programmed this way, the Arduino acts as a USB-to-Serial adapter for the SonMicro reader.

```
/*
  Spark Fun RFID shield serial pass through
  Context: Arduino
*/

#include <SoftwareSerial.h>

// using pins 7 and 8 (7 is the Arduino's RX, 8 is TX)
SoftwareSerial rfid(7,8);


void setup() {
  rfid.begin(19200);     // set up software serial port
  Serial.begin(19200);   // set up serial port
}

void loop() {
  // pass any hardware serial to the software serial:
  if (Serial.available()) {
    rfid.write(Serial.read());
  }
  // pass any software serial to the hardware serial:
  if (rfid.available()) {
    Serial.write(rfid.read());
  }
}
```

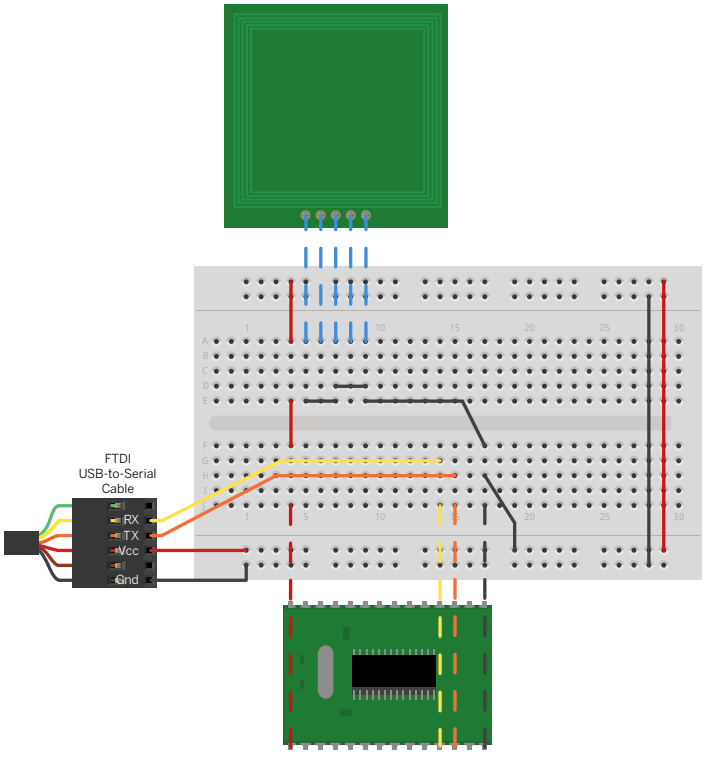
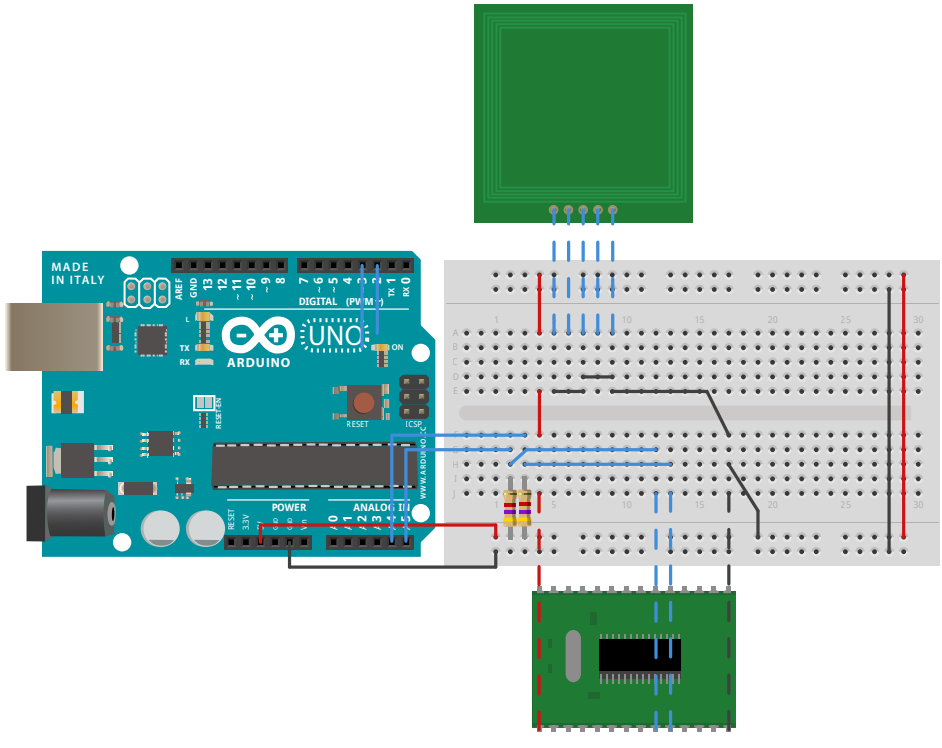


Figure 9-13
SonMicro RFID reader attached to a USB-to-Serial adapter, exploded view. The reader and antenna normally sit on the breadboard in the same position as shown in the I2C breadboard view in Figure 9-14.

Figure 9-14
SonMicro RFID reader attached to an Arduino via I2C, exploded view, showing the connections under the module on the breadboard. The circuit here is basically the same as both the TinkerKit shield and the Spark Fun shield. The I2C connections need the two 4.7-kilohm pullup resistors. You don't need this circuit if you're using an RFID shield.



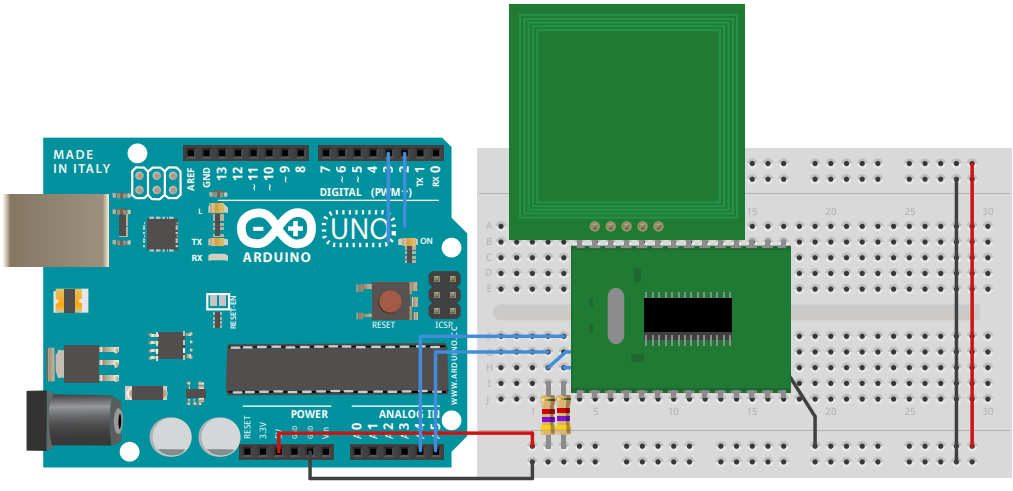


Figure 9-15
SonMicro RFID reader attached to an Arduino via I2C, normal position on the breadboard. The module sits in the same position on the board for the USB-to-Serial adapter circuit as well.

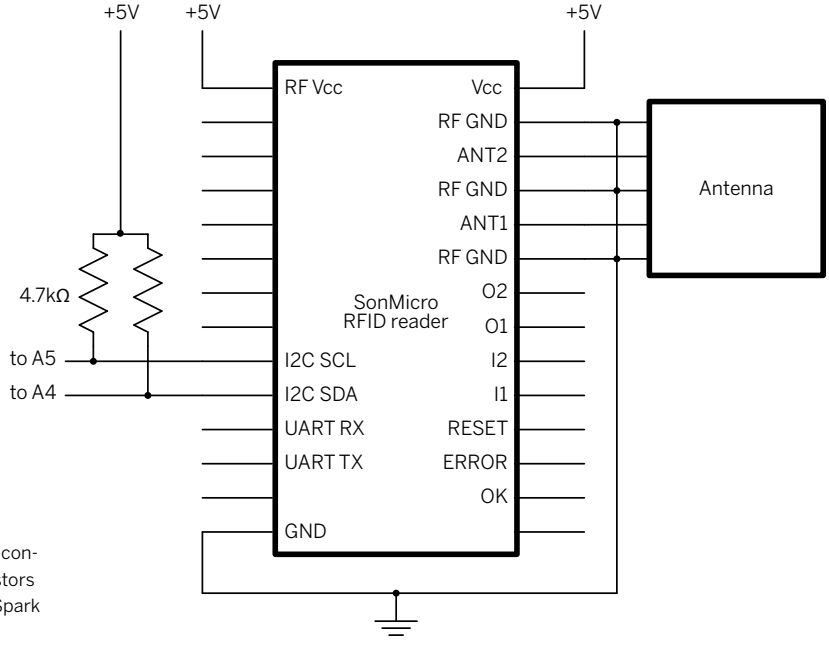


Figure 9-16
SonMicro RFID reader schematic. The I2C connections need pullup resistors. These resistors are built into the TinkerKit shield and the Spark Fun shield.

SonMicro Communications Protocol

The SonMicro readers use a binary communications protocol to communicate with other devices. The protocol is similar whether you're communicating over asynchronous serial or I2C. The I2C version looks like this:

Length	Command	Data	Checksum
1 byte	1 byte	N bytes	1 byte

The serial version just adds two extra header bytes. It looks like this:

Header	Reserved	Length	Command	Data	Checksum
1 byte	1 byte	1 byte	1 byte	N bytes	1 byte

The commands for the reader are single-byte values. The most common ones are shown below:

Byte value	Command
0x80	Reset
0x81	Get firmware version
0x82	Seek for tag
0x83	Select tag
0x85	Authenticate
0x86	Read memory block
0x89	Write memory block
0x90	Antenna power (turns on or off antenna)

The length byte indicates the length of the command, plus any data that goes with it. So, for example, a reset command—which has no data to follow the command—has a length of 1. A read command has one byte of data indicating the address you want to read from, so the length is 2.

The checksum is an error-checking value. To calculate the checksum, add together the command and data bytes; and if the value is greater than 255, take the lowest byte. For example, Here's the reset command's checksum:

```
0x01 (length) + 0x80 (command) = 0x81 (checksum)
```

If you were reading from memory address 04, the checksum would be as follows (note that these values are in hexadecimal):

```
0x02 (length) + 0x86 (command) + 0x04 (address) = 0x8C
(checksum)
```

For the asynchronous serial protocol, the header byte is always 0xFF, and the reserved byte is always 0. They're not added in the checksum, so you can use the same commands and calculations as you do for the I2C version, and just add 0xFF and 0x00 at the beginning. The replies sent back from the reader follow this same protocol.

There are both Processing and Arduino libraries to handle this protocol so you don't have to, but it's useful to know the basics. To start out, write a simple Processing sketch that sends the command to get the firmware version, and then reads the reply. Readers come with different firmware versions, and you might need to change the firmware on yours (see the sidebar on page 336), so you'll save a lot of troubleshooting time up front by knowing the firmware version.

When you run this sketch, you can see the responses in hexadecimal at the top, and in ASCII at the bottom (see Figure 9-17). The firmware command is the only one that returns any ASCII, because this is a binary protocol. The select tag command returns a binary string similar to the command protocol. It starts with the header and reserved byte (0xFF, 0x00), then the data length (0x06 if there's a tag, 0x02 if not), the command received (0x83), the tag type (0x02 means Mifare classic), four bytes representing the tag number (0x0A, 0xD4, 0xF0, 0x28, in my case), and the checksum (0x81). The ASCII representation, on the other hand, looks like garbage.

X

Talk to It This Processing sketch sends commands to the SM130 module, and reads the result. The SM130's default data rate is 19200bps. Type `s` to select a tag in the field, and type any other key to read the firmware version.

```
/*
 SonMicro Firmware version reader
 Context: Processing
 */
import processing.serial.*;
Serial myPort;           // The serial port
String binaryString = "";
String asciiString = "";

void setup() {
  size(500, 200);        // window size
  // List all the available serial ports
  println(Serial.list());
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[2], 19200);
}

void draw() {
  // Dystopian waterscape color scheme,
  // by arem114, http://kuler.adobe.com/
  background(#7B9B9D);
  fill(#59462E);
  // write the response, in hex and ASCII:
  text(binaryString, 10, height/3);
  text(asciiString, 10, 2*height/3);
}

void keyReleased() {
  int command = 0x81;    // read firmware command is the default
  int dataLength = 1;    // data length for both commands here is 1
  if (key == 's') {      //  "select tag" command
    command = 0x83;
  }
  // send command:
  myPort.write(0xFF);
  myPort.write(0x00);
  myPort.write(dataLength);
  myPort.write(command);
  myPort.write(command + dataLength);
  // reset the response strings:
  binaryString = "";
  asciiString = "";
}

void serialEvent(Serial myPort) {
  // get the next incoming byte:
  char thisChar = (char)myPort.read();
  // add the byte to the response strings:
  binaryString += hex(thisChar, 2) + " ";
  asciiString += thisChar;
}
```

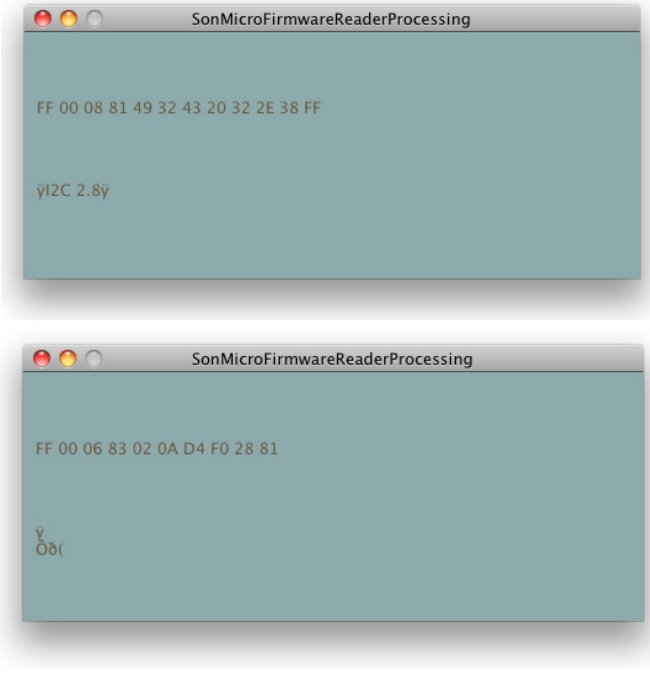


Figure 9-17
Screenshots of the Processing sketch to read SonMicro firmware, showing the results of the read firmware command (top) and the select tag command (bottom).

Writing to Mifare Tags

Now that you know a little about the SonMicro protocol, it's time to get on with writing data to the tags. To do this, you can use the Processing SonMicroReader library, which you can download from https://github.com/tigoe/SonMicroReader-for-Processing. Download the library, unzip it, copy the resulting folder to the **libraries** folder of your Processing sketch folder, and restart Processing. In the File → Examples menu, you should see a new entry for SonMicroReader under Contributed Libraries. The example called **SonMicroWriter0002.pde** follows.

This sketch has a graphical user interface with buttons for many of the common commands, and text responses in plain language as well as hexadecimal representations of the reader's responses. When you're working with a device that has a lot of functionality like this, sometimes it's worthwhile to write a tool to work all of its functions—or at least the most common ones.

When you run the sketch, you'll get the interface shown in Figure 9-18. Click the Firmware Version button to get the firmware version, just to make sure you've got good communication with the reader. Then try reading a few tags with the Select Tag button. When there's no tag present, you get a response from the reader saying that. When you choose Seek Tag, however, you get an initial response that the command's in progress. Then, when you bring a tag in range, it reads automatically without you having to send another command.

In addition to their tag IDs, which can't change, Mifare tags have a small amount of RAM that you can read from and write to. Mifare standard tags have 1 kilobyte, Mifare classic tags have 4 kilobytes, and Mifare Ultralight tags have 512 bits. The former two use encrypted communication for reading and writing, and you have to authenticate a tag before you can read from or write to it. When you authenticate, you gain access to one 16-byte block of memory at a time.

The sequence for reading or writing to a block of memory is always:

- Select Tag
- Authenticate
- Read Block (or write block)

The code for this sketch follows.

Changing the Firmware on the SonMicro Readers

SonMicro SM130 readers come with different firmware versions, depending on which reader you buy. Different vendors carry different models, so it's good to check your firmware right away. All firmware versions can communicate serially, so the sketch shown above is a good way to check. Details of the firmware versions are available on SonMicro's site, www.sonmicro.com, under Products → 13.56 MHz RFID - MIFARE → OEM Modules & Readers. If your firmware version is UM 1.3 or UM 1.3c, you'll need to change the firmware in order to control the module using I2C. If your firmware version is I2C 2.8 or another I2C version, you can do both I2C and asynchronous serial and skip the rest of this sidebar.

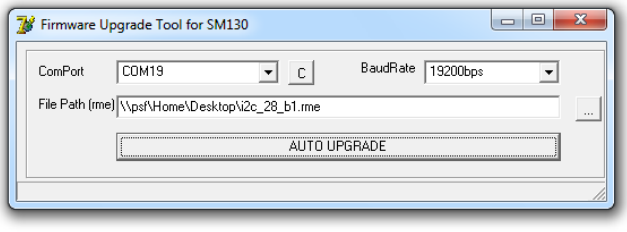
If you have to upgrade your firmware, you'll need a Windows PC and the Firmware Upgrade tool called SM13X FU from SonMicro's site (it's downloadable from the 13.56 MHz RFID Mifare Support page, on the Software tab), and the SMRFID Mifare v1.2 diagnostic software. Download them both, then unzip and install them. You'll also need the I2C firmware—to get that, you have to email SonMicro. They are fast and friendly in response, though, and there's a technical inquiries link you can use to contact them from the Support page.

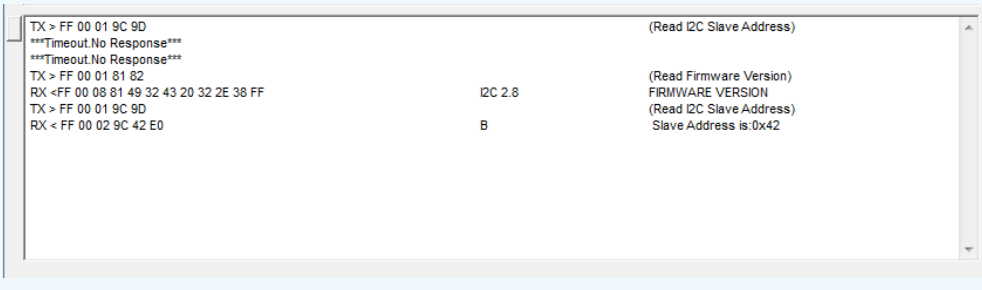
Connect your reader to your Windows PC using a USB-to-Serial adapter (the approach used for the Processing sketch above will work fine). Launch the SM13X FU application, click the "..." button, and browse to find the firmware file you want to upload. The file will be called i2c_28_b1.rme or something similar, depending on the version number. Then click Auto Upgrade. You should see messages at the bottom of the window ending with a successful upgrade.

But wait, you're not done yet. Close the firmware upgrader and open the SMRFID Mifare v1.2 application. Open the ComPort and choose Read I2C Address from the Hardware Commands menu. If the response is NOT 0x42, you should change it. Choose Set I2C Address and enter 42, then click OK. The reader's I2C address will be changed to 0x42, which is the default, and is what's used in the Arduino examples below.

You don't need to do any more with these applications to proceed with the project, but you may want to explore the SMRFID Mifare application. It's a useful tool for reading tags, writing to tags, and performing other diagnostic functions on the SM130 reader modules.

The SM Firmware Uploader (right) and a closeup of the SMRFID diagnostic application (below). When you've finished, you should see results in the SMRFID application like those shown here.





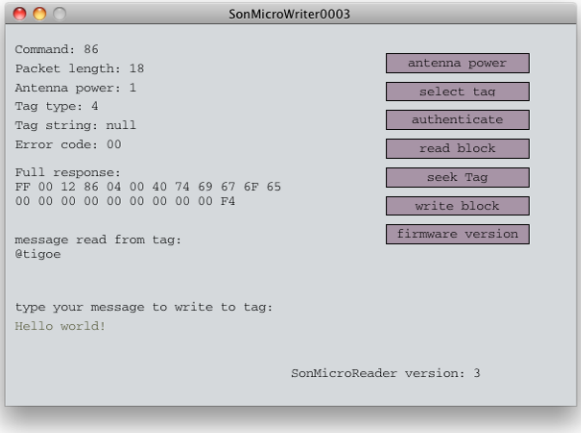


Figure 9-18
Screenshot of the Processing SonMicroWriter sketch, showing the response to a successful Read Block request.

Write to It

This Processing sketch lets you read from and write to the memory on a Mifare tag using the SM130 RFID reader. It uses the SonMicroReader library for Processing.

The global variables for this sketch are mostly properties of the last tag read. There is an ArrayList to keep track of the buttons onscreen, as well as a list of button names.

```
/*
 SonMicro RFID Writer example
 Context: Processing
 */

// import libraries:
import processing.serial.*;
import sonMicroReader.*;

String tagID = "";           // the string for the tag ID
Serial myPort;               // serial port instance
SonMicroReader myReader;     // SonMicroReader instance

int  lastCommand = 0;        // last command sent
int lastTagType = 0;         // last tag type received
int lastPacketLength = 0;    // last packet length received
String lastTag = null;       // last tag ID received
int lastErrorCode = 0;       // last error code received
int[] lastResponse = null;   // last response from the reader (raw data)
int lastAntennaPower = 0;    // last antenna power received
int lastChecksum = 0;        // last checksum received

int fontHeight = 14;         // font height for the text onscreen
String message = null;       // message read from tag
String outputString = "Hello world!";    // string to write to tag

// Color theme: Ghostly Music
// by banshee prime, http://kuler.adobe.com
color currentcolor = #CBD0D4;    // current button color
color highlight = #745370;
color buttoncolor = #968195;
color userText = #444929;
color buttonText = #ACB0B9;
```

Continued from previous page.

```
ArrayList buttons = new ArrayList();  // list of buttons
// the buttons themselves:
String[]  buttonNames = {
  "antenna power", "select tag", "authenticate", "read block", "seek Tag",
  "write block", "firmware version"
};
```

» setup() initializes the serial connection and makes an instance of the SonMicroReader library. Then it makes fonts for drawing text on the screen, and calls the makeButtons() function to make the onscreen buttons. The button creation and control functions come later, at the end of the sketch.

```
void setup() {
  // set window size:
  size(600, 400);
  // list all the serial ports:
  println(Serial.list());

  // based on the list of serial ports printed from the
  // previous command, change the 0 to your port's number:
  String portnum = Serial.list()[0];
  // initialize the serial port. default data rate for
  // the SM130 reader is 19200:
  myPort = new Serial(this, portnum, 19200);
  // initialize the reader instance:
  myReader = new SonMicroReader(this, myPort);
  myReader.start();

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

» draw() draws text and buttons to the screen.

```
void draw() {
  background(currentcolor);
  // draw the command buttons:
  drawButtons();
  // draw the output fields:
  textAlign(LEFT);
  text("Command: " + hex(lastCommand, 2), 10, 30);
  text("Packet length: " +lastPacketLength, 10, 50);
  text("Antenna power: " + lastAntennaPower, 10, 70);
  text("Tag type: " + lastTagType, 10, 90);
  text("Tag string: " + lastTag, 10, 110);
  text("Error code: " + hex(lastErrorCode, 2), 10, 130);

  // print the hex values for all the bytes in the response:
```



Continued from opposite page.

```
  String responseString = "";
  if (lastResponse != null) {
    for (int b = 0; b < lastResponse.length; b++) {
      responseString += hex(lastResponse[b], 2);
      responseString += " ";
    }
    // wrap the full text so it doesn't overflow the buttons
    // and make the screen all messy:
    text("Full response:\n" + responseString, 10, 150, 300, 200);
  }
  // print any error messages from the reader:
  text(myReader.getErrorMessage(), 10, 210);
  // print the last message read from the tag:
  text("last message read from tag:\n" + message, 10, 230);

  // print the output message:
  text("type your message to write to tag:\n", 10, 300);
  fill(userText);
  text(outputString, 10, 320);

  // show the library version:
  fill(0);
  text("SonMicroReader version: " + myReader.version(),
  width - 300, height - 30);
}
```

TheSonMicroReader library uses the Serial library to communicate with the reader—like the Serial library—it generates an event when new data is available. sonMicroEvent() occurs whenever there's a response from the reader. In this sketch, you read all the parts of the response in the sonMicroEvent() method, including the tag number, tag type, any data returned from reading a memory block, the antenna status, and more.

```
/*
 This function is called automatically whenever there's
 a valid packet of data from the reader
 */
void sonMicroEvent(SonMicroReader myReader) {
  // get all the relevant data from the last data packet:
  lastCommand = myReader.getCommand();
  lastTagType = myReader.getTagType();
  lastPacketLength =  myReader.getPacketLength();
  lastTag = myReader.getTagString();
  lastErrorCode = myReader.getErrorCode();
  lastAntennaPower = myReader.getAntennaPower();
  lastResponse = myReader.getSonMicroReading();
  lastChecksum = myReader.getCheckSum();

  // if the last command sent was a read block command:
  if (lastCommand == 0x86) {
    int[] inputString = myReader.getPayload();
    message = "";
    for (int c = 0; c < inputString.length; c++) {
      message += char(inputString[c]);
    }
  }
```

When you type something in the sketch window, it's added to a string. It's then sent to the RFID tag when you command the SM130 to write to the tag. The keyTyped() method attaches any keystrokes to the output string.

```
}
/*
  If a key is typed, either add it to the output string
 or delete the string if it's a backspace:
 */
void keyTyped() {
  switch (key) {
  case BACKSPACE:  // delete
    outputString = "\0";
    break;
  default:

    if (outputString.length() < 16) {
      outputString += key;
    }
    else {
      outputString = "output string can't be more than 16 characters";
    }
  }
```

The button functionality is handled by a separate Java class in the sketch called Button. There are also a few methods for managing the list of buttons, including makeButtons(), which creates them initially, and drawButtons(), which draws and updates them.

```
}
/*
 initialize all the buttons
 */
void makeButtons() {
  // Define and create rectangle button
  for (int b = 0; b < buttonNames.length; b++) {
    // create a new button with the next name in the list:
    Button thisButton = new Button(400, 30 +b*30,
                                   150, 20,
                                   buttoncolor, highlight, buttonNames[b]);
    buttons.add(thisButton);
  }
}
/*
  draw all the buttons
 */
void drawButtons() {
  for (int b = 0; b < buttons.size(); b++) {
    // get this button from the Arraylist:
    Button thisButton = (Button)buttons.get(b);
    // update its pressed status:
    thisButton.update();
    // draw the button:
    thisButton.display();
  }
```

mousePressed() checks to see whether one of the buttons is pressed. If so, it calls doButtonAction(), which sends the appropriate command for each button to the SM130.

```
}
void mousePressed() {
  // iterate over the buttons, activate the one pressed
  for (int b = 0; b < buttons.size(); b++) {
    Button thisButton = (Button)buttons.get(b);
    if (thisButton.containsMouse()) {
      doButtonAction(thisButton);
    }
  }
}

/*
  if one of the command buttons is pressed, figure out which one
 and take the appropriate action.
 */
void doButtonAction(Button thisButton) {
  // figure out which button this is in the ArrayList:
  int buttonNumber = buttons.indexOf(thisButton);

  // do the right thing:
  switch (buttonNumber) {
  case 0: //  set antenna power
    if (myReader.getAntennaPower() < 1) {
      myReader.setAntennaPower(0x01);
    }
    else {
      myReader.setAntennaPower(0x00);
    }
    break;
  case 1: // select tag
    myReader.selectTag();
    break;
  case 2:  // authenticate
    myReader.authenticate(0x04, 0xFF);
    break;
  case 3:   // read block
    myReader.readBlock(0x04);
    break;
  case 4:  // seek tag
    myReader.seekTag();
    break;
  case 5:  // write tag - must be 16 bytes or less
    myReader.writeBlock(0x04, outputString);
    outputString = "";
    break;
  case 6:  // get reader firmware version
    myReader.getFirmwareVersion();
    break;
  }
```

Finally, the Button class defines the properties and behaviors of the buttons themselves.

This sketch doesn't include all the functionality of the SM130. For example, it only writes to block 4 of a Mifare tag's memory, and it only uses the default authentication scheme. However, it does give you the ability to read and write from tags, as well as the structure to understand how to use the SM130's functions.

```
}
class Button {
  int x, y, w, h;                    // positions of the buttons
  color basecolor, highlightcolor;   // color and highlight color
  color currentcolor;                // current color of the button
  String name;

  // Constructor: sets all the initial values for
  // each instance of the Button class
  Button(int thisX, int thisY, int thisW, int thisH,
         color thisColor, color thisHighlight, String thisName) {
    x = thisX;
    y = thisY;
    h = thisH;
    w = thisW;
    basecolor = thisColor;
    highlightcolor = thisHighlight;
    currentcolor = basecolor;
    name = thisName;
  }

  // if the mouse is over the button, change the button's color:
  void update() {
    if (containsMouse()) {
      currentcolor = highlightcolor;
    }
    else {
      currentcolor = basecolor;
    }
  }

  // draw the button and its text:
  void display() {
    fill(currentcolor);
    rect(x, y, w, h);
    //put the name in the middle of the button:
    fill(0);
    textAlign(CENTER, CENTER);
    text(name, x+w/2, y+h/2);
  }

  // check to see if the mouse position is inside
  // the bounds of the rectangle:
  boolean containsMouse() {
    if (mouseX >= x && mouseX <= x+w &&
      mouseY >= y && mouseY <= y+h) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Reading from Mifare Tags

To prepare for the next part of this project, use the previous Processing sketch to write Twitter handles to a few tags in this format:

```
@moleitau
@Kurt_Vonnegut
@pomeranian99
```

To do so, use the sequence mentioned at the beginning of this section:

- Put the tag in the reader's field
- Click Select Tag
- Click Authenticate
- Click Write Block

Then, to verify it:

- Put the tag in the reader's field
- Click Select Tag
- Click Authenticate
- Click Write Block

When you're sure you've got a couple tags with Twitter handles on them, you're ready to build the next part of the project: an Arduino tweet reader.

Circuit Additions

The circuit is the same as in Figures 9-14 and 9-15, but with an Ethernet connection and a 2x16 LCD attached. If you're already using an Arduino Ethernet, you're all set. If not, add an Ethernet shield. Either way, add the LCD as shown in Figure 9-19. To make sure your LCD works, test it with any of the LiquidCrystal library examples included in the Arduino software, just by changing the pin numbers to match your own. The pin numbers for these examples were chosen so as not to conflict with the pins that the Ethernet controller and SD card use.

The potentiometer on Analog input 1 will be used to control the speed of scrolling on the LCD. Pins A0 and A2 will be used as digital outputs, to act as voltage and ground for this pot. Below is a quick example to test the LCD.

Display It This sketch prints the number of seconds since the sketch started on a 2x16-LCD screen attached to an Arduino.

```
/*
  LCD Example
  Context: Arduino
  */
// include the library:
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(9,8, 7, 6,5, 3);

void setup() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("I've been running for:");
}

void loop() {
  // set the cursor to column 0, line 1
  // (note: line 1 is the second row, since counting begins with 0):
  lcd.setCursor(0, 1);
  // print the number of seconds since reset:
  lcd.print(millis()/1000);
}
```

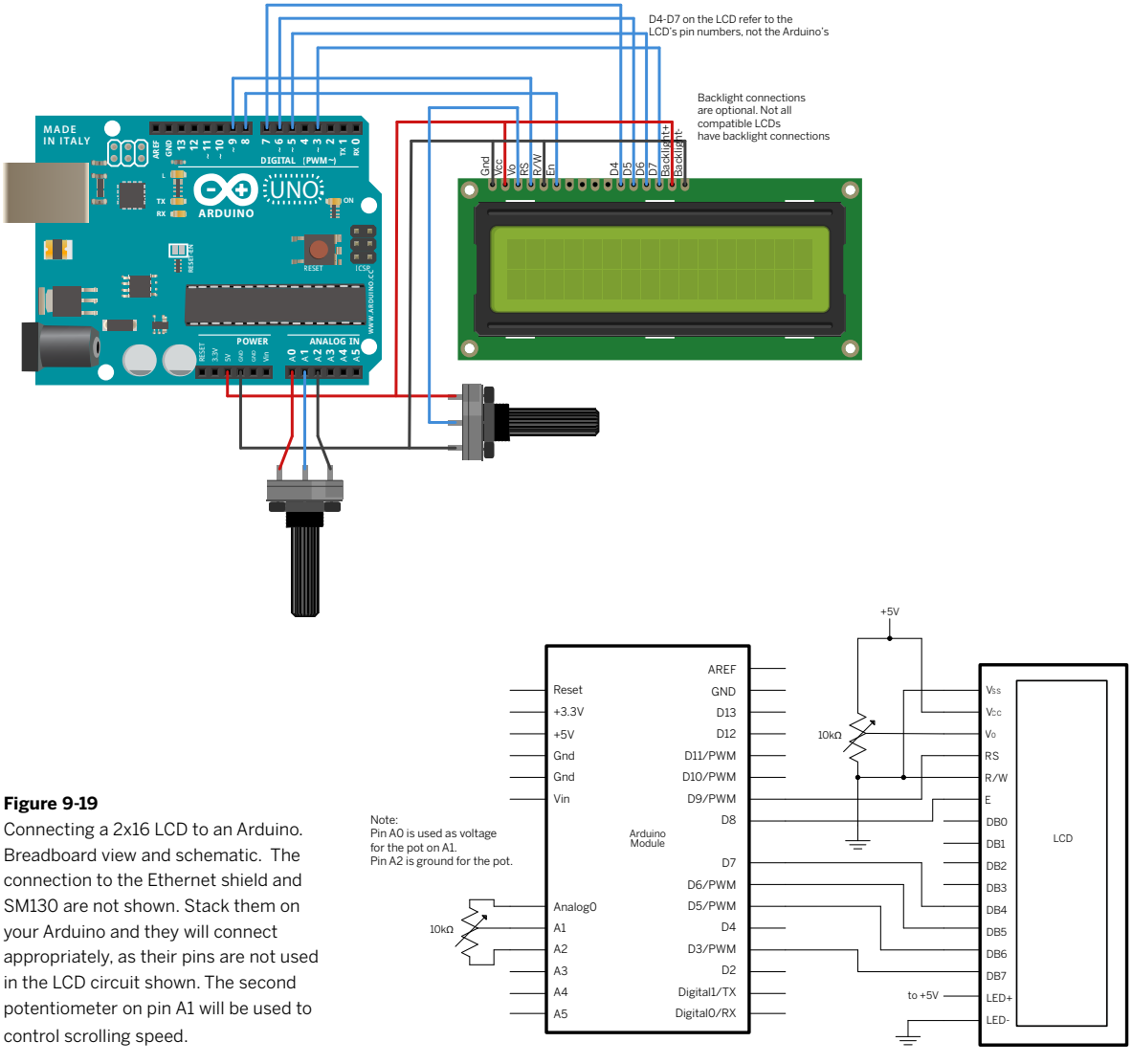


Figure 9-19
Connecting a 2x16 LCD to an Arduino. Breadboard view and schematic. The connection to the Ethernet shield and SM130 are not shown. Stack them on your Arduino and they will connect appropriately, as their pins are not used in the LCD circuit shown. The second potentiometer on pin A1 will be used to control scrolling speed.



If you're using the Ethernet shield rather than the Ethernet board, you might need to supply external power for this circuit. Together, the Ethernet shield, RFID shield, LCD display, and Arduino draw enough current that in the moment the RFID reader reads a tag, the circuit draws more current than USB can supply, causing the voltage to drop. In some cases, a 220µF capacitor across power and ground of the RFID shield will help, but if not, use an external power supply, a Power-over-Ethernet supply, or an Ethernet board instead of the Ethernet shield/Uno combination.

So far, you've only used the Arduino board as a serial pass-through, but for this phase of the project, you'll control the RFID reader using the I2C protocol. The Wire library that comes with Arduino lets you control I2C devices. For the main sketch, the Wire library commands will be wrapped inside another library that's specifically for the RFID reader. But to make sure things are working, and to see I2C in action, try the sketch below. It sends a command to the reader to ask for the firmware revision, and prints the results to the Serial Monitor. You'll recognize some of the SonMicro communications protocol in this sketch.

Read the Firmware

This sketch reads the firmware of the SM130 module.

```
/*
  SM130 Firmware reader
  Context: Arduino
*/
#include <Wire.h>

void setup() {
  // initialize serial and I2C:
  Serial.begin(9600);
  Wire.begin();
  // give the reader time to reset:
  delay(2000);

  Serial.println("asking for firmware");
  // open the I2C connection,
  // the I2C address for the reader is 0x42:
  Wire.beginTransmission(0x42);
  Wire.write(0x01);      // length
  Wire.write(0x81);      // command
  Wire.write(0x82);      // checksum
  Wire.endTransmission();

  // reader needs 50ms in between responses:
  delay(50);
  Serial.print("getting reply: ");
  // wait for 10 bytes back via I2C:
  Wire.requestFrom(0x42,10);
  // don't do anything until new bytes arrive:
  while(!Wire.available()) {
    delay(50);
  }
  // when new bytes arrive on the I2C bus, read them:
  while(Wire.available()) {
    Serial.write(Wire.read());
  }
  // add a newline:
  Serial.println();
}

void loop() {
}
```

When you run the firmware reader sketch, you should get a response like this in the Serial Monitor:

```
asking for firmware
getting reply: I2C 2.8ÿ
```

That last byte is the checksum of the string of bytes that the reader sent in response. If you get a good response, you're ready to move on. If not, revisit the sidebar on page 344.

It would be convenient if, instead of having to remember the numeric values for each command, you could just call the commands by name, like you can with the Processing library above. The Arduino SonMicro library lets you do just that. In fact, most of its methods share the same names as the Processing library's methods. Download the latest version from https://github.com/tigoe/SonMicroReader-for-Arduino. Make a new directory called **SonMicroReader** in the **libraries** directory of your Arduino sketch directory, and copy the contents of the download package to it. Then restart the Arduino application. The new library should appear in the Examples submenu of the File menu as usual. Once that's done, move on to the sketch below.

This sketch looks for RFID tags and reads block 4 when it finds a tag. If it finds a Twitter handle there, it makes an HTTP call to Twitter's XML API to get the latest tweet from that Twitter user. Then it displays the result on an LCD screen.

Saving Program Memory

The sketch for this project is complex, and it takes a large chunk of the Arduino's program memory. You'll notice that the Serial print statements have a new syntax, like this:

```
Serial.println(F("Hello"));
```

The F() notation tells the print and println statements to store the text that follows in flash memory, not in program memory. Because the print statements are for debugging purposes only and are not going to change, this saves memory for your program.

Get the Tweet

You'll be using a lot of libraries in this sketch. There are a few global variables associated with each one, starting with the current tag, the last tag read, and the address block from which to read on the tag.

The main loop() is a simple state machine. It does different things depending on what state it's in, so there's a variable to keep track of the state. There are four basic states:

- Looking for a tag
- Reading the tag you just got
- Making an HTTP request
- Waiting for the server's response

Regardless of what state the sketch is in, it should also update the LCD every time through the loop().

There are also the usual IP and Ethernet configuration variables.

```
/*
 Twitter RFID Web Client
 Context: Arduino
 */
#include <SPI.h>
#include <Ethernet.h>
#include <TextFinder.h>
#include <Wire.h>
#include <LiquidCrystal.h>
#include <SonMicroReader.h>

SonMicroReader Rfid;       // instance of the reader library
unsigned long tag = 0;     // address of the current tag
unsigned long lastTag = 0; // address of the previous tag
int addressBlock = 4;      // memory block on the tag to read

int state = 0;              // the state that the sketch is in

// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = { 0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress ip(192,168,1,20);          // will only be used if DHCP fails
IPAddress server(199,59,149,200);    // Twitter's API address
EthernetClient client;               // the client connection
```

» Change these to match your own device and network.

There are also global variables to keep track of the tweeter, the tweet, when you last made an HTTP request, and how long to delay between requests.

Finally, there are global variables for the various characteristics of the LCD display and constants for the scroll-delay potentiometer.

Continued from opposite page.

```
String twitterHandle = "";       // the tweeter
String tweet = "";               // the tweet
int tweetBufferLength;           // the space to reserve for the tweet
int tweetLength = 0;             // the actual length of the tweet

long lastRequestTime = 0;       // last time you connected to the server
int requestDelay = 15 * 1000;   // time between HTTP requests


// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(9,8, 7, 6,5, 3);
const int screenWidth = 16;         // width of the LCD in characters
long lastScrollTime = 0;            // last time you scrolled the LCD
int scrollDelay = 130;              // delay between LCD moves
int cursorPosition = 0;             // cursor position on the LCD

const int potVoltage = A0;          // voltage for the scroll delay pot
const int potGround = A2;           // ground for the scroll delay pot
const int potInput = A1;            // scroll delay pot
```

setup() initializes communications: serial, Ethernet, communications to the RFID reader via I2C, and the LCD. It also sets A0 high as a digital output, and A2 low, to supply voltage and ground for the scroll-delay potentiometer.

```
void setup() {
  // initialize serial communications and the reader:
  Serial.begin(9600);
  // start the Ethernet connection:
  if (Ethernet.begin(mac) == 0) {
    Serial.println(F("Failed to configure Ethernet using DHCP"));
    Ethernet.begin(mac, ip);
  }
  // set pins A0 and A2 to digital out, and use them as
  // power and ground for the scroll speed potentiometer:
  pinMode(potGround, OUTPUT);
  pinMode(potVoltage, OUTPUT);
  digitalWrite(potGround, LOW);
  digitalWrite(potVoltage, HIGH);

  // reserve 140 * 2 screenWidths + 3 bytes extra for tweet:
  tweetBufferLength = 140 + 2*screenWidth + 3;
  tweet.reserve(tweetBufferLength);
  Rfid.begin();
  // set up the LCD's number of columns and rows:
  lcd.begin(screenWidth,2);
  lcd.clear();
  lcd.print(F("Ready"));
}
```

loop() checks what state it's in using a switch statement. When it has finished what it needs to do in each state, it increments the state variable and moves to the next state on the next time through the loop(). If any of the later states don't succeed (for example, if it can't authenticate or read from the last tag it saw), it drops back to the first state, looking for a tag.

Once it has finished checking the state, loop() updates the LCD display. It prints the Twitter handle on the top line, and it scrolls the latest tweet on the bottom line, moving forward every scrollDelay milliseconds.

```
void loop() {
  switch(state) {
  case 0:    // get tag
    tag = Rfid.selectTag();
    if (tag != 0) {
      // you have a tag, so print it:
      Serial.println(tag, HEX);
      state++;    // go to the next state
    }
    break;
  case 1:    // read block
    if (Rfid.authenticate(addressBlock)) {
      Serial.print(F("authenticated "));
      // read the tag for the twitter handle:
      Rfid.readBlock(addressBlock);
      twitterHandle = Rfid.getString();
      // show the handle:
      lcd.clear();              // clear previous stuff
      lcd.setCursor(0,0);       // move the cursor to the beginning
      lcd.print(twitterHandle); // tweet handle on the top line
      Serial.println(twitterHandle);
      state++;    // go to the next state
    }
    else state = 0;   // go back to first state
    break;
  case 2:    //connect to server
    // if this is a new tag, or if the request delay
    // has passed since the last time you made an HTTP request:
    if (tag != lastTag ||
      millis() - lastRequestTime > requestDelay) {
      // attempt to connect:
      if (connectToServer()) {
        state++;    // go to the next state
      }
      else state = 0;   // go back to first state
    }
    else state = 0;   // go back to first state
    lastTag = tag;
    break;
  case 3:    // read response
    tweetLength = readResponse();
    state = 0;   // go back to first state
    break;
  }

  // if you haven't moved the LCD recently:
  if (tweetLength > 0 && millis() - lastScrollTime > scrollDelay) {
    // advance the LCD:
    scrollLongString(cursorPosition);
    // increment the LCD cursor position:
    if (cursorPosition < tweetLength) {
      cursorPosition++;
```

Continued from opposite page.

```
    }
    else {
      cursorPosition = 0;
    }
    // note the last time you moved the LCD:
    lastScrollTime = millis();
  }
```

Finally, it updates scrollDelay by mapping the input of the potentiometer on pin A1.

```
  // update the speed of scrolling from the second potentiometer:
  int sensorReading = analogRead(potInput);
  // map to a scrolling delay of 100 - 300 ms:
  scrollDelay = map(sensorReading, 0, 1023, 100, 300);
}
```

The scrollLongString() method takes a 16-character substring of the long tweet and displays it on the LCD display. It pads the beginning and end of the tweet with enough characters so that it scrolls all the way on and off the screen.

```
// this method takes a substring of the long
// tweet string to display on the screen
void scrollLongString(int startPos) {
  String shortString = "";           // the string to display on the screen
  // make sure there's enough of the long string left:
  if (startPos < tweetLength - screenWidth) {
    // take a 16-character substring:
    shortString = tweet.substring(startPos, startPos + screenWidth);
  }
  // refresh the LCD:
  lcd.clear();              // clear previous stuff
  lcd.setCursor(0,0);       // move the cursor to beginning of the top line
  lcd.print(twitterHandle); // tweet handle on the top line
  lcd.setCursor(0,1);       // move cursor to beginning of the bottom line
  lcd.print(shortString);   // tweet, scrolling, on the bottom
}
```

The connectToServer() method is very similar to the connect method in the air-quality client in Chapter 4. It connects to the server and makes an HTTP GET request. In this case, it's not asking for an HTML page, but for an XML version of the tweet that you can get from http://api.twitter.com. Twitter, like many sites, makes it possible to get both a human-readable form of their site and a machine-readable one, delivered in XML.

```
// this method connects to the server
// and makes an HTTP request:
boolean connectToServer() {
  // note the time of this connect attempt:
  lastRequestTime = millis();
  // attempt to connect:
  Serial.println(F("connecting to server"));
  if (client.connect(server, 80)) {
    Serial.println(F("making HTTP request"));
    // make HTTP GET request:
    client.print(F("GET /1/statuses/user_timeline.xml?screen_name="));
    client.print(twitterHandle);
    client.println(F(" HTTP/1.1"));
    client.println(F("Host:api.twitter.com"));
    client.println();
    return true;
  }
```

Continued from previous page.

```
    else {
      Serial.print(F("failed to connect"));
      return false;
    }
  }
```

The readResponse() method reads incoming bytes from the server after the HTTP request, and looks for the beginning and end of a tweet. The tweet is conveniently sandwiched between two tags: <tweet> and </tweet>. So, all you have to do is look for those and return what's between them. You can ignore the rest of the stream. If you want to see the whole XML feed, open the URL the sketch is using in a browser, and give it a Twitter handle at the end, like this:

http://api.twitter.com/1/statuses/user_timeline.xml?screen_name=halfpintingalls

```
int readResponse() {
  char tweetBuffer[141];    // 140 chars + 1 extra

  int result = 0;
  // if there are bytes available from the server:
  while (client.connected()) {
    if (client.available()) {
      // make an instance of TextFinder to search the response:
      TextFinder response(client);
      // see if the response from the server contains <text>:
      response.getString("<text>", "</text>", tweetBuffer, 141);
      // print the tweet string:
      Serial.println(tweetBuffer);
      // make a String with padding on both ends:
      tweet = "                " + String(tweetBuffer)
        +  "               ";
      result = tweet.length();
      // you only care about the tweet:
      client.stop();
    }
  }
  return result;
}
```

Troubleshooting

There are a lot of things that can go wrong in this sketch, so make sure you're clear on each of the parts, and have a plan for troubleshooting. Below are a few things to look for:

Is it reaching each of the four major states? If not, determine at which one it's stopping. Have the sketch print a message when it reaches each new state, as well as the results of the previous state.

Is it reading a tag? Can it read the tag's memory blocks? If not, try one of the example sketches that comes with the SonMicroReader library. In particular, the ReadBlock example was written to do a basic read of tags.

Can it reach the server? If not, try the HTTP test client in Chapter 4 with the URL that this sketch calls.

Can it read the user's tweets? Are you sure you didn't pick a Twitter user who protects her tweets? Or who hasn't tweeted yet? Put the URL in a browser and search for the <tweet> tags yourself. If they're not there, the sketch can't see them.

Is the LCD working? Separate this out using the Serial Monitor. This sketch contains a number of print statements to let you know what's happening. Use them when troubleshooting.

Construction

This is a simple mat board box like the ones in Chapter 5, but the structure is different. The RFID antenna and the LCD screen are attached to the microcontroller by wires. You can use ribbon wire, or you can use an old Ethernet cable with the ends chopped off.

Make sure to test the circuit before you put it in the box!

Mount the LCD and the potentiometer that controls contrast on a separate perfboard, as shown in Figure 9-19. You can use a prototyping shield if you want, but it's not necessary; you can just attach headers to the wires and plug them directly into the board. You'll want to group the wires to make them easier to plug in, like so:

- Solder the wires for Arduino pins D8 and D9 to two headers.
- Solder the wires for pins D3 through D7 to a rwo of five headers, with an extra space for pin D4.
- Solder the wires for +5V and ground to a pair of headers.

Figure 9-20 shows the wires broken out of an Ethernet cable and attached to headers using this sequence.

Connect the scroll-speed potentiometer to pins A0 through A2 as shown in Figure 9-18 using any type of wire you have handy.

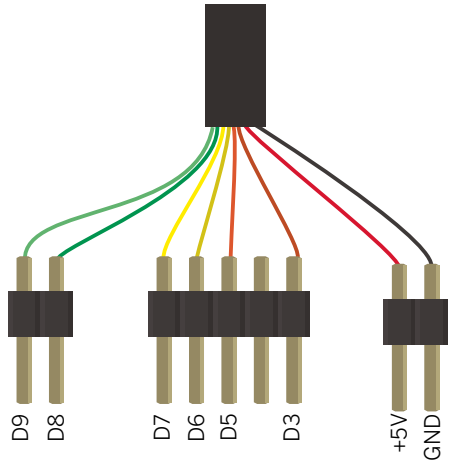
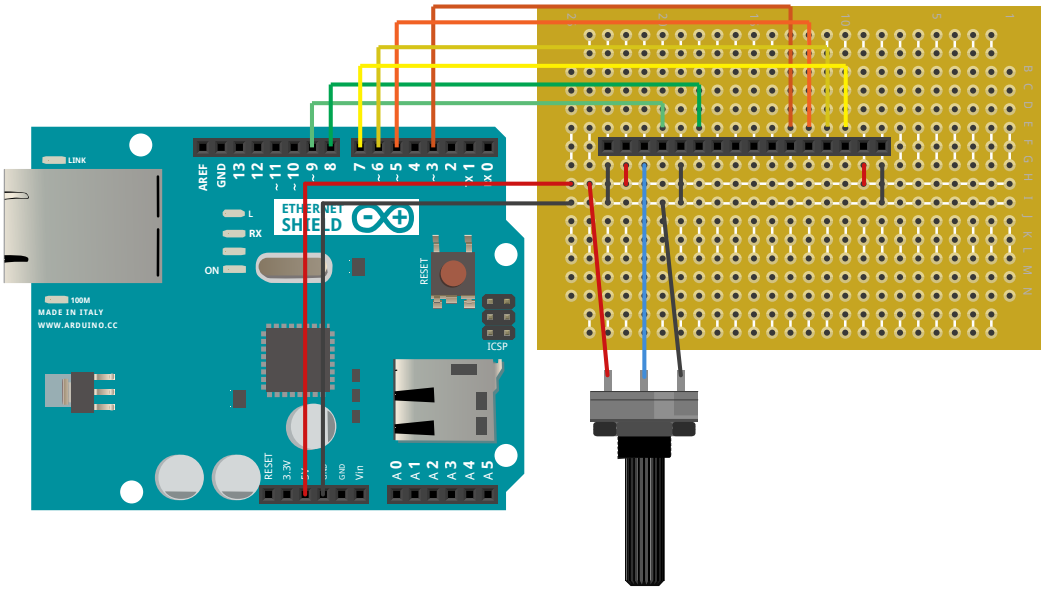


Figure 9-21
The header pin connections for the cable to the perfboard.

Figure 9-20
The LCD and potentiometer connections to a perfboard. The wire colors show which wires correspond to those in Figure 9-18.



Attach the RFID antenna to the top of the box with Velcro. A set of extra long female headers provides enough length to hold it above the wires of the LCD circuit. You can see the inside of the box in Figure 9-22.

Mount the potentiometers to the top of the box, held in place by their mounting nuts. Mount the LCD screen to the top of the box using Velcro or double-stick tape on the side of the screen itself. A tab of Velcro on the bottom of Arduino holds it nicely in place on the bottom of the box as well.

Once you're done, you've got a handy Twitter reader for your desk, as shown in Figure 9-23. Write your favorite Twitter users to a series of tags so you've got them handy to check at a moment's notice. You've made a connection between tangible identification using the tags, and network identification using the Twitter feed.

X

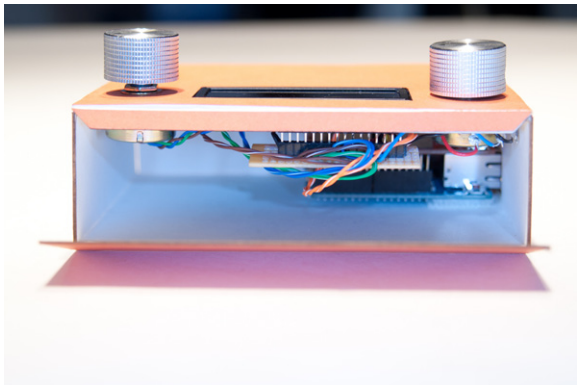


Figure 9-22
The inside of the box with the LCD perfboard in the foreground.

Figure 9-23
The final box.



Network Identification

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

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

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

HTTP Environment Variables

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

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

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

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

You should get something like this in your browser:

```
#CGI __utmz: 152494652.1295382826.13.2.utmccn=(referral)
|utmcsr=itp.nyu.edu|utmcct=/physcomp/studio/Spring2011/
TomIgoe|utmcmd=referral
__utma: 152494652.402968136.1288069605.1308754712.130876886
1.29
PATH: /usr/local/bin:/usr/bin:/bin
REDIRECT_HANDLER: php-cgi
REDIRECT_STATUS: 200
UNIQUE_ID: Thtgla3sqiUAAFdwW-UAAAAP
SCRIPT_URL: /php/09_env.php
SCRIPT_URI: http://www.example.com/php/09_env.php
HTTP_HOST: www.example.com
HTTP_CONNECTION: keep-alive
HTTP_USER_AGENT: Mozilla/5.0 (Macintosh; Intel Mac
OS X 10_6_8) AppleWebKit/534.30 (KHTML, like Gecko)
Chrome/12.0.742.112 Safari/534.30
HTTP_ACCEPT: text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8
HTTP_ACCEPT_ENCODING: gzip,deflate,sdch
HTTP_ACCEPT_LANGUAGE: en-US,en;q=0.8
HTTP_ACCEPT_CHARSET: ISO-8859-1,utf-8;q=0.7,*;q=0.3
HTTP_COOKIE: __utmz=152494652.1295382826.13.2.utmccn=(refer
ral)|utmcsr=itp.nyu.edu|utmcct=/physcomp/studio/Spring2011/
TomIgoe|utmcmd=referral; __utma=152494652.402968136.12880696
05.1308754712.1308768861.29
SERVER_SIGNATURE:
SERVER_SOFTWARE: Apache
SERVER_NAME: www.example.com
SERVER_ADDR: 77.248.128.3
SERVER_PORT: 80
REMOTE_ADDR: 66.168.47.40
DOCUMENT_ROOT: /home/username/example.com
SERVER_ADMIN: webmaster@example.com
SCRIPT_FILENAME: /home/username/example.com/php/09_env.php
REMOTE_PORT: 52138
REDIRECT_URL: /php/09_env.php
GATEWAY_INTERFACE: CGI/1.1
SERVER_PROTOCOL: HTTP/1.1
REQUEST_METHOD: GET
QUERY_STRING:
REQUEST_URI: /php/09_env.php
SCRIPT_NAME: /php/09_env.php
ORIG_SCRIPT_FILENAME: /dh/cgi-system/php5.cgi
ORIG_PATH_INFO: /php/09_env.php
ORIG_PATH_TRANSLATED: /home/username/example.com/php/09_env.
php
ORIG_SCRIPT_NAME: /cgi-system/php5.cgi
PHP_SELF: /php/09_env.php
REQUEST_TIME: 1310417045
```

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

For example, HTTP_USER_AGENT tells you the name of the software browser with which the client connected. From that, you can determine whether if it's a mobile phone, desktop, or something else, and serve appropriate content for each. HTTP_ACCEPT_LANGUAGE tells you the language in which the client would like the response. When you combine REMOTE_ADDR with the IP geocoding example to follow, you can even make a reasonable estimation as to where the client is, assuming its request did not come through a proxy.

Project 28

IP Geocoding

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

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

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

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

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

» The website www.hostip.info will return the latitude and longitude from the IP address in a convenient XML format. The latitude and longitude are inside a tag called <gml:coordinates>. That's what you're looking for. First, format the HTTP request string and make the request. Then wait for the results in a while loop, and separate the results into the constituent parts.

```
    // use http://www.hostIP.info to get the latitude and longitude
    // from the IP address.  First, format the HTTP request string:
    $IpLocatorUrl = "http://api.hostip.info/?&position=true&ip=";
   // add the IP address:
    $IpLocatorUrl .= $ipAddress;

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

      // as long as you haven't reached the end of the incoming text:
   while (!feof($filePath)) {
       // read one line at a time
       $line = fgets($filePath, 4096);
      // if the line contains the coordinates, then you want it:
       if (preg_match('/<gml:coordinates>/', $line)) {
          $position = strip_tags($line);         // strip the XML tags
          $position = trim($position);           // trim the whitespace
          $coordinates = explode(",",$position); // split on the comma
          $lat = $coordinates[0];
          $long = $coordinates[1];
       }
    }
```

»

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

Continued from previous page.

```
    // close the connection:
    fclose($filePath);

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

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

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

First, connect to the server as you did before:

```
telnet example.com 80
```

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

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

You should get a response like this:

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

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

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

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

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

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

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

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

Mail Environment Variables

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

Like HTTP, email protocols have environment variables that you can take advantage of as well. If you've ever viewed the full headers of an email in your favorite mail client, you've seen some of these. To look at mail in more depth, you can use PHP's IMAP mail functions. Internet Message Access Protocol is a mail protocol that lets you get mail from a server from multiple clients. The mail stays on the server until a client tells the server to delete a particular message. This allows you to use multiple mail clients for the same account, and keep mail coordinated across them.

Send It Put the following PHP script on your server.

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

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

// open a connection to your gmail inbox on port 993
// using SSL, but no certificate to validate the connection:
$mailServer = '{imap.gmail.com:993/imap/ssl/novalidate-cert}INBOX';
$mbox = imap_open($mailServer, $user, $pass);

// get the message headers in the inbox:
$headers = imap_headers($mbox);

// if there are no messages, there is no mail:
if ($headers == false) {
   echo "Failed to get mail<br />\n";
} else {
   // print the number of messages
   echo "number of messages: ".imap_num_msg($mbox);
   echo "\n\n";
   // print the header of the first message:
echo imap_fetchheader($mbox,1);
}
// close the inbox:
imap_close($mbox);
?>
```

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

```
<?php
$user='username';        // your mail login
$pass='password';        // exactly as you normally type it
?>
```

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

```
chmod go-rwx pwds.php
```



Figure 9-24
Permissions for the **pwds.php** file. Make sure that no one can read from and write to it—besides you.

NOTE: **If you're using a graphic SFTP or FTP client, your settings for this file will look like Figure 9-24. This protection will deter anyone who doesn't have access to your account from getting your account info. It isn't an ideal security solution, but it serves for demonstration purposes and can be made more secure by changing your password frequently.**

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

```
number of messages: 85

Delivered-To: tom.igoe@gmail.com
Received: by 10.52.188.138 with SMTP id ga10cs129118vdc;
Sat, 28 May 2011
 12:32:30 -0700 (PDT)
Received: by 10.42.176.136 with SMTP id
be8mr4324248icb.15.1306611150331; Sat,
 28 May 2011 12:32:30 -0700 (PDT)
Return-Path: <tigoe@algenib.myhost.com>
Received: from myhost.com (crusty.g.myhost.
com [67.225.8.42]) by mx.google.com with ESMTP id
f8si10612848icy.106.2011.05.28.12.32.29; Sat,  28 May 2011
12:32:29 -0700 (PDT)
Received-SPF: pass (google.com: domain of tigoe@algenib.
myhost.com designates 67.225.8.42 as permitted sender)
client-ip=67.225.8.42;
Authentication-Results: mx.google.com; spf=pass (google.com:
domain of tigoe@algenib.myhost.com designates 67.225.8.42 as
permitted sender) smtp.mail=tigoe@algenib.myhost.com
Received: from algenib.myhost.com (algenib.myhost.com
[173.236.170.37]) by crusty.g.myhost.com (Postfix) with
ESMTP id 6EAC3BE813 for <tom.igoe@gmail.com>; Sat, 28 May
2011 12:31:15 -0700 (PDT)
Received: by algenib.myhost.com (Postfix, from userid
1774740) id 00AD1156BB6; Sat, 28 May 2011 12:32:09 -0700
(PDT)
To: tom.igoe@gmail.com
Subject: Hello world!
From: cat@catmail.com
Message-Id: <20110528193210.00AD1156BB6@algenib.myhost.com>
Date: Sat, 28 May 2011 12:32:10 -0700 (PDT)
```

There's a lot of useful information in this header. Though the mail says it's from cat@catmail.com, it's actually from a server that's run by myhost.com. It's common to put an alias on the from: address, to assign a different reply-to: address than the from: address, or both. It allows sending from a script such as the **cat** script in Chapter 3, yet the reply goes to a real person who can answer it. It's important to keep this in mind if you're writing scripts that reply to each other. If you're using email to communicate between networked devices, the program for each device must be able to tell the from: address from the reply-to: address—otherwise, they might not get each other's messages.

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

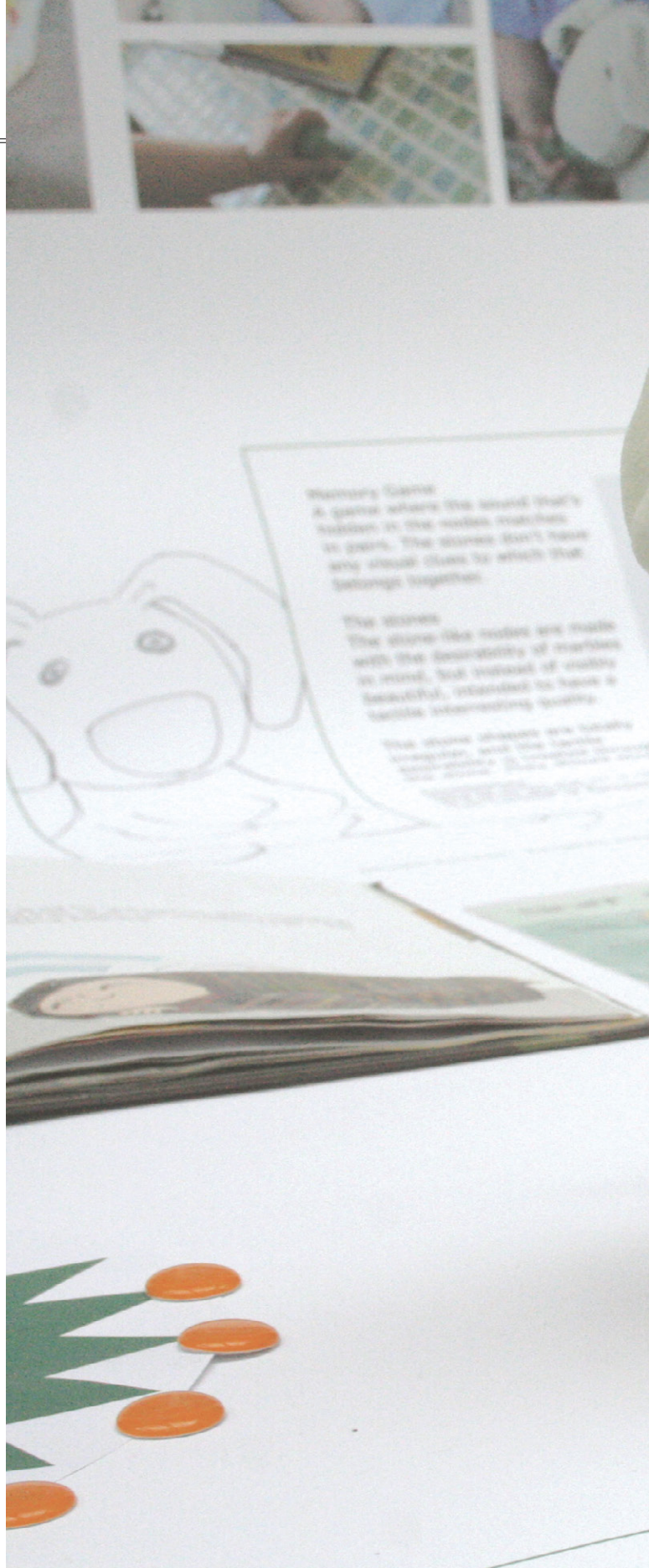
Conclusion

The boundary between physical identity and network identity always introduces the possibility for confusion and miscommunication. No system for moving information across that boundary is foolproof. Establishing identity, capability, and activity are all complex tasks, so the more human input you can incorporate into the situation, the better your results will be.

Security is essential when you're transmitting identifying characteristics because it maintains the trust of the people using what you make and keeps them safe. Once you're connected to the Internet, nothing's truly private and nothing's truly closed, so learning to work with the openness makes your life easier. In the end, keep in mind that clear, simple ways of marking identity are the most effective, whether they're universal or not. Both beginners and experienced network professionals often get caught on this point, because they feel that identity has to be absolute and clear to the whole world. Don't get caught up in how comprehensively you can identify things at first. It doesn't matter if you can identify someone or something to the whole world—it only matters that you can identify them for your own purposes. Once that's established, you've got a foundation on which to build.

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

X







10

MAKE: PROJECTS

Mobile Phone Networks and the Physical World

Ethernet and WiFi are handy ways to talk to people and things on the Internet, but there's a great big chunk missing: the mobile telephone network. Nowadays, telephony and the Internet are so intertwined that it doesn't make sense to talk about them separately. It's getting increasingly easy to connect physical devices other than mobile phones through mobile phone networks. In this chapter, you'll learn how to connect these two networks, and when it's useful to do so.

ohai lion, how r u??? txt me l8r!!!

This lion can send you an SMS. Groundlab, in conjunction with Living with Lions and Lion Guardians, developed this tracking collar that utilizes a GPS/GSM module to transmit lions' locations via SMS to researchers and Maasai herders. This open source system aims to help conservationists protect the last 2,000 lions living in the wild in Southern Kenya, and safeguard the Maasai herders' cattle, restoring Maasai land to a working ecosystem. *Photo courtesy of Groundlabs.*

Supplies for Chapter 10

There's not a lot that's brand new in this chapter. Many of the basic hardware parts will seem familiar. You will get the chance to work with 120V or 220V AC, though, and to work with conductive fabrics and threads. You'll also learn about reading and writing from an SD card.

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww/)
- **AF** Adafruit (http://adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **L** Less EMF (www.lessemf.com)
- **MS** Maker SHED (www.makershed.com)
- **RS** RS (www.rs-online.com)
- **SF** Spark Fun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 29: CatCam Redux

» **1 Arduino Ethernet board** **A** A000050
Alternatively, an Uno-compatible board (see Chapter 2) with an Ethernet shield will work.
SF DEV-09026, **J** 2124242 , **A** 139, **AF** 201, **F** 1848680

» **SD card reader that can read MicroSD** Ethernet shields and the Arduino Ethernet have one onboard.

» **MicroSD card** Available at most any electronics store.

» **IP-based camera** The examples use a D-Link DCS-930L.

» **Temperature sensor** **AF** 165, **D** TMP36GT9Z-ND, **F** 1438760, **RS** 427-351



Figure 10-1. New parts for this chapter: **1.** IP-based camera **2.** Hoodie **3.** Conductive ribbon **4.** Conductive thread **5.** LilyPad Arduino Simple **6.** Lithium Polymer battery charger **7.** Shieldex conductive fabric **8.** Embroidery thread **9.** Lithium Polymer battery **10.** Android device **11.** TMP36 temperature sensors **12.** MicroSD card **13.** MicroSD card reader **14.** Relay board **15.** AC junction box and socket **16.** AC cord.

» **Relay Control PCB** This is used to mount the rest of the components below. Alternately, you can use the Power Switch Tail below.
SF COM-09096
» **1 Relay** **SF** COM-00101, **D** T9AV1D12-12-ND, **F** 1629059
» **2 1-kilohm resistors** Any model will do.
D 1.0KQBK-ND, **J** 29663, **F** 1735061, **RS** 707-8669
» **1 10-kilohm resistor** **D** 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906
» **1 1N4148 diode** **SF** COM-08588, **F** 1081177, **D** 1N4148TACT-ND, **RS** 544-3480
» **1 2N3906 PNP-type transistor** **J** 178618, **D** 2N3906D26ZCT-ND, **SF** COM-00522, **F** 1459017, **RS** 294-328
» **1 LED** **D** 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590
» **1 2-pin Screw Terminal** **SF** PRT-08432, **D** 732-2030-ND, **F** 1792766, **RS** 189-5893
» **1 3-pin Screw Terminal** **SF** PRT-08235, **D** 732-2031-ND, **F** 1792767, **RS** 710-0166
» **1 Power Switch Tail** This is an alternative device to the relay board, relay, and support parts above. A 240V version is available from www.powerswitchtail.com.
SF COM-09842, **AF** 268, **MS** MKPS01
» **Cat**
» **Air conditioner**
» **Country estate (optional)**

PROJECT 30: Phoning the Thermostat
» **Completed Project 27**
» **Twilio account**

PROJECT 31: Personal Mobile Datalogger
» **Android device** You'll need a device running version 2.1 or later. See http://wiki.processing.org/w/Android for details.
» **LilyPad Arduino** **SF** DEV-09266, **A** A000011,
» **1 Bluetooth Mate module** **SF** WRL-09358 or WRL-10393
» **Lithium Polymer Ion Battery** **SF** PRT-00341, **AF** 258, **RS** 615-2472, **F** 1848660
» **1 270-kilohm resistor** **J** 691446, **D** P270KBACT-ND, **RS** 163-921, **F** 1565367
» **Conductive ribbon** **SF** DEV-10172
» **Thick conductive thread** **SF** DEV-10120, **L** A304
» **Shieldit Super Conductive Fabric** **L** A1220-14
» **Velcro**
» **Hoodie** **MS** MKSWT
» **Embroidery thread** Available at most fabric or yarn shops.

One Big Network

Before the Internet, there was the telephone network. All connections were analog electrical circuits, and all phone calls were circuit-switched, meaning that there had to be a dedicated circuit between callers. Then modems came along, which allowed computers to send bits over those same analog circuits. Gradually, switchboards were replaced with routers, and now telephone networks are mostly digital as well. Circuits are virtual, and what takes place behind the scenes of your phone calls is not that different from what occurs behind the scenes of your email or chat conversation: a session is established, bits are exchanged, and communication happens. The difference between a phone call and an email is now a matter of network protocols, not electrical circuits.

There are plenty of IP-based telephony tools that blur the line between phone call and Internet connection, including the open source telephony server Asterisk (www.asterisk.org), as well as telephony services from companies like Twilio (www.twilio.com), Google Voice (www.voice.google.com), and Skype (www.skype.com). These voice services are compatible with those offered by your phone company. What the phone company is giving you on top of the software service is a network of wires and routers that prioritizes voice services, so you are guaranteed a quality of service that you don't always get on IP-only telephony services.

Telephony services and Internet services meet on gateway servers and routers that run software to translate between protocols. For example, mobile phone carriers all offer SMS gateway services that allow you to send an email that becomes an SMS, or to send an SMS that emails the person you want to reach. Google Voice and many of the other online telephony services offer voicemail-to-text, in which an incoming call is recorded as a digital audio file, then run through voice-recognition software and turned into text, and finally emailed to you. The major task when building projects that use the telephony network is learning how to convert from one protocol to another.

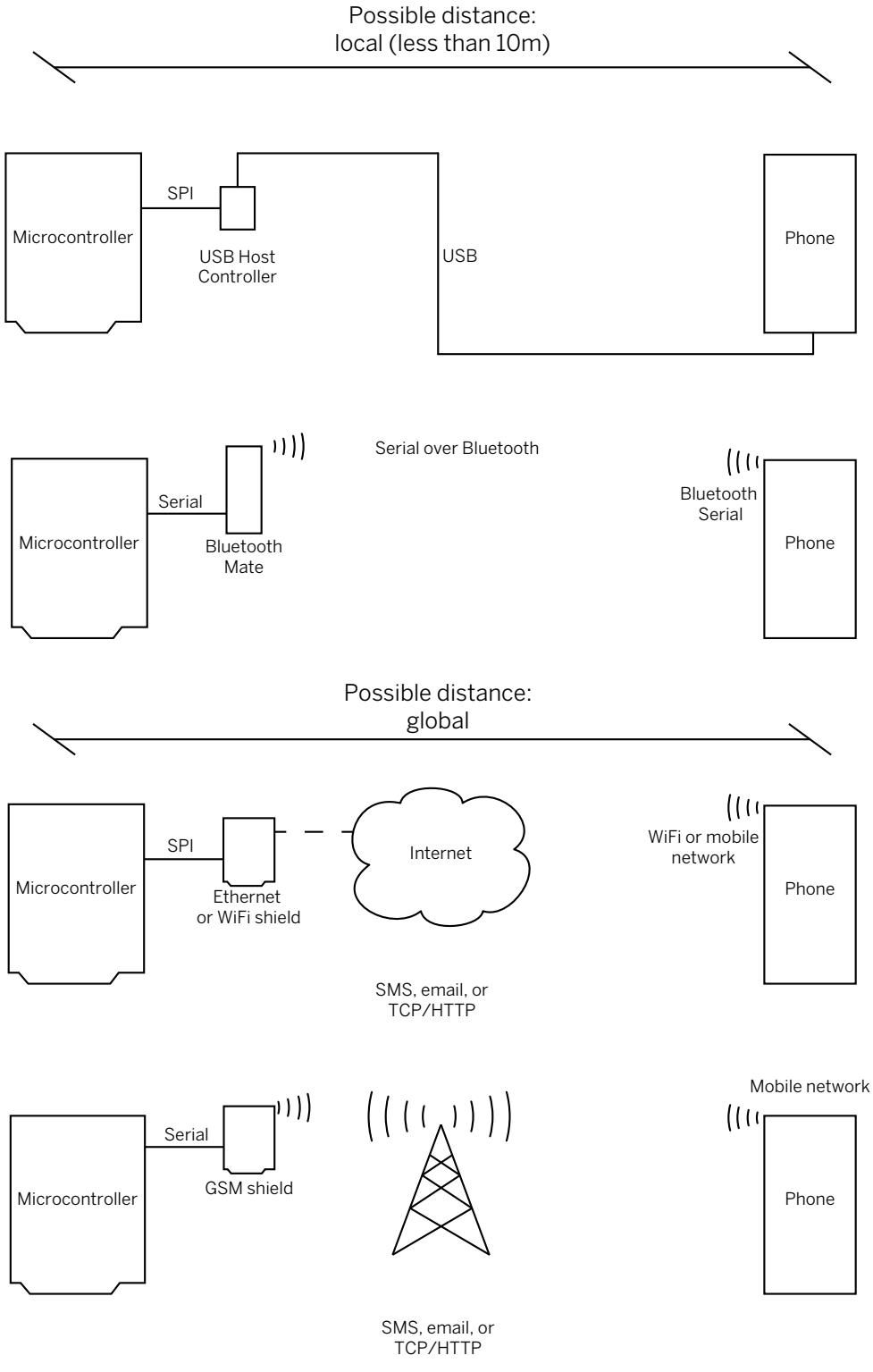
A Computer in Your Pocket

Mobile phones are far more than just phones now. The typical smartphone—such as an Android phone, iPhone, Blackberry, or Windows Phone—is a computer capable of running a full operating system. The processing power is well beyond that of older desktop machines, and smartphones run operating systems that are slightly stripped down versions of what you find on a laptop, desktop, or tablet computer. Most smartphones also incorporate some basic sensors, such as a camera, accelerometer, light sensor, and sometimes GPS. And, of course, they are networked all the time. What they lack, however, is the capability for adding sensors, motors, and other actuators—the stuff for which microcontrollers are made. When you treat the phone as a multimedia computer and mobile network gateway, you open up a whole host of possibilities for interesting projects.

Interfacing phones and microcontrollers can be done in a number of ways, depending on the phone's capabilities. For the purposes of this chapter, I'll be talking about smartphones, so you can assume most or all of the following capabilities, many of which you'll use in the projects to come:

- Programmability
- Touchscreen or keyboard
- Mobile network access
- Bluetooth serial port
- USB connection
- Microphone, speaker
- Onboard accelerometer
- Onboard GPS

Figure 10-2
Possible ways of linking microcontrollers and mobile phones.


Possible distance:
local (less than 10m)
Microcontroller
SPI
USB Host
Controller
USB
Phone
Microcontroller
Serial
Bluetooth
Mate
Serial over Bluetooth
Bluetooth
Serial
Phone
Possible distance:
global
Microcontroller
SPI
Ethernet
or WiFi shield
Internet
WiFi or mobile
network
Phone
SMS, email, or
TCP/HTTP
Microcontroller
Serial
GSM shield
Mobile network
Phone
SMS, email, or
TCP/HTTP

There are a few possible system arrangements afforded by these capabilities and the kinds of things you might want to make. Figure 10-2 shows the four most common arrangements.

Three of the four options shown use wireless links. Current mobile phones don't make it easy to make a wired serial link to external devices, though that is beginning to change. So, for most applications, a wireless link is the better option. It's not often that you'd prefer your phone be tethered to your project.

Start with What Happens

By now this should be a common theme to you: start with what you want people to do with the thing you're building, and design whatever makes that easiest to do. Are you making a system where your users have real-time interactive control over a nearby model helicopter or boat? Are they controlling a remote device like their home thermostat while on vacation? Consider how tight an interactive coupling you need. Do you need real-time interaction like you saw in Chapter 5, or will updates every few seconds or minutes be enough? And what about the distance between the devices—do you need to be able to communicate at long range, or will a local connection like Bluetooth or USB suffice? If it's a remote connection, how will your users know what happened on the other side? Can they see the result? Do they get an email or SMS message telling them about it? Evaluate whether you need a wired or wireless connection, based on how the people need to move and manipulate the devices involved. Consider what networks are available where you plan to use your system. Got WiFi? Got mobile networks? Got Ethernet? Can you create a local network between the two devices using Bluetooth or WiFi?

Browser Interfaces

If you want to control an existing networked device from a mobile phone, the easiest thing to do is to give the device an Internet connection and control it from your phone's browser. You don't have to learn anything specific to a particular phone operating system to do this—you just need to be able to make an HTTP connection between them. Make the microcontroller a simple server like you saw in Project 6, Hello Internet!, and you're done.

Native Application Interfaces

If you don't want to use the browser, you can build your own application on the phone to interact with the microcontroller via Bluetooth, network, SMS, or USB. You'll see an example of this using Processing for Android later in the chapter. The advantage of building a dedicated app is that you have complete control over what happens. The disadvantage is the challenge of learning a new programming environment, as well as the limitation of not being able to use your program on a phone with a different operating system. Because most phone operating systems don't allow the browser to access the communications hardware of the phone directly, you have to build an app if you plan to use local connections like Bluetooth or USB.

SMS and Email Interfaces

Not every project requires real-time interaction between your phone and the thing you're building. If you just want to get occasional updates from remote sensors, or you want to start a remote process that can then run by itself—like turning on a sprinkler or controlling your house lights via a remote timer—SMS or email might be useful. You can make a microcontroller application that can receive SMS messages using a GPRS modem, or you can write a networked server that checks email and sends messages to an Internet-connected microcontroller when it gets a particular email message. The advantage of this method is that you can control remote things using applications that are already on your phone, email, and SMS.

Voice Interfaces

Don't forget the phone's original purpose: two-way communication of sound over distance. Voice interfaces for networked devices can be simple and fun. Telephony gateways like Asterisk and Twilio allow you to use voice calls to generate networked actions.

Phone As Gateway

Not all applications in this area involve control of a remote device from the phone. Sometimes you just want to use the mobile phone as a network connection for sensors on your body or near you. In these cases, you'll make a local connection from the microcontroller to the phone, most likely over Bluetooth or USB, and then send the data from those sensors to a remote server. In these cases, you probably don't need to build a complex application—you just need to make the connection between the two devices, and between the phone and the network.

Project 29

CatCam Redux

If the phone has a web browser, you don't need to know how to program it to make a mobile application. In this project, you'll make a variation on the cat cam from Project 5. But this time, you'll use a microcontroller that serves files from an SD card and an IP-based camera that needs no computer to connect to the Internet.

MATERIALS

- **Arduino Ethernet or Arduino Uno and Ethernet shield**
- **SD card reader that can read MicroSD**
- **MicroSD card**
- **IP-based camera**
- **Temperature sensor**
- **Relay Control PCB**
- **Relay**
- **2 1-kilohm resistors**
- **1 10-kilohm resistor**
- **1 diode**
- **1 transistor**
- **1 LED**
- **1 2-pin Screw Terminal**
- **1 3-pin Screw Terminal**
- **Cat**
- **Air conditioner**
- **Country estate (optional)**

My neighbor, let's call her Luba, has a cat named Gospodin Fuzzipantsovich. Luba has a country estate with cherry trees. She likes to summer there, at least on summer weekends, but she can't take Fuzzipantsovich with her. Because the country estate has huge expenses (samovars and the like), Luba can't afford an air conditioner with a thermostat for her city place. So, I made her a web-based temperature monitor. It works as follows.

Luba opens a browser on her phone or laptop and logs into the thermostat's web interface. She sees the current temperature in her city apartment and the air conditioner's state, along with a camera view of Fuzzipantsovich. If it's too hot, she sets the trigger temperature lower. If it's too cool, she sets it higher. The interface updates the thermostat, and shows her the new trigger point.

Before you build this project, you should diagram the devices and protocols involved, and break the action into steps to understand what needs to happen. Figure 10-3 shows the system, and Figure 10-4 details the interaction.

There are several pieces of hardware involved and protocols to communicate between them. Many of them are familiar to you by now. The Ethernet controller on the Ethernet shield (or Arduino Ethernet)

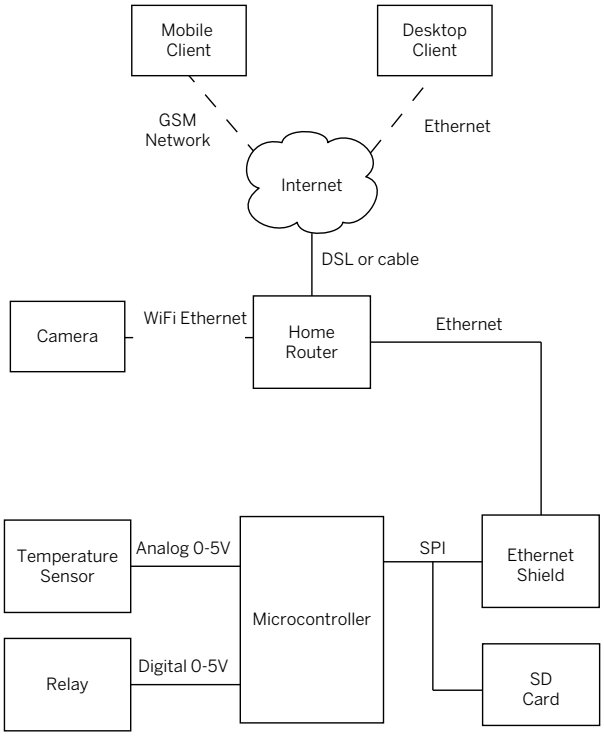


Figure 10-3
The system diagram for the CatCam 2 and air conditioner controller.

communicates with the microcontroller via SPI, as does the SD card on the shield or board. The temperature sensor is an analog input, and the 120V relay to control the air conditioner is a digital output. The IP camera in this project is a consumer item that has a built-in server, so all you have to do is configure it to speak to your router. The clients are just web browsers on your smartphone, tablet, or personal computer.

Even though it seems simple enough, there are actually 10 steps to the interaction, which are listed below:

1. The client makes an HTTP GET request for the main interface page (**index.htm**).
2. The microcontroller server reads the page from the SD card.
3. The server reads the temperature...
4. ... and the current thermostat trigger point.
5. The server sends the resulting page to the client.
6. Because the cat cam image is embedded in the page, the client requests it from the IP camera.
7. The IP camera sends the image to the client.
8. The client changes the trigger point via a form on the page, and submits it using HTTP POST.
9. The server reads the POST request and updates the thermostat.
10. The server sends back a response page.

Some of this, like getting the camera image and the resulting page, can be solved using HTML. You won't need to do any extra programming, because it will be taken care of by the browser when it interprets the HTML. The main programming challenge is to build a basic web server on the Arduino that serves files from the SD card.

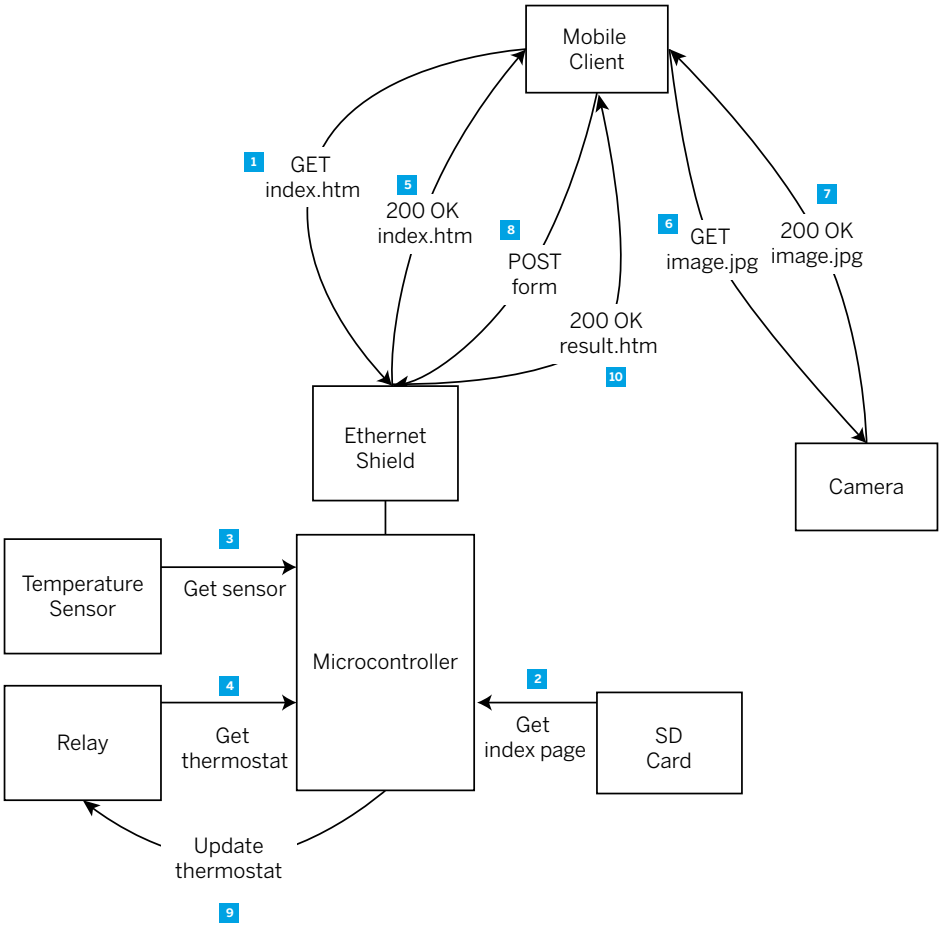


Figure 10-4
The interaction diagram for the CatCam 2 and air conditioner controller.

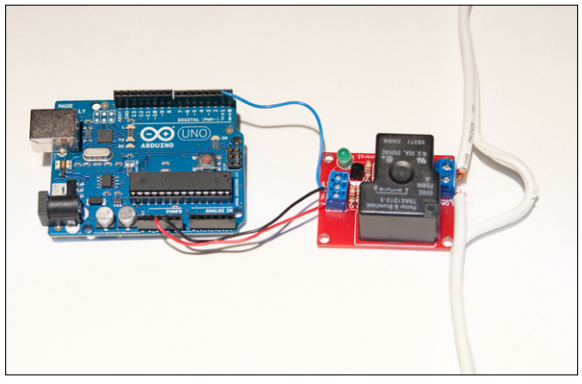


Figure 10-5
The relay board is connected to an AC cord, which has been cut on one side to attach it to the relay.

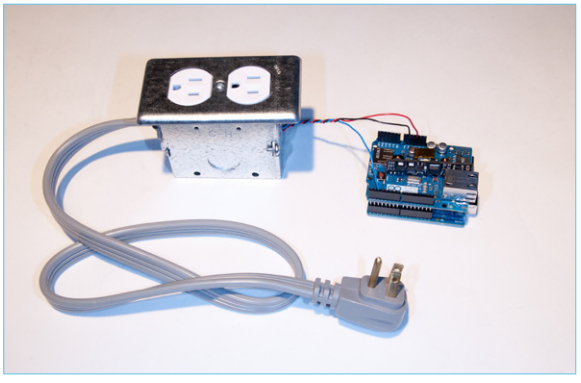
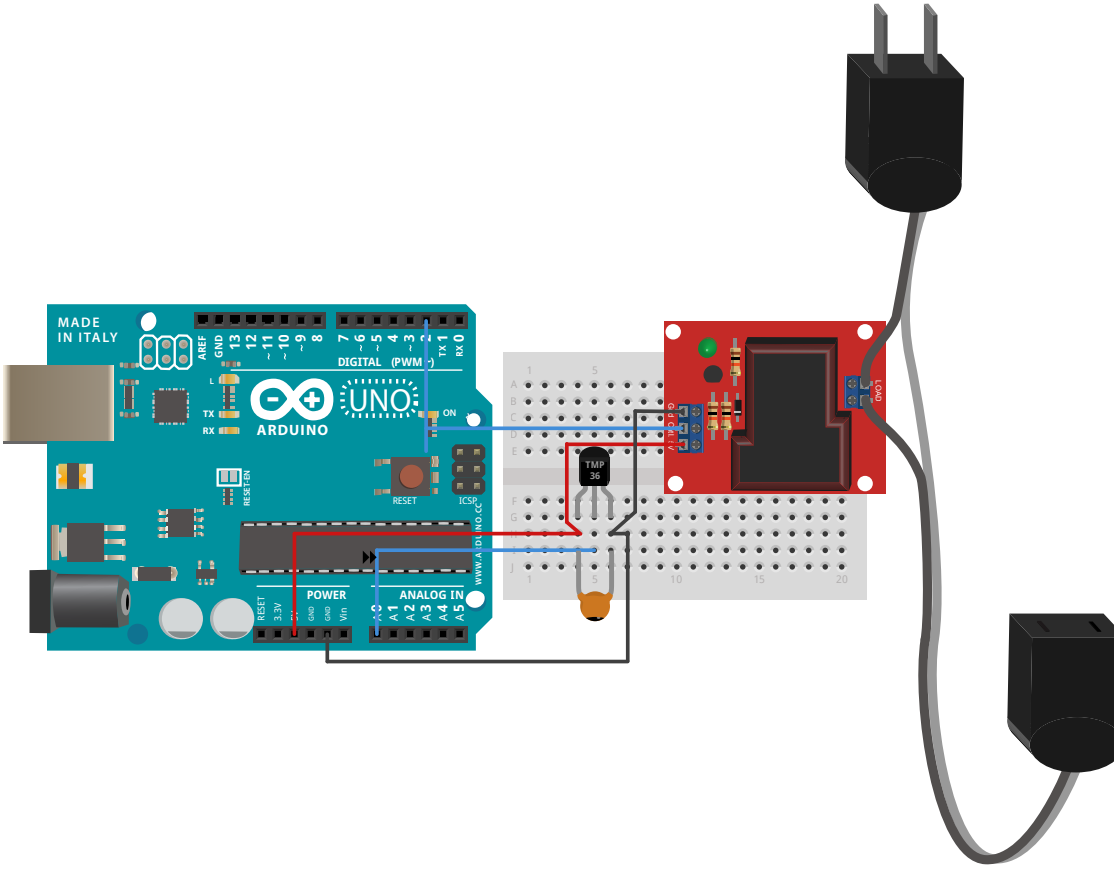


Figure 10-6
An AC junction box with the relay inside, and the control lines from the relay connecting to an Arduino. This is the safer way to build this project.

Figure 10-7
The CatCam 2 and air conditioner circuit.



The Circuit

Once you've got a grasp on the plan and the system, you're ready to build it. The circuit is shown in Figure 10-7. Because the SD card is on the Ethernet module (Ethernet shield or Arduino Ethernet), you only need to add the temperature sensor and the relay.

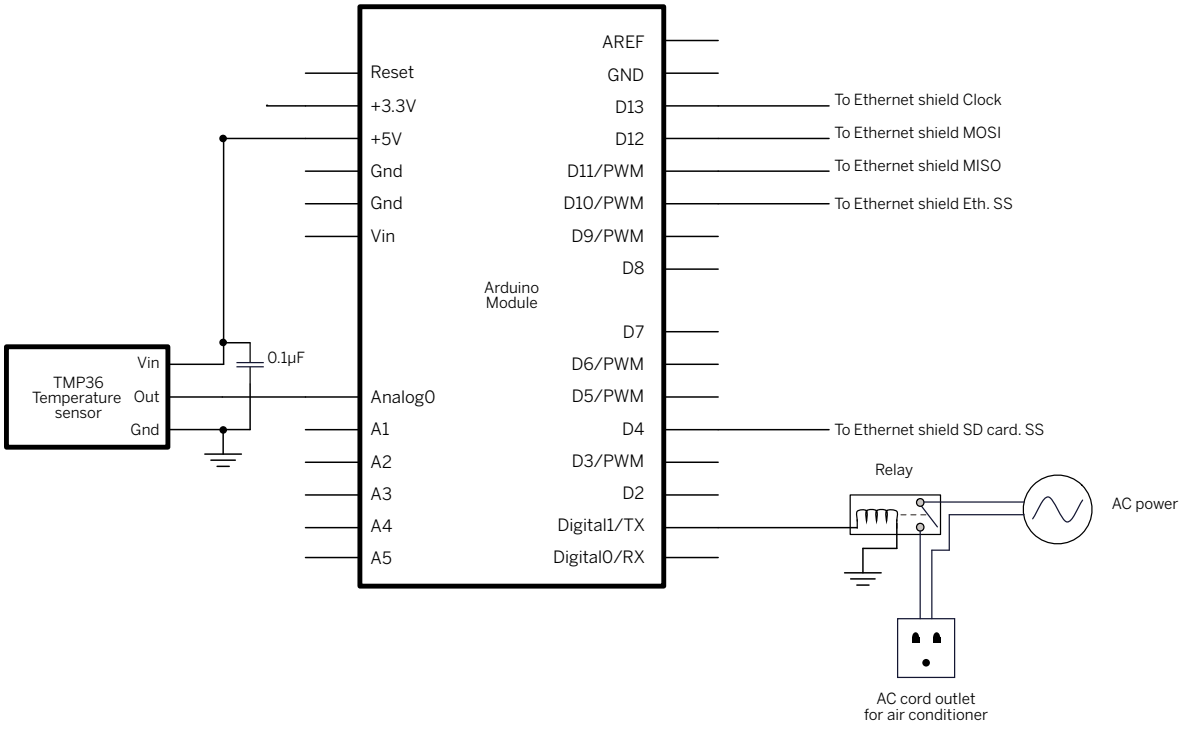
The relay contains a coil wrapped around a thin switch. When the coil gets voltage, it forms a magnetic field through induction, pulling the two sides of the switch together. To use it, take an alternating current circuit (AC) and break one of the wires, as shown in Figures 10-5 through 10-7.

Working with AC can be dangerous, so make sure everything is connected properly before plugging it in. If you're uncomfortable building an AC circuit, the Power Switch Tail will allow you to control 120V AC. A 240V version was recently announced at www.powerswitchtail.com. It's available from Adafruit, Spark Fun, and the Maker SHED.

Spark Fun's AC relay board is a safe and less expensive option if you want to build your own circuit. Solder the relay to the relay board. Then take an AC cord, cut one of the two wires, and attach it to the relay board's LOAD connections. Figure 10-5 shows a close-up of a household AC cord attached to the relay. The relay's ground, control, and 5V connections connect to your microcontroller. To build a more robust version—such as the one shown in Figure 10-6—with a household electrical outlet junction box, see Nathan Seidle's excellent tutorial at www.sparkfun.com/tutorials/119.

To test the circuit safely *before you plug it in*, set your multimeter to measure continuity, and to measure between one of the plug's pins and the corresponding socket hole on the other end of the cord. One pin/hole pair should have continuity all the time. The other should not, unless you power the 5V and ground contacts of the relay and connect the control line to 5V. Then you'll get continuity. When you connect the control line to ground, you won't get continuity. When you're sure the circuit's working, test it by attaching the control line to a pin of your microcontroller and use the Blink sketch to turn the relay on and off.

Figure 10-8
The CatCam 2 and air conditioner schematic.



Make sure to install your circuit in an insulated box and add plenty of strain relief to the AC lines. If they accidentally disconnect, they can cause a short circuit—and AC shorts are not pleasant. An electronic project case is safest. You can add strain relief to the wires by using hot glue or rubber caulking, though some cases come with wire strain relief glands to hold the wire in place. Once you've done that, give the wires a tug to make sure they won't move. Take your time and get this part very secure before you proceed.

The Code

Since the sketch is complex, it's built up piece-by-piece below, as follows:

1. Read the temperature sensor (for more on this sensor, see Adafruit's tutorial at www.ladyada.net/learn/sensors/tmp36.html).
2. Control the relay.
3. Read from the SD card.
4. Write a web server. This one will be more full-featured than the one you wrote in Chapter 4.

Try It Reading the temperature sensor is pretty simple. It outputs an analog voltage that's proportional to the temperature. The formula is:

```
Temp(°C) = (Vout (in mV) - 500) / 10
```

Since you'll need to do other things in the main loop, read the temperature and convert it to Celsius in a separate function. Return the result as a float.

Your reading will look like this:

```
Temperature: 26.17
```

The reading is in Celsius, but if you want to convert it to Fahrenheit, the formula is:

```
F = 9/5*C + 32
```

When you run the sketch at this point, you'll get a continual printout of the temperature in the Serial Monitor.

Next, it's time to add the relay control.

```
/*
 TMP36 Temperature reader and relay control
 Context:  Arduino
 Reads a TMP36 temperature sensor
 */

void setup() {
  // initialize serial communication:
  Serial.begin(9600);
}

void loop() {
   Serial.print("Temperature: ");
   Serial.println( readSensor() );
}

// read the temperature sensor:
float readSensor() {
  // read the value from the sensor:
  int sensorValue = analogRead(A0);
  // convert the reading to volts:
  float voltage = (sensorValue *  5.0) / 1024.0;
  // convert the voltage to temperature  in Celsius
  // (100mv per degree - 500mV offset):
  float temperature = (voltage - 0.5) * 100;
  // return the temperature:
  return temperature;
}
```

Control It The relay should be switched only every few seconds at most, so when you add code to control it, wrap the check in an if statement that checks to see whether an appropriate delay has passed. There are some global variables and constants to add, as well as changes to setup() and loop(). New lines are shown in blue.

It would be useful to be able to store the thermostat value, even when the Arduino is not powered. Adding the EEPROM library will allow that. At the top of the sketch, you'll read from EEPROM to get the thermostat value. Later on, you'll store new values back to EEPROM.

Add a new method, checkThermostat(), at the end of your sketch. It checks the temperature and compares it to a set thermostat point. If the temperature is greater, it turns the relay on. If the temperature is less, the relay is turned off.

When you run the sketch at this point, you'll see the relay turn on or off depending on the thermostat value. The value read from the EEPROM will probably be 255, so the relay won't turn on. Try changing the thermostat value to something less than the temperature sensor's reading and uploading your code again to see what happens.

```
#include <EEPROM.h>
const int relayPin = 2;                 // pin that the relay is attached to
const long tempCheckInterval = 10000;  // time between checks (in ms)
const int thermostatAddress = 10;       // EEPROM address for thermostat
long now;                               // last temperature check time
// trigger point for the thermostat:
int thermostat = EEPROM.read(thermostatAddress);

void setup() {
  // initialize serial communication:
  Serial.begin(9600);
  // initialize the relay output:
  pinMode(relayPin, OUTPUT);
}

void loop() {
  // periodically check the temperature to see if you should
  // turn on the thermostat:
  if (millis() - now > tempCheckInterval) {
    Serial.print("Temperature: ");
    Serial.println(readSensor());
    if (checkThermostat()) {
      Serial.println("Thermostat is on");
    }
    else {
      Serial.println("Thermostat is off");
    }
    now = millis();
  }
}

// NOTE: the readSensor() method shown earlier goes here.

// Check the temperature and control the relay accordingly:
boolean checkThermostat() {
  // assume the relay should be off:
  boolean relayState = LOW;
  // if the temperature's greater than the thermostat point,
  // the relay should be on:
  if(readSensor() > thermostat) {
    relayState = HIGH;
  }
  // set the relay on or off:
  digitalWrite(relayPin, relayState);
  return relayState;
}
```

Read It Next, it's time to read from the SD card. To do this, you'll need to add the SD library. On the Arduino shields that have an SD card, the SD Chip Select pin is pin 4. It varies for other manufacturers' shields, though, so make sure to check if you're using a different company's SD shield.

First, check whether the card is present using SD.begin(). If it's not, there's not much point in continuing. If it is, you can read from it. Add a new method, sendFile(), which will take an array of characters as a filename. New lines are shown in blue.

This new code won't work until you have a properly formatted MicroSD card inserted in the shield with a file on it called **index.htm**. Put the bare bones of an HTML document in the file, like so:

```
<html>
   <head>
      <title>Hello!</title>
   </head>
   <body>
   Hello!
   </body>
</html>
```

When you run the sketch at this point, it will print out the **index.htm** file in the Serial Monitor at the beginning, then go into the temperature reading and relay control behavior from the previous page.

```
#include <EEPROM.h>
#include <SD.h>
const int sdChipSelect = 4;            // SD card chipSelect

// NOTE: the constants and global variables shown earlier must go here

void setup() {
  // initialize serial communication:
  Serial.begin(9600);
  // initialize the relay output:
  pinMode(relayPin, OUTPUT);
  if (!SD.begin(sdChipSelect)) {
    // if you can't read the SD card, don't go on:
    Serial.println(F("initialization failed!"));
  }
  else {
    Serial.println(F("initialization done."));
    sendFile("index.htm");
  }
}

// NOTE: the loop(), readSensor(), and checkThermostat() methods shown
// earlier go here.

// send the file that was requested:
void sendFile(char thisFile[]) {
  String outputString = "";      // a String to get each line of the file

  // open the file for reading:
  File myFile = SD.open(thisFile);
  if (myFile) {
    // read from the file until there's nothing else in it:
    while (myFile.available()) {
      // add the current char to the output string:
      char thisChar = myFile.read();
      outputString += thisChar;

      // when you get a newline, send out and clear outputString:
      if (thisChar == '\n') {
        Serial.print(outputString);
        outputString = "";
      }
    }
    // close the file:
    myFile.close();
  }
  else {
    // if the file didn't open:
    Serial.print("I couldn't open the file.");
  }
}
```

Writing to an SD Card

Arduino's SD library makes it easy to write to an SD card, and there are several SD card-mount options available. All of them use the SPI synchronous serial protocol that the Arduino Ethernet shield and Ethernet board use to communicate with the Ethernet chip. As you saw in Chapter 4, this protocol can be used with a number of different devices, all attached to the same serial clock and data pins, but each device will need its own Chip Select pin. The Arduino communicates with all SPI devices through pin 11 for MOSI (Master Out, Slave In), Pin 12 for MISO (Master In, Slave Out), and pin 13 for Clock. The Arduino Ethernet shield and board use pin 10 for the Ethernet module's Chip Select, and pin 4 for the SD card's Chip Select. SD boards from other companies use other pins for chip select. Spark Fun's SD card shield uses pin 8, and Adafruit's SD card breakout board (shown in Figure 10-9) uses pin 10. Whenever you're using multiple SPI devices in a project, like the Ethernet and the SD card together, you need to check to see that they all have individual Chip Select pins.

SD cards operate on 3.3V only! So if you're using a socket that attaches your SD card directly to the pins of any microcontroller operating 5 volts, you need to make sure the input pins receive only 3.3V.

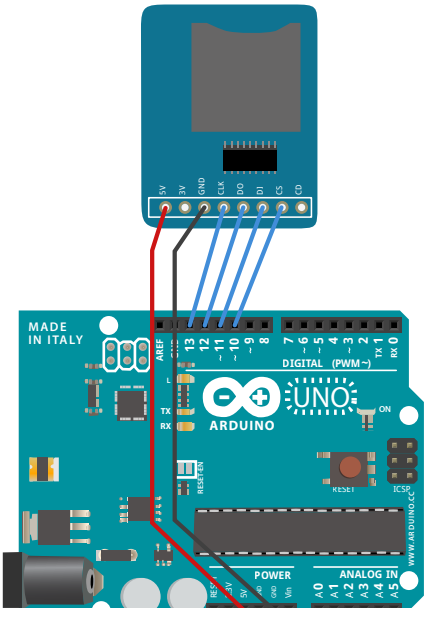


Figure 10-9
The Adafruit SD card shield connected to the Arduino using the SPI pins 11 (MOSI), 12 (MISO), and 13 (Clock). Different SPI card boards use different pins for the Chip Select pin, however. This one uses pin 10.

Good SD Card Practice

To get the most reliable results out of an SD card shield or an adapter with an Arduino, there are some habits you should develop.

Never insert or eject the card while the sketch is running. You'd never pull a card out of your computer while it's in use, and the same goes for the Arduino. Make sure the microcontroller's not using the card when you insert or remove it. The simplest way to do this is to hold the reset button of the Arduino whenever you're inserting or removing the card.

Format cards as FAT16 or FAT32. The SD card library works only with these two file formats. Fortunately, every operating system can format disks this way. Check your operating system's disk-utility application for how to format your card as FAT16 or FAT32.

Filenames have to be in 8.3 format. The SD library uses the old DOS file-naming convention that all filenames are a maximum of eight-characters long with a three-character exension. All file names are case-insensitive and spaces are not allowed. So, **datalog1.txt** and **mypage.htm** are OK, but **really long file name** or **someArbitraryWebPage.html** are not.

Writing to the card takes time. Normally, the file write(), print(), and println() operations saves data in a volatile buffer on the card, which is lost when the card is removed. Only the flush() or close() operations save to the card permanently, but they take time. To keep up with user interaction, use flush() and close() the way you would to save a file on a regular computer: do it frequently, but not too frequently, and preferably when the user or other devices aren't doing anything else.

Practice safe file management. The SD library gives you some helpful tools for managing your files. filename.exists() lets you check whether a file already exists. if (filename) lets you check whether the file can be accessed. filename.remove() lets you remove a file. size(), position(), seek(), and peek() let you see how big a file is, where you're at in the file, and to move around in the file. mkdir() and rmdir() let you make and remove whole directories. Use these methods, especially those that let

you look without changing, like exists(), size(), and peek(), which keep track of what you're doing when you write more complex file-handling sketches.

Indicate when you're accessing the card, and when you can't. Using LEDs to indicate when the Arduino is accessing the card and when there's an error is a handy way to know what's going on when you can't see the output on a screen.

For this project, you'll only read from the card from the microcontroller. To write to the card, you'll need an SD card reader/writer that can handle MicroSD cards. Readers are cheap, and they are becoming more and more ubiqitous. If your computer is a recent model, it may even have one built in. Most microSD cards come with an adapter to allow you to fit them into a regular SD card slot. Some cards even come with their own readers.

Once you've formatted your card, make a text file called **index.htm** and save it to the card. Then insert it into your shield. When the sketch runs, it should print the file out once at the end of setup(), then proceed to read the temperature sensor and control the relay as it did before. When that's working, you're ready to write the server. This code can be added to the sketch you've already got in progress.

Serve It

At the beginning of the sketch, add the Ethernet library and the necessary constants and global variables to use it and to set up a server. New lines, as usual, shown in blue.

```
/*
 GET/POST Web server with SD card read
 Context:  Arduino
*/
#include <EEPROM.h>
#include <SD.h>
#include <SPI.h>
#include <Ethernet.h>
#include <TextFinder.h>

// configuration for the Ethernet connection:
byte mac[] = {
  0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x01 };
IPAddress gateway(192,168,1,1);
IPAddress subnet(255,255,255,0);
IPAddress ip(192,168,1,20);
// Initialize the Ethernet server library:
EthernetServer server(80);

const int inputLength = 16;            // length of the file requested
const int typeLength = 6;              // length of GET or POST
const int sdChipSelect = 4;            // SD card chipSelect
const long tempCheckInterval = 10000;  // time between checks (in ms)
const int thermostatAddress = 10;      // EEPROM address for thermostat

char inputString[inputLength];         // for input from the browser
char requestTypeString[typeLength];    // what type of request: GET or POST
int nextChar = 0;                      // index counter for requestTypeString
const int fileStringLength = 16;       // length of the file requested
char fileString[fileStringLength];     // for input from the browser
long now;                              // last temperature check time
```

▸▸ Change these to match your own device and router.

▸▸ These will be used to manage the incoming HTTP requests.

In setup(), replace the call to sendFile() with this code to get an IP address and start the server. New lines are shown in blue.

```
void setup() {
  // initialize serial communication:
  Serial.begin(9600);
  // initialize the relay output:
  pinMode(relayPin, OUTPUT);

  //see if the SD card is there:
  Serial.print(F("Initializing SD card..."));
  if (!SD.begin(sdChipSelect)) {
    // if you can't read the SD card, print the error and go on:
    Serial.println(F("initialization failed!"));
  }
  else {
    Serial.println(F("initialization done."));
  }
  // give the Ethernet controller time to start:
  delay(1000);
  Serial.println(F("attempting to get address"));
  // Attempt to start via DHCP. If not, do it manually:
  if (!Ethernet.begin(mac)) {
    Ethernet.begin(mac, ip, gateway, subnet);
  }
  // print IP address and start the server:
  Serial.println(Ethernet.localIP());
  server.begin();
}
```

The loop() will change quite a bit. Starting at the beginning, add the following local variables. The final one listens for new clients to connect. If you get a client, make an instance of the TextFinder library to look for text in the incoming stream from the client.

```
void loop() {
  String fileName = "";  // filename the client requests
  char inChar = 0;       // incoming character from client
  int requestType = 0;   // what type of request (GET or POST);
  int requestedFileLength = 0;  // length of the filename they asked for

  // listen for incoming clients:
  EthernetClient client = server.available();

  if (client) {
    // make an instance of TextFinder to look for stuff from the client:
    TextFinder finder(client );
```

Remember what a client GET or POST request looks like. Here's the GET request:

```
GET /index.htm HTTP/1.0
```

And here's a POST request that provides a variable:

```
POST /response.htm HTTP/1.0
thermostat=23
```

Put this while() block inside the if statement that checks to see whether the client exists. It checks to see whether the client is connected, and whether it has sent any bytes. If so, it looks for the GET or POST requests, and gets the file requested as well. If it gets a POST, it extracts the thermostat variable. Then it calls sendFile() to send the file to the client.

```
while (client.connected()) {
    if (client.available()) {
      // look for whatever comes before the /. It should be GET or POST:
      if(finder.getString("","/", requestTypeString,typeLength)){
        // Do something different for GET or POST:
        if(String(requestTypeString) == "GET " ) {
          requestType = 1;
        }
        else if(String(requestTypeString) == "POST ") {
          requestType = 2;
        }

        // gather what comes after the / into an array,
        // it's the filename the client wants:
        requestedFileLength = finder.getString("", " ",
            fileString, fileStringLength);

        // now you're done with the GET/POST line, process what you got:
        switch (requestType) {
        case 1:    // GET
          // do nothing with GET except send the file, below
          break;
        case 2:    //POST
          // skip the rest of the header,
          // which ends with newline and carriage return:
          finder.find("\n\r");
          // if the client sends a value for thermostat, take it:
          if (finder.find("thermostat")) {
            int newThermostat = finder.getValue('=');
            // if it's changed, save it:
            if (thermostat != newThermostat) {
              thermostat = newThermostat;
             // constrain it to a range from 20 to 40 degrees:
              thermostat = constrain(thermostat, 20, 40);
              // save it to EEPROM:
              EEPROM.write(thermostatAddress, thermostat);
            }
          }
          break;
        }

        // whether it's GET or POST, give them the string they asked for.
        // if there's nothing after the /,
        // then the client wants the index:
        if (requestedFileLength < 2) {
          sendFile(client, "index.htm");
        }
        // otherwise send whatever file they asked for:
        else  {
          sendFile(client, fileString);
        }
      }
```

Wait, this sendFile() call looks different from the one on page page 375! The change is coming up.

» After the while() block, the thermostat check block from the original sketch closes out the loop() method.

Continued from previous page.

```
        // give the client time to receive the data:
        delay(1);
        // close the connection:
        Serial.println(F("Closing the connection"));
        client.stop();
      }    // close of the if (client.available() block
    }      // close of the while (client.connected() block
  }        // close of the if (client) block
  // NOTE: the thermostat check (the body of the loop() method on
  // page 374) goes here,
}
```

» The sendFile() method needs to change so you can send to the client instead of printing it out. It also needs to send the HTTP headers that come before the file. If the file isn't available, you should tell the client that, too. And you might as well use the standard HTTP error codes to do it. So now sendFile() sends an HTTP 200 OK header if it can read the file, and an HTTP 404 File Not Found header if it can't. Changes are shown in blue.

```
// send the file that was requested:
void sendFile(EthernetClient thisClient, char thisFile[]) {
  String outputString = "";      // a String to get each line of the file

  // open the file for reading:
  File myFile = SD.open(thisFile);
  if (myFile) {
    // send an OK header:
    sendHttpHeader(thisClient, 200);
    // read from the file until there's nothing else in it:
    while (myFile.available()) {
      // add the current char to the output string:
      char thisChar = myFile.read();
      outputString += thisChar;

      // when you get a newline, send out and clear outputString:
      if (thisChar == '\n') {
        thisClient.print(outputString);
        outputString = "";
      }
    }
    // if the file does not end with a newline, send the last line:
    if (outputString != "") {
      thisClient.print(outputString);
    }

    // close the file:
    myFile.close();
  }
  else {
    // if the file didn't open:
    sendHttpHeader(thisClient, 404);
  }
}
```

The sendHttpHeader() method looks like this.

```
// send an HTTP header to the client:
void sendHttpHeader(EthernetClient thisClient, int errorCode) {
  thisClient.print(F("HTTP/1.1 "));
  switch(errorCode) {
  case 200:      // OK
    thisClient.println(F("200 OK"));
    thisClient.println(F("Content-Type: text/html"));
    break;
  case 404:    // file not found
    thisClient.println(F("404 Not Found"));
    break;
  }
  // response header ends with an extra linefeed:
  thisClient.println();
}
```

If you run the code you've got so far, you should have a working web server. Whatever text files you put on the SD card should be accessible via the browser. You're not set up to serve anything other than HTML text documents (note that the sendHttpHeader() method only returns Content-Type: text/html), but you can do a lot with that. Try making a few pages that link to each other and put them on the SD card, then navigate through them with a browser. Try it on a mobile phone browser, too, since that's your original goal here. Resist the temptation to send the URL to all your friends just yet—there is more work to do.

Next, it's time to write the actual HTML interface pages. You need a page that can show the temperature and provide a form to enter a new thermostat setting. When the user submits the form, she should get feedback that her new setting has been received and stored, and the interface should reset itself. You also need a camera to get a picture of the room. Finally, it should look good on the small screen of a tablet or smartphone. All of this can be done in HTML.

Mark It Up

The main interface page will be called **index.htm**. You're using the three-letter extension rather than .html because the SD library only takes eight-letter, three-letter extension names.

The head of the document needs a little meta info. There's a link to a .css stylesheet—so you can set fonts and colors and such—and two meta tags allowing mobile browsers to format the page to suit their screens.

Finally, there's JavaScript that allows the image to refresh without changing the rest of the page.

```
<html>
   <head>
      <link rel="stylesheet" type="text/css" href="mystyle.css" />
      <meta name="HandheldFriendly" content="true" />
      <meta name="viewport"
         content="width=device-width, height=device-height" />

      <script type="text/javascript">
         function refresh() {
            var today=new Date();
            document.images["pic"].src=
   "http://visitor:password!@yourname.dyndns.com/image/jpeg.cgi"+"?"+today;

            if(document.images) window.onload=refresh;
            t=setTimeout('refresh()',500);
         }
      </script>
   </head>
```

The body of the document has an image tag that links to an image on another server; more details on that below.

The form in the body is for setting the thermostat level. It calls a separate document, **response.htm**, using a POST request, to which the server has to respond. You already wrote the initial response in your sketch above, when the server looks for the "thermostat" value at the end of the POST request. Check out case 2 in the switch statement in the sketch above. You can see that the temperature is constrained to a range from 20 degrees to 40 degrees, which is reasonable in Celsius.

Save this as **index.htm**, and then start a new document called **response.htm**.

Continued from previous page.

```
   <body onload="refresh()">
      <div class="header"><h1>Thermostat Control</h1></div>

      <div class="main">
         <img src="http://visitor:password!@yourname.dyndns.com/image/jpeg.
cgi" width="300" id="pic">
         <form name="tempSetting" action="response.htm" method="POST">
            <p>Current temperature:
            $temperature
            </p>
            Thermostat setting (&#176;C):
            <input name="thermostat" type="number"
               min="20"
               max="40"
               step="1"
               value=$thermostat>
            <input type="submit" value="submit">
         </form>
         Air conditioner is $status
      </div>
   </body>
</html>
```

The server will need to replace these variables before serving the file. You'll see how to do this later in this chapter.

Give a Response

The response page, **response.htm**, is a lot simpler than the index page. All it does is report the temperature, the thermostat setting, and the status of the air conditioner attached to the relay. It uses the same CSS, and the same mobile phone meta tags, as well as a meta http-equiv tag that sends the browser back to the index after three seconds. You may recognize this technique from the cat cam page in Chapter 3.

Did you notice the variables in the HTTP documents that look like PHP variables, starting with dollar signs? These need to be replaced with data from the server before the page is served. You don't have PHP on the Arduino, but you can write code into your sketch to look for these strings and replace them.

Before that, though, you need to finish by writing a CSS document for the site.

```
<html>
   <head>
      <meta http-equiv="refresh" content="3; URL=index.htm" />
      <link rel="stylesheet" type="text/css" href="mystyle.css" />
      <meta name="HandheldFriendly" content="true" />
      <meta name="viewport" content="width=device-width, height=device-
height" />
   </head>
   <body>
      <div class="header"><h1>Thermostat Control</h1></div>

      <div class="main">
         <p>Current temperature:
         $temperature
         &#176;C
         </p>
         <p>
         Thermostat setting changed to $thermostat&#176;C
         </p>
         <p>Air conditioner is $status
         </p>
         <a href="index.htm">Return to controls</a>
      </div>
   </body>
</html>
```

The server will need to replace these variables before serving the file.

Style It The final document is the stylesheet. Save this as **mystyle.css**, then copy this and the two preceding files to the SD card. Don't put them in a folder, but in the main directory (the root) of the SD card.

Feel free to change the colors and fonts for this document—they're purely cosmetic.

Once you have these files on the SD card, reinsert it into the Ethernet module, and then try to view them in a browser. You should get something like Figure 10-10. Your current version won't have values for the temperature, thermostat setting, and status, but read on.

```
.main {
   background-color: #fefefe;
   color: #0a1840;
   font-family: "Lucida Grande", Verdana, Arial, sans-serif;
}

.header {
   background-color: #0a173E;
   color: #f1fffe;
   font-family: sans-serif, "Lucida Grande", Verdana, Arial;
}
```

Making Your Server Public

Since your Ethernet module is running as a server, you need it to be publicly visible on the Internet if you plan to access it from outside your own network. If you can assign the module to a fixed address on your network, that helps, but if you're setting it up at home, that address is not likely to be visible outside your home network. You'll need to set up port forwarding on your router, as explained in the sidebar, "Making a Private IP Device Visible to the Internet," in Chapter 4. Using port forwarding, you can give devices that are attached to your home router a face on the public Internet. They will only appear as numeric addresses, because they don't have names assigned to them. However, that may be enough.

There are two solutions to giving your devices names you can remember through port forwarding. The simplest way is to embed the address of the module in a page on your public web page—you'll never need to remember the numeric address. For example, if your router's public address is 63.118.45.189, set up port forwarding so that port 80 of the router points to port 80 of the Ethernet module. Then restart your router. When you have a connection again, open a browser and go to http://63.118.45.189, and you should have access to the module. Put this link in a web page on a hosted web server with a real hostname, and you're all set.

The second solution is Dynamic DNS.

Dynamic DNS

If you definitely need a named URL for your module, you can use a Dynamic DNS (DDNS) host, such as www.dyndns.com. A DDNS host is simply a DNS host that continually updates the DNS record of your domain. Your router or device becomes a client, and when it connects to the DDNS host, it requests that the domain name you choose, such as yourserver.com, should point to the number from which the client is connecting. For example, your Ethernet module (local IP address 192.168.1.20) makes a DDNS request to dyndns.com. Because it goes through your router, which has IP address 63.118.45.189, dyndns.com links yourserver.com to 63.118.45.189. The numeric address is referred to as the CNAME, or Canonical Name record, of the name address. So when someone browses to yourserver.com, he's pointed to 63.118.45.189. And if you set up your router to have port forwarding turned on, pointing its public port 80 at 192.168.1.20 port 80, your Ethernet module appears to the public Internet as yourserver.com.

Dynamic DNS hosting does not have to be expensive. For a single domain that ends with a name the DDNS host owns, like fuzzipantsovich.dyndns.tv, it can be free. For custom names, or multiple names, it's usually a reasonable monthly fee that gives you a way to provide named hostnames for devices on your home network.

X

Network Cameras

Cameras that connect to the Internet have been available for several years now, and the prices on them, predictably, get cheaper and cheaper. For about $60, you can get a small camera that has a WiFi module onboard and that runs as its own server. The one used for this project, the D-Link DCS-930L, was purchased at an office-supply store for $70.

Setting up these cameras is very straightforward, and is explained in the documentation that comes with them. First, you need to connect to the camera through a wired Ethernet connection, and open its administrator page in a browser (just like you're building for the air conditioner—hey, how about that!). There, you configure the WiFi network you want to connect to, save it to the camera's memory, and restart the camera.

Since the camera is running a server, you need to know its address. You can use the same methods to make your camera public as you did for your Ethernet module. You may have to change the port that your camera serves images on, because you probably want port 80—the default HTTP port—to point to the Ethernet module. Port 8080 is usually a good second bet.

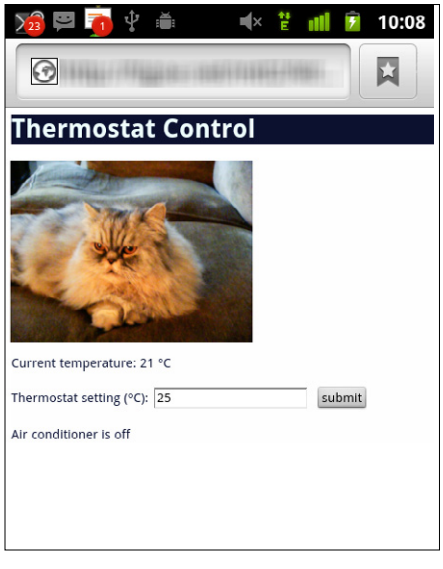
Once you've set up port forwarding for your camera, you can embed the address in a link in the Ethernet module's HTML. For example, if you're using the DCS-930L, the path to the image as a .jpg file is **/image/jpeg.cgi**. Let's imagine the public address of your home router is 63.118.45.189 (it isn't, so go look it up), and you set port forwarding on port 8080 to point to your camera's port 80. The link for the public page would then be <a href ="http://63.118.45.189:8080/image/jpeg.cgi">. However, the D-Link cameras, and many other brands, expect you to supply a password to access the camera. So, add a visitor user account that has view-only access to the image, and change the URL to <a href ="http://username :password@63.118.45.189:8080/image/jpeg.cgi"> (fill in the username and password that you set). This is highly insecure because anyone who views source on the page can see the password, so make sure the account has no privileges whatsoever to change or view settings!

Once you think you know the public URL of your camera's image, test it in a browser. It's a good idea to test away from your home network. When you're satisfied it works, embed the link in the **index.htm** page that you wrote previously, and try it out. You'll also need to make the change to the image's URL in the JavaScript in the document's head. Now your server is on the Internet for real!

There's one last stage of this project. You have to replace those placeholder variables for the temperature, thermostat setting, and air conditioner status. To do that, you need to add some more code to your sketch.

X

Figure 10-10
Screenshot of the final CatCam Thermostat control, taken on an Android phone.



Report It To change the variables in the HTML documents, the server needs to read those documents as it serves them. It does this inside the while() block in the sendFile() method. Add the following to that method. New lines are shown in blue.

When you run this, the server will read the file. Whenever it hits one of the named strings, $temperature, $thermostat, or $status, it will replace that string with the corresoponding variable's value.

Notice that the if() statement to change the temperature is different than the other two. There is no function to convert a float variable to a String, so you have to use print() or println() to do the job. That means you need to print the temperature variable directly to the client rather than just swapping out that part of the string before you print it.

```
while (myFile.available()) {
    // add the current char to the output string:
    char thisChar = myFile.read();
    outputString += thisChar;

    // check for temperature variable and replace
    // (floats can't be converted to Strings, so send it directly):
    if (outputString.endsWith("$temperature")) {
      outputString = "";
      // limit the result to 2 decimal places:
      thisClient.print(readSensor(),2);
    }

    // check for thermostat variable and replace:
    if (outputString.endsWith("$thermostat")) {
      outputString.replace("$thermostat", String(thermostat));
    }

    // check for relay status variable and replace:
    if (outputString.endsWith("$status")) {
      String relayStatus = "off";
      if (checkThermostat()) {
        relayStatus = "on";
      }
      outputString.replace("$status", relayStatus);
    }

    // when you get a newline, send out and clear outputString:
    if (thisChar == '\n') {
      thisClient.print(outputString);
      outputString = "";
    }
  }
```

Once you've made this change and uploaded it, you're done. When you go to the Arduino's address in your browser, you'll get the temperature, the current thermostat setting, and the status of the air conditioner. If you change any of the settings, you'll change the temperature in your home as a result. With the change you made to the image tag and the JavaScript in the **index.htm** document to accommodate your camera, you'll have an image from home as well. In a browser, your final application should look like Figure 10-10.

The great thing about using the browser as your application interface for mobile applications is that you only have to write the app once for all platforms. Many of the popular mobile applications are developed by making a basic browser shell without any user interface, then developing the user interface in HTML5. When you click through the app's interfaces, you're just browsing pages on their site. Because so many mobile phone applications are mainly network applications, it makes sense to use this as your primary approach. By doing so, you avail yourself of all the protocols the Web has to offer.

The web approach to mobile apps means you get to take advantage of protocols that don't run in a browser, but still rely on HTTP. You'll see this in practice in the next project. **X**

Project 30

Phoning the Thermostat

You put a lot of work into the last project. Fortunately, you get to reuse it in this project. You're going to keep the hardware exactly the same, but change the software in order to build an interface that lets you call the thermostat on the phone, hear the temperature and status by voice, and set the thermostat with your phone keypad.

MATERIALS

- **Completed Project 27**
- **Twilio account**

Luba is a bit of a luddite when it comes to new technologies, and she's not really fond of the mobile web interface for the thermostat. "But it's a phone!", she complains. "Couldn't I just call someone and have them stop by to change the temperature?" It's a fair point: if you've got a phone, you should be able to make things happen with a phone call.

IP-based telephony has taken leaps and bounds in the last several years, to the point where the line between a phone call and a web page is very blurry. Server applications such as Google Voice and Asterisk are like virtual switchboards—they connect the public switched telephone network (PSTN), and the Internet. These servers use a protocol called Session Initiation Protocol, or SIP, to establish a connection between two clients and determine what services they are capable of sending and receiving. For example, a SIP client might be able to handle voice communications, text messages, route messages to other clients, and so forth. Sometimes a SIP server sets up the connection between the clients, then gets out of the way and lets them communicate directly. Other times it manages the traffic between them, translating the protocols of one into something that the other can understand. It's the 21st-century version of the old switchboard operators. When application designers have done their job right, you never need to know anything about SIP, because your phone or software just tells you what it can do, and provides you a way to address other people. The phone does this by presenting you with a dial tone and a keypad.

If you've called an automated help service in the last few years, chances are you were talking to a SIP server. When you spoke, it tried to recognize your words using speech-to-text software, or it directed you to touch numbers on your keypad using text-to-speech software. When you entered numbers or words, it translated your input into HTTP GET or POST requests to query a remote server or local database. When it got results, it read those back to you using text-to-speech again. If it couldn't understand what you were saying (perhaps you were screaming "Operator! Operator! Give me a human being!" as I often do), it rerouted your call to a number where a human would answer.

For this project, you're going to use a commercial SIP service from Twilio to make a voice interface to the thermostat you just built. Twilio provides a variety of Voice over IP (VoIP) services like voicemail, conference calling, and more. With their commercial accounts, you can buy phone numbers to which you attach these services, so your customers can call your service directly. They also have a free trial service. With the free service, you must use the phone number they assign to you, and you have to use a passcode to access your application once you've called in. For readers outside the U.S. and Canada, Twilio offers only U.S./Canada-based numbers at the time of this writing. Read on to get the general idea, then find an equivalent service that works in your country. Phone number exchange is unfortunately one place the PSTN lags considerably behind the Internet, mostly for commercial and political reasons. If you really want to get deep into VoIP and SIP, check out **Asterisk: The Future of Telephony** by Jim Van Meggelen, Jared Smith, and Leif Madsen (O'Reilly).

X

What's the Standard?

Here's the bad news about SIP and VoIP applications: there's not a standard approach to them yet. They all offer slightly different services, though the basics of making calls, sending SMS messages, recording calls, and reading touchtones are available most everywhere. Each server and each commercial provider has a different approach to providing an application programming interface (API) for its service. The markup you learn here for Twilio won't apply when you're building an application using another service like Tropo, Google Voice, or any of the dozens of other providers out there.

There are four questions that can help you choose which tool to use for any project:

- Does it offer the features I need?
- Is it available in my area?
- Is it simple to use?
- Will it work with my existing tools?

For this application, I chose Twilio because its markup language, TwiML, is very simple. Also, Twilio's examples make it clear how to separate the markup language from another server-based language, like PHP or Ruby. Everything happens through GET or POST. Twilio has excellent PHP examples, but you don't need them to get started. Its debugger is useful, and its technical support is good as well. It lacks features that other services have—like the ability to get the audio level while recording, or speech-to-text conversion—but, on the whole, the benefits of its simplicity outweigh any of the missing features.

To complete this project, you'll need an account on www.twilio.com. You can build and run this project with a free account, or you can use a paid account. If you use the free account, you will have to enter a passcode in addition to the phone number every time you call, and you'll only be able to support one application at a time. For this introduction, that's enough.

You'll also need the URL of your Arduino server from the previous project. For example, if your server is at 63.118.45.189, the address for this project will be http://63.118.45.189/voice.xml. Log in to your Twilio account and go to the dashboard. There, you'll be given a sandbox phone number and passcode. The phone number will connect you to Twilio's gateway. The gateway will connect you to an HTTP server whose address you provide (give it the one above). Once you've done that, it's time to write an XML file for the server, and to modify the sketch so it will respond as needed. Figure 10-11 shows the dashboard panel.

A Brief Introduction to XML

XML, or eXtensible Markup Language, is a general markup language used by many web and database services. XML allows you to describe nearly anything in machine-readable form. XML is made up of tags that begin and end with < and >. Tags describe elements, which can be any concept you want to label. Elements can have subelements; for example, a <body> might have <paragraph> subelements, marked up like so:

```
<body>
   <p>This is the content of the paragraph</p>
</body>
```

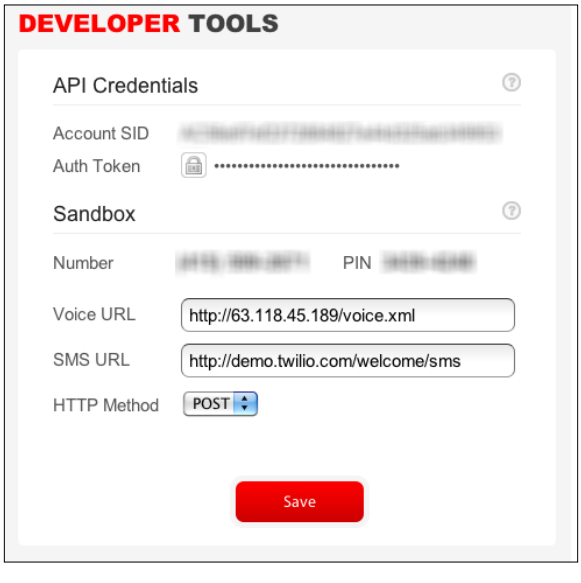


Figure 10-11
The Twilio dashboard. Enter the URL for your Arduino server in the Voice URL box.

The stuff between the tags is the content of an element. It's usually the stuff that humans want to read, but that the machines don't care about.

Every element should have an opening and closing tag, though sometimes a tag can close itself, like this:

```
<Pause length="10" />
```

What's inside a tag other than the tag's name are its attributes. The length above is an attribute of a Pause, and this specific pause's length is 10 seconds. Attributes allow you to describe elements in great detail.

If you're thinking all of this looks like HTML, you're right. XML and HTML are related markup languages, but XML's syntax is much stricter. Because it's such a general language, though, it's possible to write XML-parsing programs that work from one schema to another.

TwiML

Twilio's markup schema, TwiML, is a description of the functions Twilio offers written in XML. It describes what you can do with Twilio. There are elements for handling a voice call, and elements for handling an SMS message. The list of elements is pretty short:

Voice elements:
<Say>
<Play>
<Gather>
<Record>
<Sms>
<Dial>
<Number>
<Conference>
<Hangup>
<Redirect>
<Reject>
<Pause>

SMS elements:
<Sms>
<Redirect>

They're all explained in depth on Twilio's documentation pages, but you can probably guess what most of them do.

For this project, you're going to describe a <Response>. Within that, you'll use <Gather> to collect keypresses from the caller. The <Gather> element is TwiML's version of an HTML form, so it has an action attribute and a method attibute that you'll use to tell it where and how to send its results. You'll also use the <Say> element to speak to the caller using text-to-speech.

Mark It Up

There's only one TwiML document you need for this project. When a caller dials in, this document will be the initial response; when she enters keypresses, the document will call itself again to update the settings. Save it as **voice.xml** to the SD card from your previous project.

You can see the same variables from your previous project: $temperature, $thermostat, and $status. The server will replace them just like it does with the HTML documents. That part of your sketch won't need to change.

The changes that need to be made to the server sketch follow on the next page.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Gather action="voice.xml" method="POST">
        <Say>
            The current temperature is
            $temperature
            degrees Celsius.
            The thermostat is set to $thermostat degrees Celsius.
            The air conditioner is $status.
            If you would like to change the thermostat, please enter
            a new setting.
            If you are satisfied, please hang up.
        </Say>
    </Gather>
    <Say>
          You didn't give a new setting,
          so the thermostat will remain at $thermostat degrees. Goodbye!
    </Say>
</Response>
```

Modify the Server

There are only a few changes you need to make to the server to get it to be a voice server. The first is in the loop(). When Twilio makes a POST request, it sends back the digits that the caller pressed on the phone keypad using a variable called Digits. So where you were looking for a variable called thermostat in the POST request, you'll now look for Digits. The new line is shown in blue.

```
case 2:    //POST
           // skip the rest of the header,
           // which ends with newline and carriage return:
           finder.find("\n\r");
           // if the client sends a value for thermostat, take it:
           if (finder.find("Digits")) {
             int newThermostat = finder.getValue('=');
```

The Twilio gateway that acts as a client to your server is a bit more picky about what it expects than most browsers. It works best when you tell it in advance the length of the content you're going to send it. That means you need to make some changes to sendFile() and sendHttpHeader(). First, in sendFile(), you're going to get the size of the file you're sending, and identify whether the file you're sending is HTML or XML. Then you'll send the file size and type to the sendHttpHeader() method. The beginning of the sendFile() changes, as follows, are in blue.

```
// send the file that was requested:
void sendFile(EthernetClient thisClient, char thisFile[]) {
  String outputString = "";      // a String to get each line of the file

  // open the file for reading:
  File myFile = SD.open(thisFile);
  if (myFile) {
    // get the file size:
    int mySize = myFile.size();
    // determine whether the file is XML or HTML
    // based on the extension (but assume it's HTML):
    int fileType = 1;            // 1 = html, 2 = xml
    if (String(thisFile).endsWith("xml")) {
      fileType = 2;
    }

    // send an OK header:
    sendHttpHeader(thisClient, 200, mySize, fileType);
    // read from the file until there's nothing else in it:
    while (myFile.available()) {
```

You also need to change the call to sendHttpHeader() for a 404 error, which occurs later in sendFile(), as follows.

```
else {
   // if the file didn't open:
   sendHttpHeader(thisClient, 404, 0, 1);
}
```

NOTE: When developing this project, I benefited greatly by writing a test server—like the one described in Chapter 4—to simply read the whole request from Twilio before I wrote my final code. Twilio's online debugger also helped me find the problems.

The sendHttpHeader() method will now send a lot more information. Previously, you were sending the minimum amount needed to respond to most browsers, but the Twilio gateway client expects more. You've seen more detailed responses from the server—for example, in the air-quality project in Chapter 4. What you're sending here is similar. You're sending the name of the server application (Arduino), and then determining the content type based on what you learned in sendFile(). Then you're sending the content length, based on the file size. This tells the client how many bytes to expect before it closes the connection.

```
// send an HTTP header to the client:
void sendHttpHeader(EthernetClient thisClient, int errorCode,
                    int fileSize, int fileType) {
  thisClient.print(F("HTTP/1.1 "));
  switch(errorCode) {
  case 200:      // OK
    thisClient.println(F("200 OK"));
    break;
  case 404:    // file not found
    thisClient.println(F("404 Not Found"));
    break;
  }
  thisClient.println(F("Server: Arduino"));
  thisClient.print(F("Content-Type: text/"));
  if (fileType == 1) {
    thisClient.println(F("html"));
  }
  if (fileType == 2) {
    thisClient.println(F("xml"));
  }
  thisClient.print(F("Content-Length: "));
  thisClient.println(fileSize);
  // response header ends with an extra linefeed:
  thisClient.println();
}
```

When you've made these changes and put the **voice.xml** file on your SD card, restart the server and call your Twilio sandbox number. Enter the PIN, and the phone will ring. If you did everything right, you'll hear the text of the document read out to you, and you'll be able to enter a new thermostat setting using your phone keypad. You can keep changing the thermostat setting for as long as you like; when you're done, hang up.

Getting the Content Length Right

The server can theoretically serve both the XML and HTML documents now, so you should be able to see the pages in a browser as well. However, your browser may have problems with the content length. If you paid close attention, you may have realized that the size of the file is not actually the number of bytes you're sending as content.

The file size is determined before you replaced the $temperature, $thermostat, and $status variables with their actual values. For example, when the temperature is 27.28 degrees Celsius, you're replacing "$temperature" with "27.28"—that's 5 characters instead of 12. The same is true with $thermostat, which you're replacing with a two-digit number, and $status, which you're replacing with a two- or three-character string. To calculate the content length correctly, you have to account for this. There are two possible solutions.

The hack solution: Since most browsers don't care about the content length, don't print it if you're serving HTML. Put an if statement around the two lines that print the content length, like so:

```
if (fileType == 1) {
  thisClient.print(F("Content-Length: "));
  thisClient.println(fileSize);
}
```

The thorough solution: You know that the thermostat is a two-digit number because you constrained it. That's nine fewer characters than "$thermostat". You also know that the temperature is a five-character string because you limited it to two decimal places (and over 100°C is deadly). That's seven fewer than "$temperature". You know that the status string is two or three characters,

which is five or four fewer than "$status". So, calculate the difference before you send the HTTP header. The total is 20 or 21 characters shorter than the original file, depending on the relay status, which you can determine by calling checkThermostat(). You can change the sendFile() to adjust for this right before it sends the OK header, like this:

```
if (checkThermostat()) {
  mySize = mySize - 21;
}
else {
  mySize = mySize - 20;
}
// send an OK header:
sendHttpHeader(thisClient, 200, mySize, fileType);
```

When you're changing the contents of a file dynamically before you serve it, you have to make adjustments like this all the time. So, it's good to know a few methods for doing it. On a server with plenty of memory, you might simply put the whole file in an array and get the size of the array. One of the valuable things you learn when working on microcontrollers with limited memory is how to do workarounds like this.

I'll admit it: I went for the hack solution first.
X

HTML5 and Other Mobile Web Approaches

There's currently a lot of excitement among mobile application developers around HTML5, the next version of the HyperText Markup Language standard. HTML5 aims to make HTML able to support a greater amount of interaction and control over how web pages look and behave. In fact, what makes HTML5 interesting is not just the markup language itself, but the possibilities it offers when combined with JavaScript and CSS3, the Cascading Style Sheets standard.

So, how do HTML5, JavaScript, and CSS3 work together? Roughly, you could say that HTML gives you nouns, CSS gives you adjectives and adverbs, and JavaScript gives you the verbs to put them into action. HTML describes the basic page structure and the elements within it: forms, input elements, blocks of text, and so forth. CSS primarily describes the characteristics of the visual elements: the colors, fonts, spacing, etc. The scripting language JavaScript allows you to make connections between the elements of a page, between pages in a browser, and between the browser and remote servers. The grammar analogy isn't perfect, but the point is that the three tools give you a wide range of means to present information, listen to user input, generate interactive responses, and send and retrieve data to locations other than the browser—whether the data is on the user's hard drive or a remote server.

Previously, applications running within a web browser had very limited access to the hardware of the computer on which they were running. Operating system manufacturers felt that it was unsafe to allow an application that you downloaded from the Internet to access your hard drive, camera, microphone, or other computer hardware. Hackers might do evil things! Of course, now that nearly every application is downloaded from the Internet, that thinking seems dated. You could still download a program that does malicious things—but, by now, most people have a sense of what makes an online source more or less trustworthy, whether that source is delivering a web page or an application that's native to your operating system. At the same time, more of the data we need to access lives online today, whether on a social media site or a web data storage service like Google Docs or Dropbox. There's no need for a browser to access your hard drive if your files are online.

The blurring of the distinction between browser security and general security is good news if you like to build physical interfaces. HTML5 and JavaScript include methods to access some of the hardware of your

computer or phone. For example, there are methods now that let you determine the device's orientation by reading an accelerometer, if you have one built in (and most smartphones do); reading the compass and GPS receiver, if they're available; and more.

The bad news is that not all of these new methods are universally agreed upon among the companies who make browsers. Not every browser gives you access to all of these features, and not every browser implements them the same way. For example, the Safari browser on Apple's iPhone and iPad gives you access to the accelerometer as of this writing, but the standard browser on Android does not. So, while it's easy to access devices external to your phone through the browser, getting access to the ones on your phone may take a bit more work.

PhoneGap

If you're interested in getting access to the sensors on your phone, and you're comfortable working in JavaScript, PhoneGap (www.phonegap.com) is a very promising option. PhoneGap is a platform that gives you access to all the phone's hardware sensors through HTML5 and JavaScript. Basically, PhoneGap embedded the phone's built-in browser engine (which implements all of HTML5's new standards), added a bunch of hooks into useful functionality (such as built-in sensors) and released it as a basic frame for you to develop in. You download the application framework, which is written in your phone's preferred programming language (Java on Android, Objective-C on iOS). You don't need to do anything to that code—it's just there to act as a shell for your app. You write your own HTML5 and JavaScript documents, which form the core of your app. Then you compile it all and upload it to your phone. PhoneGap also offers an online compilation service: upload your HTML files and it compiles an application for you to download to your phone.

There are things you don't get with PhoneGap. For example, you can't open a connection to your phone's Bluetooth serial port, if it has one, nor can you access the phone's USB connection. Despite that, it's a promising start.

PhoneGap is not the only platform for developing mobile phone apps for multiple operating systems. MoSync (www.mosync.com) also offers application frameworks for multiple platforms, but you develop your own application in C++, not in HTML. You get access to more of the hardware with MoSync, but it definitely requires a greater familiarity with programming than anything you've seen in this book so far. Several other companies are offering cross-platform tools like this now, and more will certainly come along. If you're interested in developing applications for the phone that use built-in sensors, and you don't want to write an application native to each different phone operating system, PhoneGap is your best option until the browsers catch up.

X

Text-Messaging Interfaces

Text messaging is fast becoming the most common use for mobile phones. Many people regard texts as less intrusive than voice calls in the flow of daily life. With a text message, you can get information across quickly and with no introduction. In addition, it's easy to send email to SMS, and vice versa; they're not platform-dependent—as long as you have a mobile phone or email account. For situations where you need an immediate, unobtrusive notification, or to give a single instruction, they work wonderfully.

SMS, or Short Messaging Service, began as a way of sending information over the mobile phone networks' signaling channel. The idea was to send bytes as part of the data sent to signal incoming calls, but to do so when there was no incoming call. These short messages could be used for diagnostic purposes, to notify the receiver of voicemail, or for other quick notifications. When SMS was rolled out as a commercial service option for customers, however, it became much more than that. SMS may be limited to 160 characters per message, but people have found many creative ways to pack a lot of info into those 160 characters.

Almost all mobile carriers provide an SMS-to-email gateway as part of their service, which means you can send an SMS from an email client and receive SMS messages in your inbox. To test this out, send a text message from your phone, but instead of sending it to a phone number, enter your email address as the destination. Depending on your carrier, it may be sent via MMS, or Multimedia Message Service, or it may simply go as an SMS. Check your inbox, and you'll see a message from yourself. Now you'll know the email address to use if you want to send yourself a text message via email as well. Most of the time it's simply your phone number @ your carrier's email address. Here are some common SMS-to-email servers for a few U.S., Canada, and European carriers:

AT&T: phonenumber@txt.att.net
T-Mobile: phonenumber@tmomail.net
Virgin Mobile: phonenumber@vmobl.com
Sprint: phonenumber@messaging.sprintpcs.com
Verizon: phonenumber@vtext.com
Bell Canada: phonenumber@txt.bellmobility.ca
Telenor Norway: phonenumber@mobilpost.no
Telia Denmark: phonenumber@gsm1800.telia.dk
Swisscom: phonenumber@bluewin.ch
T-Mobile Austria: phonenumber@sms.t-mobile.at
T-Mobile Germany: phonenumber@t-d1-sms.de
T-Mobile UK: phonenumber@t-mobile.uk.net

A longer list can be found at www.emailtextmessages.com (warning: I have not verified every one of these). With U.S. carriers, many expect a simple 10-digit phone number without the leading country code (which is +1 for the U.S.), but carriers in the U.S. and around the world tend to be idiosyncratic about this. Check with your carrier's customer support to find out how they handle it.

The PHP script on the next page creates a simple form that lets you send a text message to any of the carriers listed above, and a few others. Modified, it can be used as an SMS gateway script for a networked device.

Try It To send an SMS, just send mail to the 10-digit phone number at the recipient's carrier. Here's a PHP script to send yourself an SMS. Save this as **sms.php**.

> **Consider password-protecting this script, or removing it after you're done testing. It could become the target of abuse if it's found by a roving spambot with a pocket full of phone numbers.**

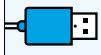
```
<?php
/*
    SMS messenger
    Context:  PHP
*/
   $phoneNumber = $_REQUEST["phoneNumber"];    // get the phone number
   $carrier   = $_REQUEST["carrier"];          // get the carrier
   $message   = $_REQUEST["message"];          // get the message
   $recipient = $phoneNumber."@".$carrier;     // compose the recipient
   $subject = "Message for you";

    // if all the fields are filled in, send a message:
    if (isset($phoneNumber)&& isset($carrier) && isset($message)) {
       mail($recipient, $subject, $message);
    }
?>

<html>
<head></head>
<body>
   <h2>SMS Messenger</h2>
   <form name="txter" action="sms.php" method="post">

      Phone number: <input type="text" name="phoneNumber"
      size="15" maxlength="15"><br>
      Message:<br>
      <textarea name="message" rows="5" cols="30" maxlength="140">
Put your sms message here (140 characters max.)
      </textarea>
      </br>
      Carrier:
      <select name="carrier">
         <option value="txt.att.net">AT&T US</option>
         <option value="txt.bellmobility.ca">Bell Canada</option>
         <option value="messaging.nextel.com">Nextel US</option>
         <option value="messaging.sprintpcs.com US">Sprint</option>
         <option value="bluewin.ch">Swisscom</option>
         <option value="sms.t-mobile.at">T-Mobile Austria</option>
         <option value="t-d1-sms.de">T-Mobile Germany</option>
         <option value="t-mobile.uk.net">T-Mobile UK</option>
         <option value="tmomail.net">T-Mobile US</option>
         <option value="gsm1800.telia.dk">Telia Denmark</option>
         <option value="mobilpost.no">Telenor Norway</option>
         <option value="vtext.com">Verizon</option>
         <option value="vmobl.com">Virgin Mobile US</option>
      </select>

      <input type="submit" value="send message">
   </form>
</body>
</html>
```



GPRS for Microcontrollers

There are a handful of devices on the market that allow you to connect a microcontroller to mobile phone networks directly. Using one of these, your microcontroller connects to the Internet the same way that your mobile phone does. It has a phone number, and it can send and receive SMS messages, make HTTP calls, and do anything else you can do on the Internet. The tradeoffs to using them are that they can get expensive—in terms of both power usage and connectivity costs.

Telit's modules are the high end of the GPRS module market, and there are a few evaluation boards and shields for Arduino based on these. They feature a TTL serial interface, an AT-style command set (meaning they operate as a modem, like the Bluetooth Mates, but with different commands). They have their own TCP/IP libraries on board, so you can make network connections, and some models feature GPS as well. One model even has a Python interpreter so you can run programs written in the Python programming language on it.

Spark Fun sells breakout boards for the Telit GE865 and GM862, as well as one for ADH Tech's ADH8066 module, These breakout boards give you all the pins of the module on 0.1-inch spacings, and usually a USB-to-Serial connection. They're not designed to be used with any particular microcontroller, so you'll have to start from the command set and the manufacturer's datasheet if you want to use these.

They also sell a GPRS shield for Arduino based on Spreadtrum Technologies' SM5100B module. This module isn't as feature-laden as the Telit modules, but it has an AT command set, and it can send and receive SMS messages and make network connections. John Boxall has a nice set of tutorials for using this shield at tronixstuff.wordpress.com/2011/01/19/tutorial-arduino-and-gsm-cellular-part-one.

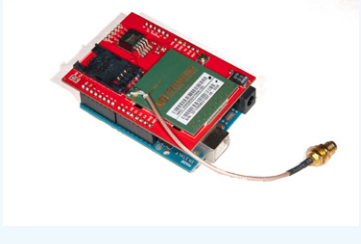
Libelium (www.libelium.com) sells a number of different GPRS shields for Arduino through their Cooking Hacks site (www.cooking-hacks.com). They have dual-band and quad-band modules, meaning that some of their modules can operate in the U.S., Europe, Africa, and much of Asia, depending on the country and carrier.

Seeed Studio carries a quad-band GPRS shield as well, with built-in audio jacks for voice communications in case you want to build your own phone.

The two challenges to using any of these GPRS shields are power and price. The SM5100B shield, for example, can draw up to 2 amps when it's making a call. That's more than the Arduino's regulator can feed it, so the shield connects to the board's Vin pin. That means you need to supply the board with at least 2 amps just for the GPRS module. The board won't draw all that current the whole time, but if it can't get it when it needs it, you won't make the connection.

For any of these modules, you'll need a mobile subscription and working SIM card from your favorite mobile carrier. Unless you have a flat-rate data and text plan, you can spend a lot of money testing and debugging GPRS projects. A second alternative is a pay-as-you-go plan for your SIM card. Neither is an ideal choice, unfortunately, so do as much troubleshooting as you can offline to save money where possible.





Two ways to get GPRS to a microcontroller: the Telit GM862 evaluation board breaks out all the pins of the module to 0.1" spacing; the SM5100B shield breaks out the serial pins of its module to the Arduino's serial pins.

Native Applications for Mobile Phones

Even though web and SMS interfaces offer many possibilities, there are projects for which you really need access to a mobile phone's operating system. Web applications make it difficult—if not impossible—to access the phone's hardware or file system, for example. In these cases, you need to get to know your phone's operating system and the programming tools available for it. If you're aiming to make an application for all mobile phones, you need to become familiar with several operating systems.

The mobile phone industry is fast changing, as are its smartphone operating systems, and keeping up with it all can be exhausting. Google's Android OS, Blackberry, Palm's webOS, Windows Mobile, Symbian, and Apple's iOS (for iPad, iPhone, and iPod touch) are currently the largest players. Of those, Android, iOS, and Blackberry are the three largest at the moment, covering most of the market. Nokia's Symbian was one of the largest, but Nokia is no longer supporting it as its primary operating system. As of the last quarter of 2010, Android was the most popular of smartphone operating systems, taking 35% of the world market.

There's a lot of interest in programming for iOS because it's such an attractive environment, but Apple is notoriously controlling about iOS development. You have to register as a developer, and they've limited the available toolkits severely. In addition, you can't just distribute your app independently—you have to do it through the App Store. That's fine for commercial developers, but for hobbyists and home hackers, it can be quite forbidding. For more on iPhone programming, see Alasdair Allan's books, *Learning iPhone Programming* (O'Reilly) and *iOS Sensor Programming* (O'Reilly). Alasdair has done a great deal to make iPhone programming accessible.

Android, on the other hand, is somewhat more accessible to programmers. Applications can be installed on Android phones using the Android Market, or they can be installed over USB. Based on Linux and programmed in Java, it offers an environment familiar to many experienced developers. For those less comfortable with code, Google offers a graphic programming environment called App Inventor (http://appinventor.googlelabs.com) that allows you to create applications by combining graphic objects. For the rest of this chapter, you'll learn to make Android apps using a familiar environment: Processing. As of version 1.5, Processing can compile and install sketches as Android apps.

Processing for Android

Processing for Android is a very exciting update to Processing, but as it is very new, it is still under development. The examples that follow were developed at the same time as the tools to make them possible—in some cases, they are the first real tests of the libraries they use. So be warned, you're sailing in uncharted waters now. As the Processing for Android wiki (http://wiki.processing.org/w/Android) explains: "Do not use this code while operating heavy equipment. Do not rely on this code for thesis or diploma work, as you will not graduate. Do not use this code if you're prone to whining about incomplete software that you download for free." The libraries used here are likely to change after the publication of this book, and they will hopefully become more stable and easier to use. So make sure to check the online documentation of anything mentioned here for updates.

Setting Up Processing for Android

Processing for Android has a slightly different workflow than standard Processing, so you'll need to install some new components and get used to some new tools. The first thing you'll need is Processing version 1.5.1 or later, which can be downloaded from the Processing site (www.processing.org/download). You'll also need the Android Software Developers' Kit (SDK), available at http://developer.android.com/sdk. Download and install both. Then open Processing. You'll see a new button in the main toolbar labeled Standard, as shown in Figure 10-12. Click it and choose Android. You'll get a dialog box asking you if the Android SDK is installed. Click Yes. Another dialog box will pop up asking you where it is. Navigate to the Android ADK folder that you just installed. The Processing editor color scheme will turn green, and you'll be in Android mode.

You now need to install some components of the Android SDK, so go to the new Android menu and choose Android SDK & AVD Manager. This will open the Android SDK Manager, where you can install new Android packages as new versions become available. Click Available Packages to install new packages. At minimum, you need to install three things.

Underneath Android Repository, check the boxes for "Android SDK Platform-tools" and "SDK Platform Android 2.1, API 7".

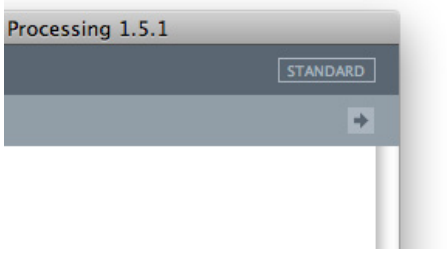


Figure 10-12
Processing's Mode button set to Standard mode.

Beneath "Third party Add-ons", expand the Google Inc. entry, and select "Google APIs by Google Inc., Android API 7".

You can install more recent packages if you want, depending on what version of Android your device is running, and what features you want to access (but you need at least these for Processing for Android to work).

To check your device's operating system version, go to the home sceeen, and from the menu, choose Settings. Scroll to the bottom and choose "About phone". In that menu, you'll see a listing of the Android version. Figure 10-13 shows the menu. If you have a later version than 2.1, you might want to install the latest that your phone can run. For example, I installed the SDK Platform Android 2.3.3, API 10, revision 1 to work with a Nexus S phone that was running version 2.3.4, and the Google APIs Android API 10, revision 2, as well.



Figure 10-13
The "About phone" menu, showing your Android version.

Where Does the App Run?

You have two choices as to where your app will run when you click the Run button: it can run in an emulator on your desktop, or it can run on your actual phone. The emulator is useful if you don't have an Android phone handy. The SDK Manager should install an emulator for you automatically, but if not, you can install one by clicking Virtual Devices in the SDK Manager. Emulators tend to be slow, though, and they're nowhere near as exciting as running the app on your actual phone.

To enable your Processing sketches to run directly on your Android device, first go to the device's Settings menu and choose Applications. From there, choose Development. Finally, click USB Debugging. This enables your device to install applications over USB, and to send debugging information back to Processing or any other development environment over USB as well.

Once you've enabled USB debugging, plug your phone into your computer with a USB cable, and open one of the Processing for Android example sketches (remember, examples are in the File menu, under Examples). The Accelerometer example in the **Sensors** folder is fun. While holding down the Shift key, click Run. This should change the title in the toolbar from "Run in Emulator" to "Run on Device" (see Figure 10-14). The sketch will compile, then install on your device and start running. Ta Dah! You're now an Android developer!

The handy thing about "Run on Device" is that you can get messages back from the sketch while it's running on your device. This is because you enabled USB debugging. Any print() or println() commands in your sketch will print to the debugger pane of Processing on your desktop just like always. This makes debugging your sketch so much easier. Make sure to remove your print() and println() statements when your sketch is done, though. When the device is not connected to your computer, those commands write to a log file on the device, and they will start to fill up its available storage space.

Differences to Watch Out For

There are a few big differences between Standard-mode Processing and Processing for Android.

Many libraries for Standard mode do not work in Android mode. This will likely change as library developers adapt the useful ones to Android mode, but for the moment, it's best not to assume.

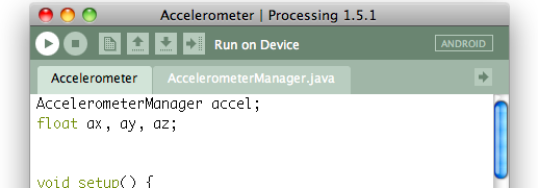


Figure 10-14
To run a sketch directly on your phone, hold down Shift while clicking the Run button.

An Android mode sketch is an activity in Android Java terms. Activities have a few core methods: onCreate(), which is equivalent to Processing for Android's setup() method; and onPause() and onResume(), which are called by Processing's pause() and resume() methods. Activities (and therefore sketches) pause and resume frequently: whenever your device goes into idle mode, and every time you rotate the orientation, and every time you switch to another application. So, you should put in pause() and resume() methods to manage those transitions smoothly.

Interaction is different with a touchscreen. There's no mouse button, so the mouse button events don't work. mouseX and mouseY still work, and you've also got new variables—motionX, motionY, pMotionY, and motionPressure—that you can use to tell something about touch as well. mousePressed also still works.

Since an Android sketch takes over the whole screen, there's no point in setting size(), so it doesn't work in Android mode. You can get the screenWidth and screenHeight if you want them, though. You can also lock the orientation, like so:

```
orientation(PORTRAIT);
orientation(LANDSCAPE);
```

Fonts still work as before. To get a list of the system fonts, use this line of code:

```
println( PFont.list() );
```

You need to enable different permissions to get at different functions of the device. In the Android menu, you'll see a list of Sketch Permissions. You can set most of them from this menu. This menu writes to the **AndroidManifest.xml** file in the sketch directory.

Whenever possible, use the permissions menu rather than editing the manifest. For everything you need in this book, you can use the permissions menu and be safe. It's easy to mess things up by editing it wrong—remember, XML is not forgiving.

Since your sketch restarts every time you change orientation, you might want to use saveStrings() and loadStrings() to save and load variables that need to persist.

For more tips on the differences, see http://wiki.processing.org/w/Android for the latest information.

Now here's a quick sketch to get you started. It reads the mouse position and saves data when you pause and resume. You'll need to set the Sketch Permissions to enable WRITE_EXTERNAL_STORAGE.

X

Touch It This sketch will show you how mouseX, mouseY, and motionPressure work on Android. It will reorient itself, and save and reload data when you rotate the device.

```
/*
  Processing for Android test
 Context:  Processing
 */
float ballX, ballY;        // position of the ball
// file to save data for pause and resume:
String datafile = "sketchFile.dat";

void setup() {
  // create a font for the screen:
  String[]fontList = PFont.list();
  PFont androidFont = createFont(fontList[0], 24, true);
  textFont(androidFont, 24);
}

void draw() {
  // color theme: Sandy stone beach ocean diver by ps
  // http://kuler.adobe.com:
  background(#002F2F);
  fill(#EFECCA);
  // show the mouse X and Y and finger pressure:
  text("mouseX:" + mouseX, 10, 50);
  text("mouseY:" + mouseY, 10, 80);
  text("motionPressure:" + motionPressure, 10, 170);
  // move the ball if the person is pressing:
  if (mousePressed) {
    ballX = mouseX;
    ballY = mouseY;
  }
  // draw a nice blue ball where you touch:
  fill(#046380);
  ellipse(ballX, ballY, 50, 50);
}
```

The pause() and resume() methods use an external file saved to your device to save variables that persist when the sketch restarts. Each variable is a separate string, saved on its own line. A newline character (\n) separates each line. Figure 10-14 shows the results.

```
void pause() {
  // make a string of the ball position:
  String ballPos = ballX+ "\n" + ballY;
  /// put the string in an array and save to a file:
  String[] data = {
    ballPos
  };
  saveStrings(datafile, data);
}

void resume() {
  //load the data file:
  String[] data = loadStrings(datafile);
  // if there's a file there:
  if (data != null) {
    // and there are two strings, get them for X and Y:
    if (data.length > 1) {
      ballX = float(data[0]);
      ballY = float(data[1]);
    }
  }
}
```

Play around with some of the Android examples that come with Processing as well. The Accelerometer and Compass sketches are good fun, and they show you how to get at those useful components on your phone. Once you're familiar with Processing for Android, you're ready to make your own datalogger on the phone.

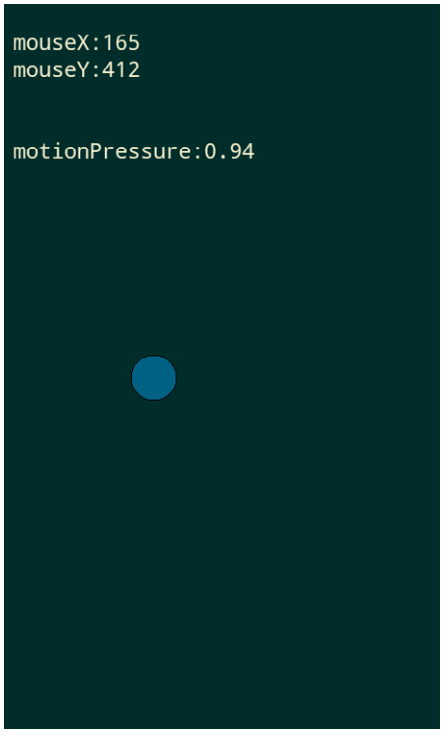


Figure 10-15
The Processing for Android sketch that you just wrote.

Project 31

Personal Mobile Datalogger

One popular reason to develop native applications on a mobile phone is to use the phone's Bluetooth radio as a serial connection to other devices. In this way, your phone can become a mobile datalogger, or a conduit to send the data to a database on the Internet. In this project, you'll sense your Galvanic skin response using an Arduino, send the data via Bluetooth to your Android phone, and log the result to a file on the Internet.

MATERIALS

- **Android device**
- **LilyPad Arduino Simple**
- **Bluetooth Mate**
- **Lithium Polymer Ion Battery**
- **1 resistor, 270-kilohm**
- **Conductive ribbon**
- **Conductive thread**
- **Shieldit Super 14" fabric**
- **Velcro**
- **Hoodie**
- **Embroidery thread**

A growing number of personal data enthusiasts are gathering personal biometric data for many different purposes, from visualizing their activity patterns in order to improve exercise habits, to tracking sleep patterns in order to find solutions to insomnia. Quantified Self meetups (http://quantifiedself.com) are popping up around the world for people to share tips and tricks on how to do this, and devices like the FitBit (www.fitbit.com) and the Zeo (www.myzeo.com) have come on the market to make biometric tracking easier.

This project is based on the work of ITP alumnus Mustafa Bağdatlı, shown in Figure 10-15. Mustafa wanted to track his Galvanic skin response (GSR) and heart rate against his calendar, so he could see when his mood—as reflected in his heart rate and GSR—were affected by the events of the day. His project, Poker Face, tracked the two biometric characteristics on a LilyPad Arduino, transmitted them via Bluetooth to a mobile phone, and logged the result on the Web. You can find more on Poker Face at http://mustafabagdatli.com In this project, you'll build the same, but without the heart rate sensor to keep it simple.

Feel free to change the sensor for whatever you want to track yourself.

Figure 10-17 shows the system for this project. A microcontroller reads the analog voltage from the sensor and sends it serially to a Bluetooth Mate. The data is then transmitted over Bluetooth to a mobile phone using the Serial Port Profile, or SPP, that you used in other Bluetooth projects in this book. The phone then makes an HTTP GET request to a PHP script on a web server. The script saves the incoming data to a file. What you do with the data from there is up to you.

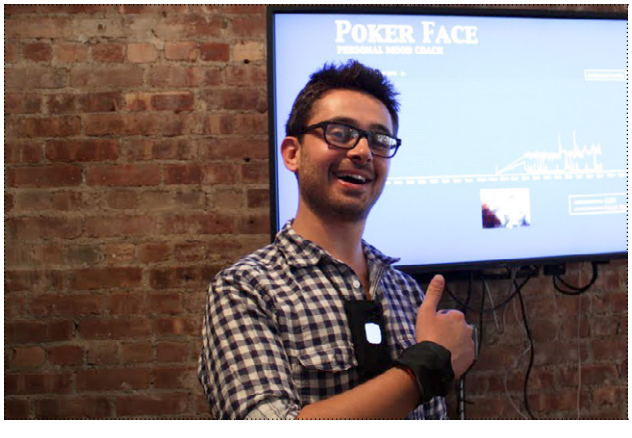


Figure 10-16
Mustafa Bağdatlı wearing Poker Face, a biometric datalogger linked to a mobile phone. *Photo courtesy of Mustafa Bağdatlı.*

The Circuit

The circuit for this project is quite simple. To measure Galvanic skin response, all you need is a high-value resistor and your skin. As a test, take a multimeter and measure the resistance across your wrist. You'll find that the resistance is high, probably in the megohm range if your skin is cool and dry. Work up a sweat and measure it again. You'll see that the resistance has gone down. When you exercise, or when you're faced with stressful or arousing situations (good or bad), you perspire more, changing the conductance of your skin. That's what this project will measure.

The sensor is basically a voltage divider, with your skin as the variable resistor and a fixed resistor completing the circuit. A 270-kilohm resistor is used here (see Figure 10-19 and Figure 10-20), but feel free to change it to suit your skin. Generally, below 10 kilohm won't work.

Conductive fabric and conductive thread were used to make the sensor for this circuit, and to attach the LilyPad Arduino to the garment. Different conductive fabrics and threads have different material and electrical properties, so you might want to experiment with a few to find what works for you. Mustafa used stretchable conductive fabric sewn into wristbands for his cuffs, as you can see in Figure 10-18. For this project, I used Shieldit Super fabric with adhesive backing. For more information about conductive fabrics, see Leah Buechley's excellent introduction at http://web.media.mit.edu/~leah/grad_work/diy/diy.html; Syuzi Pakhchyan's book *Fashioning Technology: A DIY Intro to Smart Crafting* (O'Reilly); or Hannah Perner-Wilson's excellent online resource at web.media.mit.edu/~plusea.

Test the whole circuit with the code below before sewing it into the garment. Hardware debugging after you sew it is difficult.

The Construction

For this project, you'll use iron-on conductive fabric attached inside the pocket of a hoodie as your sensor contacts. When you reach in your pocket, you can touch the fabric contacts with the palm of your hand, and take a reading on your phone with the other hand. Or, you can let the mobile app run continually to take a reading every two minutes.

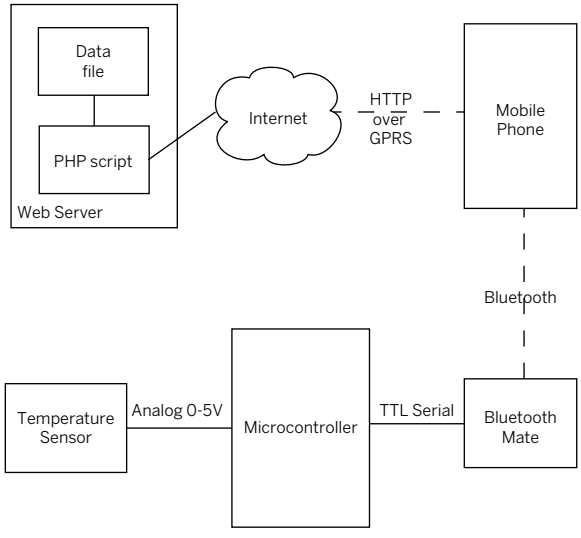
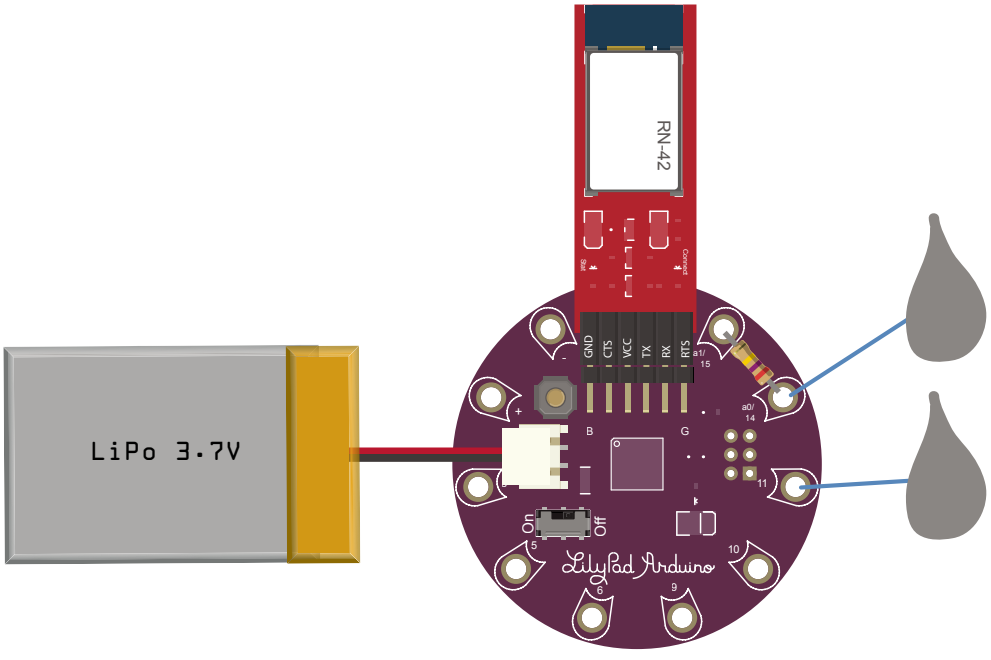


Figure 10-17
The system diagram for the mobile datalogger project.

Figure 10-18
Detail photo of the Poker Face GSR wristband. Here, the wristband is inside-out to show the conductive fabric contacts. *Photo courtesy of Mustafa Bağdatlı.*





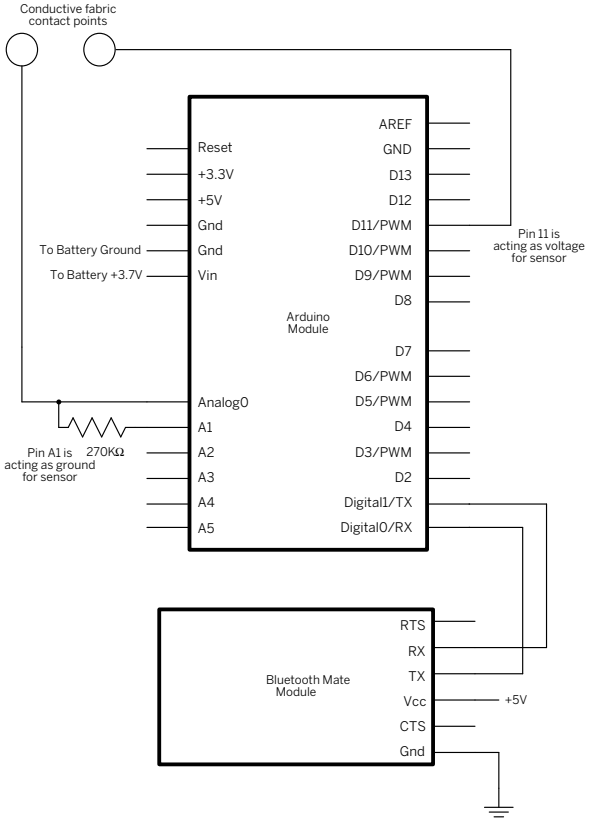


Figure 10-19
The mobile datalogger circuit. The contacts for the sensor are conductive fabric.

Figure 10-20
The mobile datalogger circuit schematic.

NOTE: When you're using Lithium Polymer batteries in your project, get a USB charger for them. Adafruit's USB Lilon/LiPoly charger (ID: 259) or Spark Fun's LiPo Charger Basic - Micro-USB (sku: PRT-10217) work well.

Conductive thread connects the fabric contacts on one side of the garment to a three-wire conductive ribbon on the other side of the garment. The microcontroller and battery will be sewn into the bottom band of the hoodie just below the pocket, and the conductive ribbon will connect them, sewn onto the inside lining of the hoodie. Figure 10-21 shows the layout on the inside of the garment.

Iron the conductive fabric contacts into the pocket. Space them so they cover about the width of the heel of your palm, and contact it comfortably when you put your hand in the pocket. Figure 10-22 shows you the inside of the pocket.

Split the ends of the three-wire conductive ribbon about an inch on either end with a pair of scissors. You need enough distance between the wires to span the space between the conductive fabric contacts on one end, and between two contacts of the LilyPad Arduino Simple on the other. Tin the tips of the outside wire ends to keep them from fraying. You're not using the middle wire, so cut it out, down to the end of each split. You can also embroider the two outside conductors with yarn that matches the hoodie to keep them from fraying, if you wish. Solder one pair of the conductive ribbon's leads to pins 11 and A0 of the LilyPad Arduino.

Cut a hole in one side of the bottom band just below the pocket to insert the microcontroller with the Bluetooth Mate and battery attached. You'll need to be able to get the battery in and out for charging, so finish the edges of the hole so they won't fray with repeated use. Iron-on fabric, or embroidery thread will do the job. Then add Velcro to the inside edges to provide a closure.

Cut another small slit in the band about six inches (10cm) away from the large hole so you can insert the ribbon through the bottom band. Figure 10-21 shows the two cuts.

Make sure the LilyPad Arduino Simple is programmed and the circuit is fully assembled before you sew it into the garment. The sketch is below.

Position the ends of the ribbon on the inner lining of the garment opposite the fabric contacts in the pocket. Sew the conductive ribbon to the contacts through the material with conductive thread. Use tight stitches for a good connection. Use a meter to check connectivity between the contacts and the opposite end of the ribbon when you've finished your sewing. This way, you know the signal makes it from the contacts through the garment to the end of the ribbon that will attach to the LilyPad.

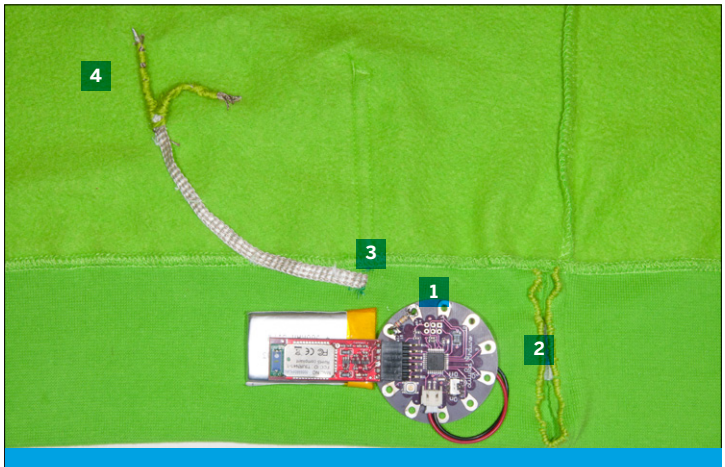


Figure 10-21. The layout of the hoodie components on the inner lining: 1. The LilyPad Arduino and the battery are actually inside the bottom band of the garment, inserted through 2. the large cut on the right. The conductive ribbon comes out of the bottom band through 3. the small cut, and goes to 4. the pocket contacts at the top (opposite side of the garment).

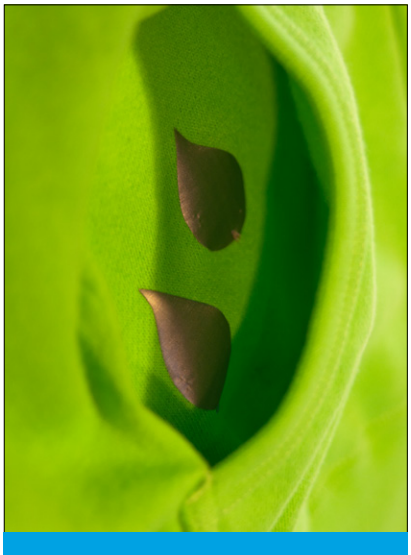


Figure 10-22. The conductive fabric contacts inside the hoodie pocket.

Tack the ribbon down to the garment's inner lining about every inch, so it's secure and won't fold over itself when worn. Don't sew the battery in—you'll need to remove it for recharging. You shouldn't need to sew the other components in either, but Velcro on the back of the LilyPad Arduino and in the lining of the band will secure it nicely.

At this point, the circuit should be fully functional. Test it with a wired connection before going Bluetooth. To test it, attach a USB-to-Serial connector to the LilyPad Arduino (which will take some creative cabling) and open the serial connection in the Serial Monitor or another serial terminal program at 115200bps. Put your hand on the contacts in the pocket. Then send any byte, and you should get back a sensor value. To see a change in the value, either work up a sweat or lick your hand and put it on the contacts again. When you know it works, remove the USB-to-Serial line, and connect the Bluetooth Mate and the battery and turn it on. Now connect from the Serial Monitor or a serial terminal program via the Mate's serial port. This will work like it did in Chapter 2 when you added the Bluetooth Mate to Monski Pong.

The Code

There are three pieces of code for this project: the PHP script, the Arduino sketch, and the Processing sketch. The first two are relatively simple, and having them done and tested makes the Processing sketch easier to understand.

The Arduino sketch listens for serial input and when a byte arrives, it takes a reading, maps it to a voltage range, and sends it out.

The PHP script accepts a request string and looks for one variable, called data. It appends everything from that variable into an existing text file called **datalog.txt**. It returns a basic HTML page with the data that the client sent.

The Processing sketch checks if it's connected to the Arduino's Bluetooth Mate. If it's connected, it asks for a reading once every 10 seconds. Once every two minutes, it sends the accumulated readings to the PHP script. There are also two buttons: one to get a reading and another to send the reading to the server.

X

Read It The Arduino sketch's global constants include the two pins to be used as voltage and ground for the sensor.

setup() initializes serial at 115200bps (the default rate for the Bluetooth Mate) and sets the voltage and ground pins for the sensor appropriately.

loop() checks to see whether there's been any serial input. When there is, it reads the byte just to clear the serial buffer, then takes a sensor reading, maps it, and returns it.

Notice that the mapping is to a range from 0 to 3.7V. That's because the Lithium Polymer battery supplies the microcontroller at 3.7V when fully charged.

```
/*
 Galvanic Skin Response reader
 Context:  Arduino
 */
const int voltagePin = 11;      // use pin 11 as voltage
const int groundPin = A1;       // use pin A1 as ground

void setup() {
  // initialize serial:
  Serial.begin(115200);
  // set powerPin and groundPin as digital outputs:
  pinMode(voltagePin, OUTPUT);
  pinMode(groundPin, OUTPUT);
  // set them high and low respectively:
  digitalWrite(voltagePin, HIGH);
  digitalWrite(groundPin, LOW);
}

void loop() {
  // if serial available, send average
  if (Serial.available() > 0) {
    int inByte = Serial.read();
    int sensorReading = analogRead(A0);
    float voltage = map(sensorReading, 0, 1023, 0,3.7);
    Serial.println(voltage);
  }
}
```

Log It The PHP script uses the $_REQUEST array variable to get the results of the HTTP request. Even though the Processing sketch is making a GET request, this script doesn't care—it will read GET or POST.

First, it checks to see whether the $_REQUEST array has a value set. If so, it looks for a variable within it called data.

Next, it checks to see whether the datalog file exists. If so, it opens it and puts its contents in a variable called $currentData. Then it appends the new data from the client to the end of that variable, and overwrites the file with the result.

The HTML following the script echoes back to the client what it sent. Putting it in HTML format is handy for checking it in a browser.

```
<?php
/*
          Data logger
          Context:  PHP
*/

// name of the file on the server
// where you want to save stuff:
$dataFile = 'datalog.txt';

// see if the client uploaded a file:
if (isset($_REQUEST)) {
   $newData = $_REQUEST['data'];
   if (file_exists($dataFile)) {
      // Open the existing file to get existing content
      $currentData = file_get_contents($dataFile);
      // Add what you got from the client:
      $currentData .= $newData;
      // Write everything back to the existing file
      file_put_contents($dataFile, $currentData);
   }
}
?>

<html>
   <body>
          Here's what you sent:
      <?php echo $newData; ?>
   </body>
</html>
```

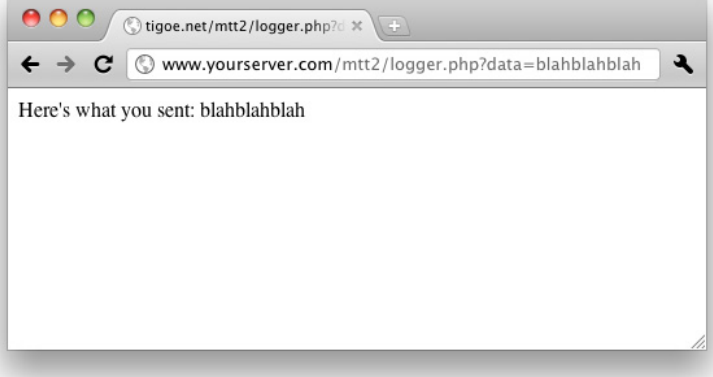


Figure 10-23
What the logger script returns in a browser.

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

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

If you're creating the file using a GUI-based program, get info on the file and set the permissions that way. Figure 10-24 shows the Get Info window from BBEdit, which is similar to many other programs. Once you've made this file, call it from the browser with a query string like this:

```
http://www.yourserver.com/logger.php?data=blahblahblah
```

You should get a page that looks like Figure 10-23. Open the **datalog.txt** file. You'll see something like this:

```
blahblahblah
blahblahblah
blahblahblah
```

If you do, you know everything is working and you're ready to write the Processing sketch.
X

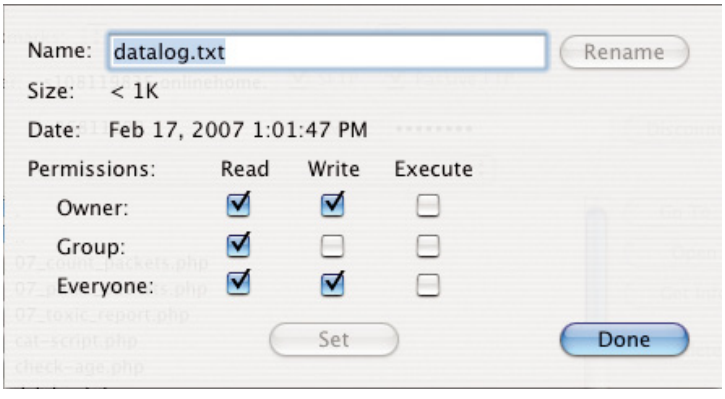


Figure 10-24
Setting the read-write permissions for a file from a GUI-based program.

Bluetooth Serial Library

In order to use the Processing for Android sketch, you'll need the Bluetooth Serial (BtSerial) library. This library allows you to make serial connections over Bluetooth using similar commands to those you used with the serial library in Processing's Standard mode. You can download it from https://github.com/arduino/BtSerial. Click the Downloads link and pick the latest download package (0.1.6 as of this writing). Make sure to download the download package not the source. Unzip the file. You'll get a directory called btserial-0.1.6. Inside it is a directory called **btserial**. Place that in the libraries folder of your Processing sketch directory, as you've done with other libraries.

In Android mode in Processing version 1.5.1, libraries are not always imported into your sketch automatically. This will be fixed in future versions so that it works like the Standard mode. If when you try to run your sketch you get an error that says package cc.arduino.btserial does not exist, make a subdirectory, **code**, in the sketch's directory and copy the file **btserial.jar** from the **library** subdirectory of the **btserial** directory that you just installed into **code**. Then try running your sketch again.

This is not ideal, but it should be addressed by the time this book is in publication—the Processing team is very good with bug fixes.
X

Send It The global variables for this sketch include: values for the sensor reading interval, in seconds, and the interval for updating the server, in minutes; the times of the last read and send; the address of your server PHP script; the state of the connection; two buttons for reading and sending on demand; and the color scheme for the app.

```
/*
 Datalogger
 Receives data via Bluetooth every ten seconds
 Uploads data via HTTP every two minutes
 Context:  Processing, Android mode
 */
import cc.arduino.btserial.*;
// instance of the library:
BtSerial bt;

int readInterval = 10;      // in seconds
int sendInterval = 2;       // in minutes

int lastRead = second();     // the seconds last time you read
int lastSend = minute();     // the minutes last time you read
String lastSendTime;         // String timestamp of last  server update

// URL of your PHP Script:
String url = "http://www.yourserver.com/logger.php?data=";
String currentReadings = ""; // group of readings, with datestamps
String thisReading;          // most recent reading

String connectionState = ""; // connected to Bluetooth or not?

Button readButton;           // Button for prompting immediate read
Button sendButton;           // Button for prompting immediate send
boolean updateNow = false;   // flag to force an update
boolean sendNow = false;     // flag to force a send

// color scheme from http://kuler.adobe.com
// deep optimism by nicanore:
color bgColor = #2B0D15;
color textColor = #FFEB97;
color buttonColor = #565F63;
color buttonHighlightColor = #ACBD9B;
```

» Change this to match your PHP script's URL.

» The setup() method sets the background and fill colors, and initializes the text parameters. Then, it makes a new instance of the BtSerial library and tries to connect to the Bluetooth Mate using the connect() method. Finally, it initializes the two onscreen buttons.

```
void setup() {
  // set color scheme:
  background(bgColor);
  fill(textColor);

  // Setup Fonts:
  String[] fontList = PFont.list();
  PFont androidFont = createFont(fontList[0], 24, true);
  textFont(androidFont, 24);

  // instantiate the library:
  bt = new BtSerial( this );
```



Note that the button and text positions are all relative to the screenWidth and screenHeight variables. This is so that they stay in the same relation to each other and the screen if the device is rotated.

Continued from opposite page.

```
  // try to connect to Bluetooth:
  connectionState = connect();

  readButton = new Button(screenWidth/2 - 100, 2*screenHeight/3,
     200, 60, buttonColor, buttonHighlightColor, "Get Reading");
  sendButton = new Button(screenWidth/2 - 100, 2*screenHeight/3 + 80,
     200, 60, buttonColor, buttonHighlightColor, "Send Reading");
}
```

The draw() method starts by drawing the text strings and buttons. Then it checks to see whether the sensor reading interval has passed, or whether the updateNow variable has been set (when you click the Read Now button, this variable is set to true). If either of these is true, the sketch takes a reading via Bluetooth using the getData() method, and adds it to a string of readings taken since the last time it updated the server.

Next, it checks whether the send interval or sendNow has been set by a click of the Send Now button. If either of these is true, it sends the current string of readings to the server using the sendData() method.

```
void draw() {
  // display data onscreen:
  background(bgColor);
  fill(textColor);
  textAlign(LEFT);
  text(connectionState, 10, screenHeight/4);
  text(getTime(), 10, screenHeight/4 + 60);
  text("latest reading (volts): " + thisReading, 10, screenHeight/4 + 90);
  text("Server updated at:\n" + lastSendTime, 10, screenHeight/4 + 120);

  // draw the buttons:
  readButton.display();
  sendButton.display();

  if (sendNow) {
    textAlign(LEFT);
    text("sending to server, please wait...", 10, screenHeight/4 - 60);
  }

  // if the update interval has passed,
  // or updateNow is true, update automatically:
  if (abs(second() - lastRead) >= readInterval || updateNow) {
    thisReading = getData();

    // if you got a valid reading, add a timestamp:
    if (thisReading != null) {
      currentReadings += getTime() +"," + thisReading;
      // take note of when you last updated:
      lastRead = second();
      // you've updated, no need to do it again until prompted:
      updateNow = false;
    }
  }

  // if the send interval has passed,
  // or sendNow is true, update automatically:
  if (abs(minute() - lastSend) >= sendInterval || sendNow ) {
    sendData(currentReadings);
    // get the time two ways:
```

Last, the draw() method reads the buttons, then sets their previous states once it's read them. This sketch uses a modified version of the Button class from the Processing RFID writer sketch in Chapter 9. Because the cursor on a touchscreen (namely, your finger) can disappear, there's no equivalent to the mousePressed and mouseReleased events, so you have to make one up. You're doing that in this case by having a method in the Button class called isPressed(), a variable called pressedLastTime, and two methods that let you get and set this variable. With these, you can check the current state of the button (isPressed()), save it when you're done (setLastState()), and get the state last time you checked it (getLastState()).

Continued from previous page.

```
    lastSendTime = getTime();    // a String to print on the screen
    lastSend = minute();         // an int for further comparison
  }

  // if the read button changed from not pressed to pressed,
  // set updateNow, to force an update next time through the
  // loop. Do the same for the send button and sendNow, right below:
  if (readButton.isPressed() && !readButton.getLastState()) {
    updateNow = true;
  }
  //save the state of the button for next check:
  readButton.setLastState(readButton.isPressed());

  if (sendButton.isPressed() && !sendButton.getLastState()) {
    sendNow = true;
  }
  //save the state of the button for next check:
  sendButton.setLastState(sendButton.isPressed());
}
```

When the sketch pauses, the pause() method is called. This method sends any current readings to the server, then disconnects the Bluetooth serial connection so you can open it again on resume. There is no resume() method, since setup() gets called by default every time an Android activity resumes (remember, a Processing sketch for Android is an activity).

```
void pause() {
  // if you have any readings, send them:
  if (!currentReadings.equals("")) {
    sendData(currentReadings);
  }
  // stop the Bluetooth connection so you can start it again:
  if (bt != null && bt.isConnected()) {
    bt.disconnect();
  }
}
```

The connect() method tries to make a Bluetooth connection. First, it makes sure there's a valid instance of the BtSerial library and that it's not already connected to the remote device (the Bluetooth Mate in this case). Then it gets a list of all paired devices on your phone and tries to connect to the first one.

Before you run the sketch, go to the Bluetooth Settings on your phone (under Settings→Wireless Settings), and pair with your Bluetooth Mate. If

```
String connect() {
  String result = "Bluetooth not initialized yet...";
  if (bt !=null) {
    // if you are connnected, get data:
    if (!bt.isConnected() ) {
      // get the list of paired devices:
      String[] pairedDevices = bt.list();

      if (pairedDevices.length > 0) {
        println(pairedDevices);
        // open a connection to the first one:
        bt.connect( pairedDevices[0] );
        result = "Connected to \n" + bt.getName();
      }
```

» you don't, the app can't see the Bluetooth Mate. If your Bluetooth Mate isn't the first item in the list of devices, be sure to change the call to:

```
bt.connect( pairedDevices[0] );
```

to match the number of your Bluetooth Mate in the list.

Continued from opposite page.

```
    }
    else {
      result = "Couldn't get any paired devices";
    }
  }
  return result;
}
```

» The getData() method checks to see that there's a valid Bluetooth connection. Then, it sends a byte prompting the Arduino to send a reading in return. It adds all the bytes it gets in response to a result string.

Since the Bluetooth connection might get broken, the method checks to see that what it got ended with a newline, which is the last byte that the Arduino will send. If the method has a valid reading string, it returns it. Otherwise, it returns null.

If there isn't a valid connection, getData() attempts to make one.

```
String getData() {
  String result = "";

  if (bt != null) {
    // if you are connnected, get data:
    if ( bt.isConnected() ) {
      // send data to get new data:
      bt.write("A");
      // wait for incoming data:
      while (bt.available () == 0);
      // if there are incoming bytes available, read them:
      while (bt.available () > 0) {
        // add the incoming bytes to the result string:
        result += char(bt.read());
      }
      // get the last character of the result string:
      char lastChar = result.charAt(result.length() - 1);
      // make sure it's a newline, or you don't have valid data:
      if (lastChar != '\n') {
        result = null;
      }
    } // if you're not connected, try to pair:
    else {
      connectionState = connect();
    }
  }
  return result;
}
```

The sendData() method sends the current readings to the server using Processing's loadStrings() method. loadStrings() does a basic HTTP GET request, and returns whatever the server sends it.

```
void sendData(String thisData) {
  // if there's data to send
  if (thisData != null) {
    // URL-encode the data and URL:
    String sendString = formatData(url + thisData);
    //send the data via HTTP GET:
    String[] result = loadStrings(sendString);
    // clear currentReadings to get more:
    String currentReadings = "";
  }
}
```

URL strings need to be formatted cleanly (no spaces, no newlines or carriage returns, and so forth), so sendData() calls formatData(), which converts spaces, newlines, and returns into their HTTP-safe equivalents.

```
String formatData(String thisString) {
  // convert newlines, carriage returns, and
  // spaces to HTML-safe equivalent:
  String result = thisString.replaceAll(" ", "%20");
  result = result.replaceAll("\n", "%0A");
  result = result.replaceAll("\r", "%0D");
  return result;
}
```

The final method in the main sketch is getTime(), which just returns a formatted date/time string.

```
// get the date and time as a String:
String getTime() {
  Date currentDate = new Date();
  return currentDate.toString();
}
```

The Button class for this sketch differs from the one for the RFID Writer sketch in Chapter 9, in that it has a variable and methods to track its pressed state the last time you checked it.

```
// The Button class defines the behavior and look
// of the onscreen buttons.  Their behavior is slightly
// different on a touchscreen than on a mouse-based
// screen, because there is no mouseClick handler.
class Button {
  int x, y, w, h;                    // positions of the buttons
  color basecolor, highlightcolor;   // color and highlight color
  color currentcolor;                // current color of the button
  String name;                       // name on the button
  boolean pressedLastTime;           // if it was pressed last time

  // Constructor: sets all the initial values for
  // each instance of the Button class
  Button(int thisX, int thisY, int thisW, int thisH,
      color thisColor, color thisHighlight, String thisName) {
    x = thisX;
    y = thisY;
    h = thisH;
```

The constructor for this Button class is almost the same, except for the new variable, pressedLastTime.

Continued from opposite page.

```
    w = thisW;
    basecolor = thisColor;
    highlightcolor = thisHighlight;
    currentcolor = basecolor;
    name = thisName;
    pressedLastTime = false;
  }
```

The display() method combines the functions of update() and display() from Chapter 9's RFID Writer example, both changing the color as needed and drawing the actual button.

```
  // draw the button and its text:
  void display() {
    // if pressed, change the color:
    if (isPressed()) {
      currentcolor = highlightcolor;
    }
    else {
      currentcolor = basecolor;
    }
    fill(currentcolor);
    rect(x, y, w, h);

    //put the name in the middle of the button:
    fill(textColor);
    textAlign(CENTER);
    text(name, x+w/2, y+h/2);
  }
```

isPressed() checks not only whether the mouseX and mouseY are within the button's bounds, but it also uses mousePressed to indicate whether the user is touching the screen at all.

```
  // check to see if the mouse position is inside
  // the bounds of the rectangle and sets its current state:
  boolean isPressed() {
    if (mouseX >= x && mouseX <= x+w &&
      mouseY >= y && mouseY <= y+h && mousePressed) {
      return true;
    }
    else {
      return false;
    }
  }
```

setLastState() and getLastState() give you access to the last state of the button from outside the class.

```
  //this method is for setting the state of the button
  // last time it was checked, as opposed to its
  // current state:
  void setLastState(boolean state) {
    pressedLastTime = state;
  }
  boolean getLastState() {
    return pressedLastTime;
  }
}
```

That's the end of the whole sketch; see Figure 10-24 for an illustration.

When you run this sketch on an Android device, it will try to connect. When it succeeds in doing so, it will check for new readings every 10 seconds. It will then attempt to send those to the server every two minutes. It doesn't check that it got a valid response, however; that is left as an exercise for you.

Armed with this much, there's a lot you can do with Processing for Android. You've got the ability to contact the Internet and to contact local devices wirelessly. You can also contact the built-in sensors, of course. The phone is now your hub to connect a whole lot of things.

What About USB?

Android devices are normally USB end devices, not USB hosts. So, you can't plug an end device like a mouse, a keyboard, or an Arduino into them and have them work like they do on your computer. However, Google decided that you might want to develop USB accessories, so they recently announced the Google Accessory Development Kit, an open hardware platform for developing USB accessories for Android. The Accessory Development Kit is based on an Arduino Mega 2650 with a USB Host shield (originally designed by Oleg Mazurov of www.circuitsathome.com) built into it. Circuits@Home sells a few good variants on the USB Host shield: for the standard Arduino shield form and for the Mini Pro. The Arduino Store carries the Arduino Mega ADK, and a Processing for Android library is in development. Several others are in the works from different companies, so by the time you read this, you'll have lots of options.

X

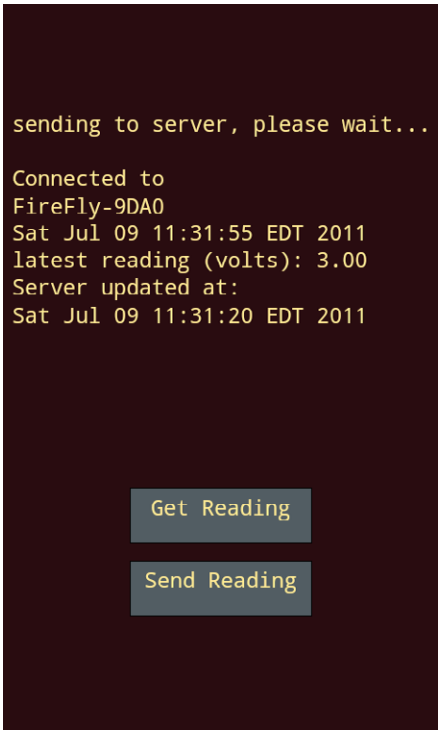


Figure 10-25
The datalogger sketch.



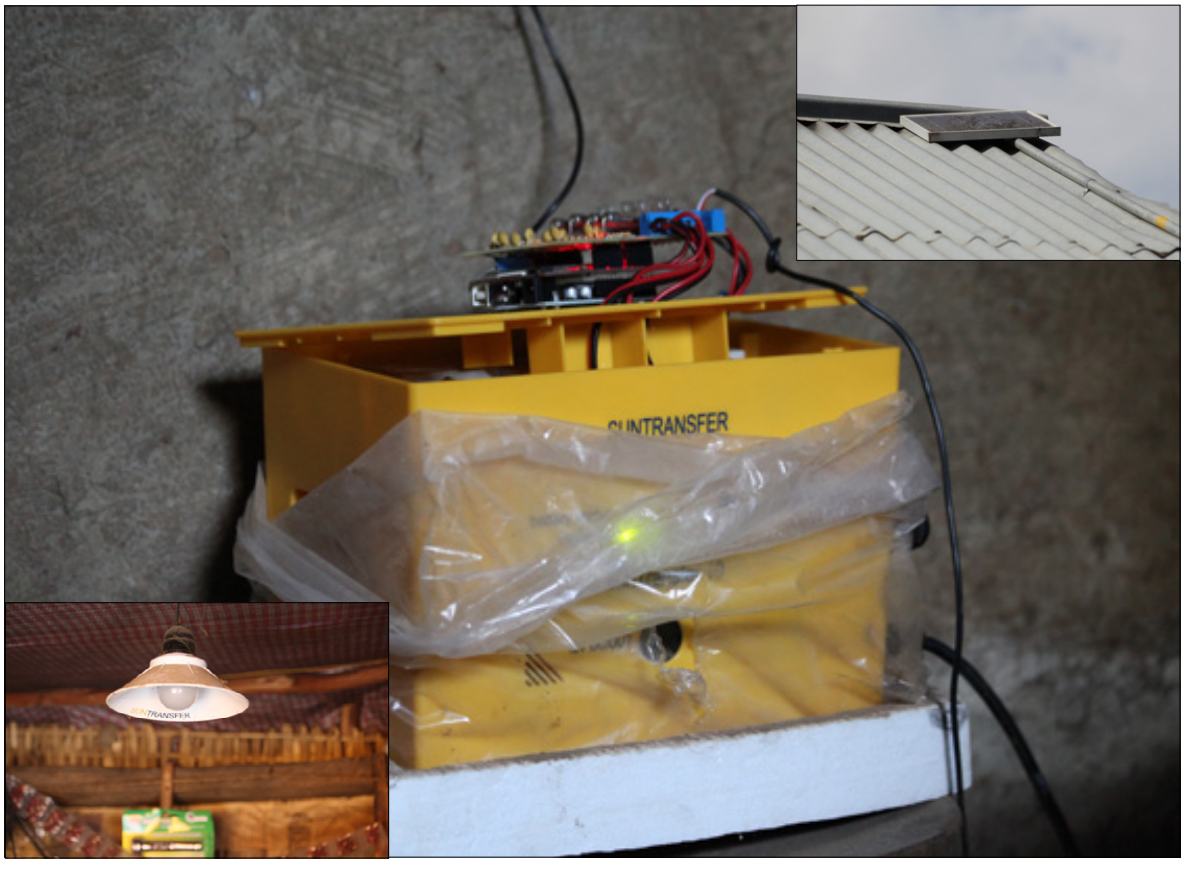
Figure 10-26
The datalogger hoodie in action. How excited is he to be wearing it? Check his GSR readings online to find out!

Conclusion

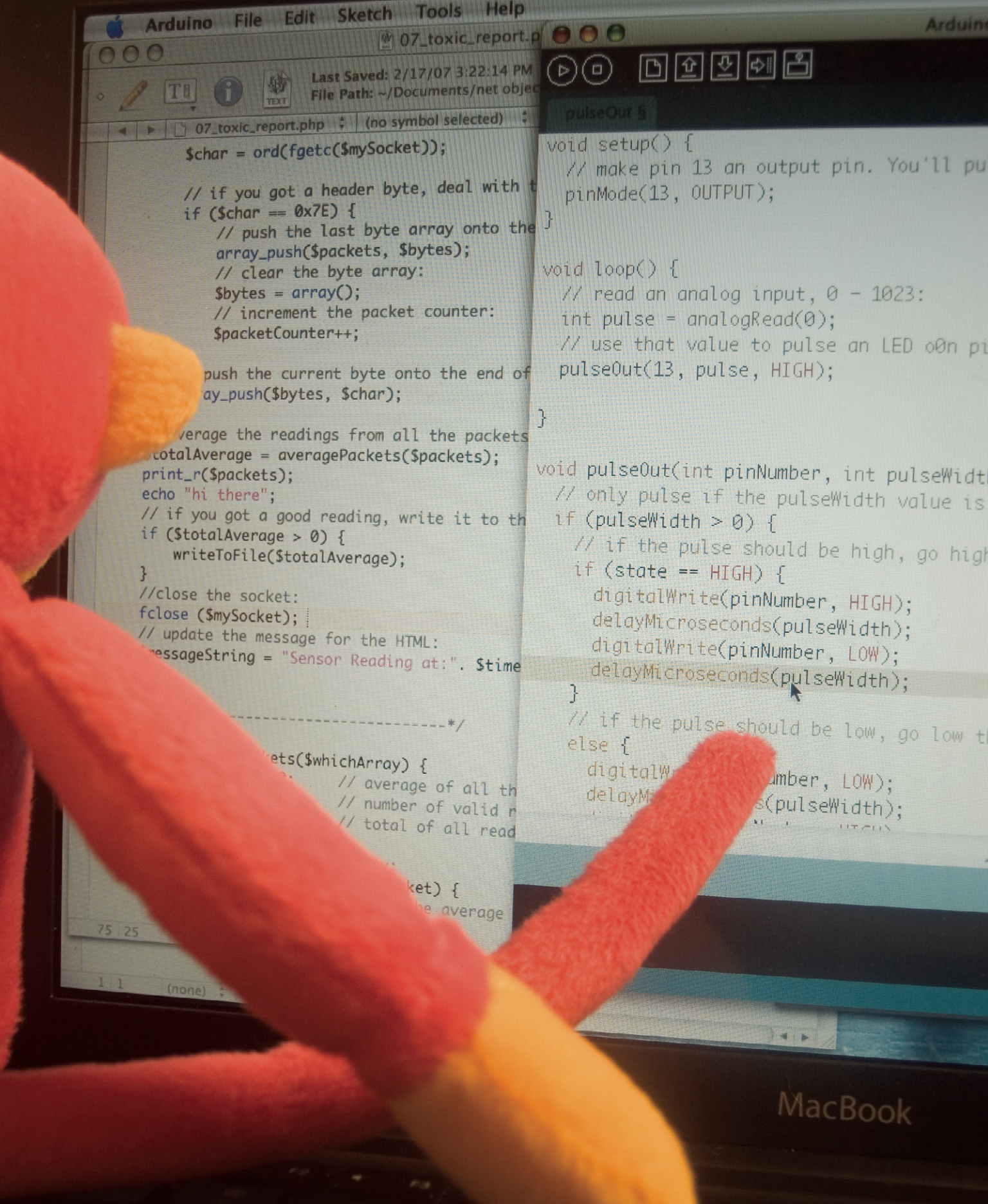
The connection between the Internet and mobile networks widens your options by offering a ubiquitous connection and a variety of communication methods. Taking advantage of it requires some creative thinking about how to hop across different interfaces, systems, and protocols.

These days, mobile phone networks cover almost the entire planet, more than any other form of network connection. The technology of mobile networks changes quickly, and many of the tools disappear almost as fast as they appeared. If you're making projects using mobile network technologies, it's best to take a broad view. Look for the things that seem simpler and stabler—like SMS and HTTP—and be prepared to shift your approach when the tools change. In Chapter 11, you'll review some of the protocols you've seen throughout the book in order to get a wide view of what's possible.

X



SIMbaLink, by Meredith Hasson, Ariel Nevarez, and Nahana Schelling. The SIMbaLink team developed a device that remotely monitors the health of a solar home system and reports it back to the SIMbaLink client website via a GPRS modem. Working in conjunction with a solar company in Ethiopia, SIMbaLink remotely monitored solar home systems outside Awassa, Ethiopia. The systems consisted of a 10W solar panel, a battery, and four 1-watt LED lamps. This simple setup brings light to an otherwise dark and unpowered rural home. *Photos courtesy of Meredith Hasson.*


Arduino File Edit Sketch Tools Help
07_toxic_report.p
Last Saved: 2/17/07 3:22:14 PM
File Path: ~/Documents/net objec
07_toxic_report.php
(no symbol selected)
$char = ord(fgetc($mySocket));
// if you got a header byte, deal with t
if ($char == 0x7E) {
// push the last byte array onto the
array_push($packets, $bytes);
// clear the byte array:
$bytes = array();
// increment the packet counter:
$packetCounter++;
push the current byte onto the end of
ay_push($bytes, $char);
verage the readings from all the packets
totalAverage = averagePackets($packets);
print_r($packets);
echo "hi there";
// if you got a good reading, write it to th
if ($totalAverage > 0) {
writeToFile($totalAverage);
}
//close the socket:
fclose ($mySocket);
// update the message for the HTML:
essageString = "Sensor Reading at:". $time
*/
ets($whichArray) {
// average of all th
// number of valid r
// total of all read
ket) {
e average
75 25
1 1
(none)
Arduino
pulseOut §
void setup() {
// make pin 13 an output pin. You'll pu
pinMode(13, OUTPUT);
}
void loop() {
// read an analog input, 0 - 1023:
int pulse = analogRead(0);
// use that value to pulse an LED o0n pi
pulseOut(13, pulse, HIGH);
}
void pulseOut(int pinNumber, int pulseWidt
// only pulse if the pulseWidth value is
if (pulseWidth > 0) {
// if the pulse should be high, go high
if (state == HIGH) {
digitalWrite(pinNumber, HIGH);
delayMicroseconds(pulseWidth);
digitalWrite(pinNumber, LOW);
delayMicroseconds(pulseWidth);
}
// if the pulse should be low, go low t
else {
digitalW umber, LOW);
delayM s(pulseWidth);
MacBook

Chapter 11 

MAKE: PROJECTS

Protocols Revisited

One of the most valuable skills you can develop is the ability to create graceful transitions between seemingly incompatible systems. An understanding of different protocols is central to that skill. You've used a number of communications protocols in this book, but by the time you read this, the protocols you learned may have changed or become obsolete. However, if you understand how to learn new ones, you'll do fine. This final chapter is a view from the high ground, looking over the landscape of protocols you've seen in this book to consider their similarities and differences, how they all fit together, and how to go about learning new ones in the future.

Supplies for Chapter 11

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww)
- **AF** Adafruit (www.adafruit.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **MS** Maker SHED (www.makershed.com)
- **RS** RS (www.rs-online.com)
- **SF** Spark Fun (www.sparkfun.com)
- **SS** Seeed Studio (www.seeedstudio.com)

PROJECT 32: Fun with MIDI

» **1 Arduino module** An Arduino Uno or something based on the Arduino Uno, but the project should work on other Arduino and Arduino-compatible boards. **D** 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4

» **1 Spark Fun Musical Instrument shield** **SF** DEV-10587

» **1 MaxBotix LV-EZ1 sensor** **SF** SEN-00639

» **100µF capacitors** **J** 158394, **D** P10269-ND, **F** 1144642, **RS** 715-1657

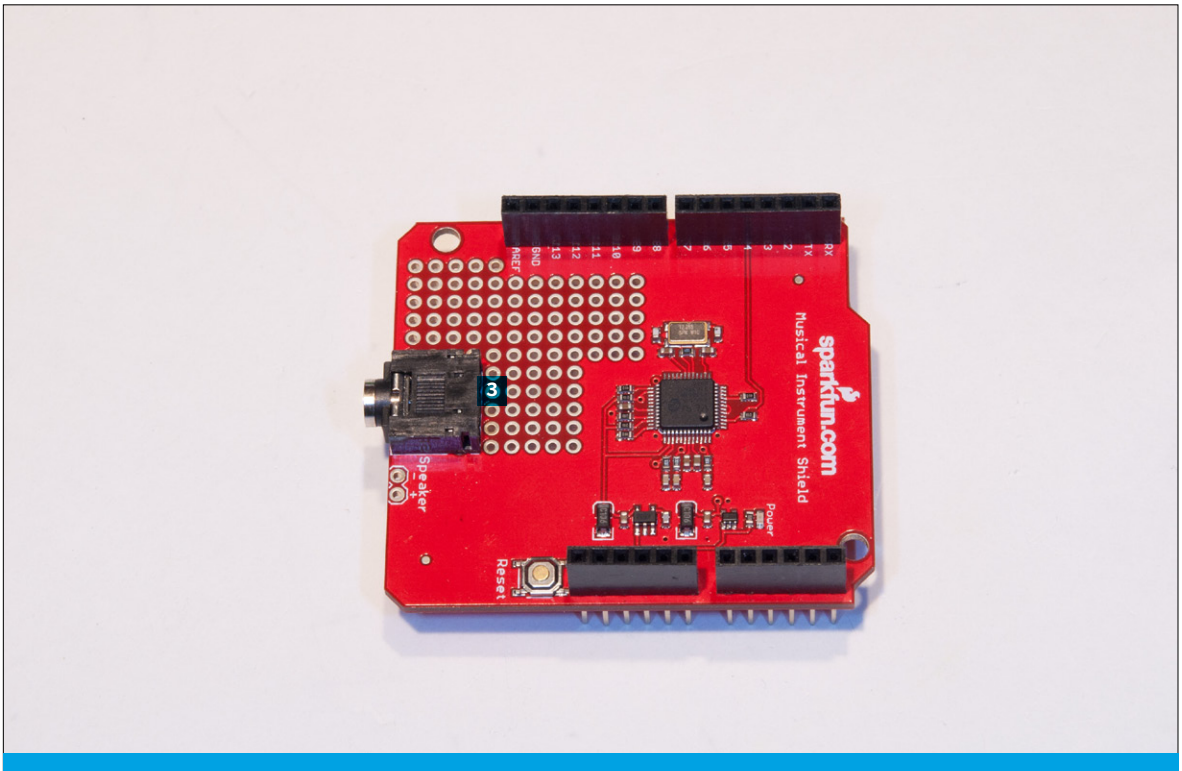


Figure 11-1. New parts for this chapter: Spark Fun Musical Instrument shield.

Make the Connections

Protocols are much easier to learn when you can relate them to ones you know already. Fortunately, you now know several. You can compare them at any of the levels discussed in Chapter 2. The three that will affect your project the most are the physical, the data, and the application. You can also consider the structures of the networks on which they communicate: direct one-to-one connections, networks with a central hub or controller, multitiered networks, rings, and buses. Once you know the structure of the system, you can consider the grammar and syntax of the communications protocols being used. Each of these things will help you translate from one device or system to another, and each aspect of the communication will tell you something about the other aspects.

For example, if you're dealing with a protocol that uses addresses, you can be sure that it's designed for communication between more than two things. If the address is broken into several pieces, like IP addresses, you can assume it's probably used on a multitiered network.

Know Your Options at the Physical Level

When you start to work with a new electronic device, get familiar with what protocols it speaks. By now you're

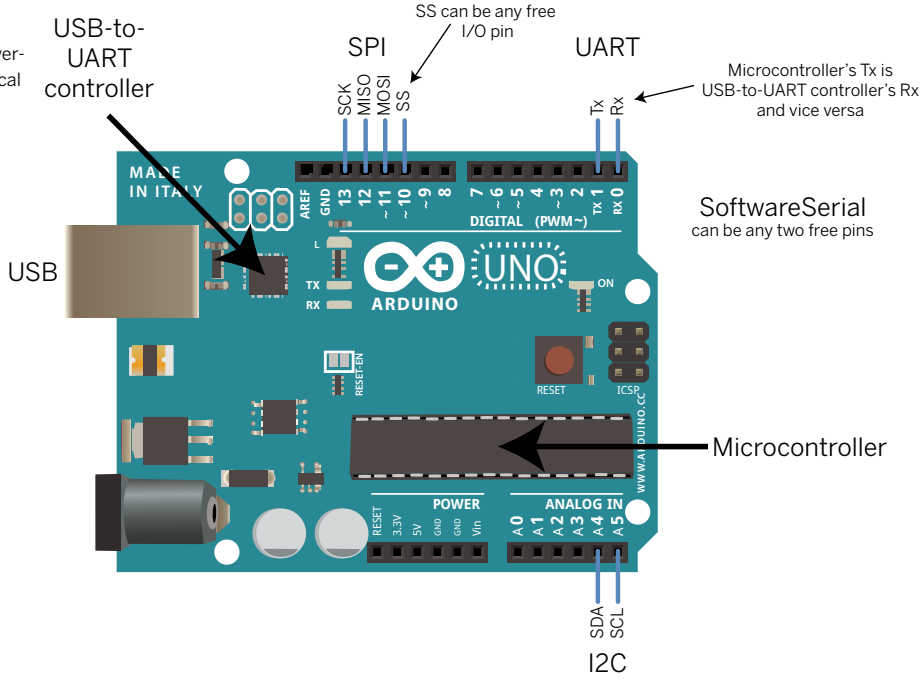


Figure 11-2
The Arduino's communications options. Asynchronous serial ports are referred to as Universal Asynchronous Receiver-Transmitters (UARTs) in technical documentation, so get familiar with the term.

familiar with the various protocols that the Arduino speaks. Figure 11-2 gives you an overview of them.

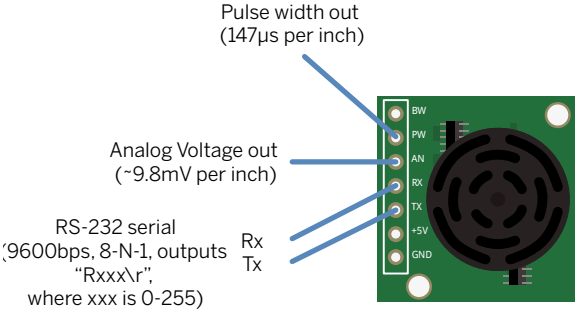
This figure tells you something about the physical layer (which pins are used for which modes of communication) and the data layer (which data protocols the device can speak). You know by now that if you're using a particular set of pins for its communications functions, you can't use it for other functions.

Most of the single-purpose devices you used in the book spoke only one protocol, but that's not always the case. For example, the ultrasonic ranger you used in Chapter 8 offers three different interfaces. It has an analog voltage output (0-5V, the one you used); an asynchronous RS-232 serial port (Tx and Rx) that sends data at 9600 bits per second; and a pulse-width output (PW), which outputs a pulse that goes from low to high to low again, changing the width of the pulse with the distance of the detected object. All of this is described in the ranger's datasheet, which you can find online at www.maxbotix.com. These multiple protocols are useful when you're interfacing the sensor to a microcontroller with no analog inputs; when you want to talk directly to a computer that has an RS-232 serial port; or when you're not using a computer at all, and want to control a simple circuit with a changing pulse width or voltage.

The SonMicro RFID reader you used in Chapter 9 also offered two different communications options: TTL serial and I2C. The serial port was useful for communicating directly with your personal computer; the I2C option was useful when working with the microcontroller, because it left the microcontroller's serial port open for programming and debugging.

When you're planning a project, consider which of your devices' ports are used for configuring, programming, and debugging, and try to leave them free when possible. On the other hand, if the programming interface offers the fastest and most reliable means of communication, it may be worthwhile to deal with having to use it for both programming and its final application.

Figure 11-3
MaxBotix LV-EZ-1 communications options.



Picking a Serial Protocol

When planning a project, picking a serial protocol may seem confusing: synchronous or asynchronous? TTL serial or RS-232? I2C or SPI? USB? Most of the time, the choices are made for you because there's some component that you have to use—and it only speaks one protocol. But when you have a choice, here are a few things to consider.

What protocols are common to all the devices in your system? If all of your devices speak the same protocol, it's probably the obvious choice.

Do your devices need to be able to decide independently when they'll send information? If so, then any system that combines them in a master-slave relationship, like any of the synchronous serial protocols, won't work.

Do you need multiple devices to communicate over the same wires? If so, consider protocols that use a bus system. Synchronous serial protocols and some asynchronous ones, like USB and RS-485, use such a system. All of these require some form of addressing.

Is distance or speed a factor? Many protocols that need to be able to transmit over long distances use differential signaling, in which there's both a data+ and data- line. The same data is sent over both lines, always adding up to zero volts. This minimizes electrical noise. USB, RS-485, and Ethernet all use differential signaling. Differential signaling generally allows for faster data rates as well.

This table shows a summary, and lists the characteristics, of a few serial protocols mentioned in the book.

Synchronous Serial Protocols	Asynchronous Serial Protocols	Asynchronous Serial Bus Protocols
• Master-slave relationship between devices • Several devices share the same data lines • Needs a clock line from the master to slaves • Distance: 1m or less • Various voltage ranges, commonly 5V or 3.3V	• No common clock, but agreement on data rate • Typically used for one-to-one communication, not networks	• No common clock, but agreement on data rate • Often used for longer distances or networks • Typically differential signaling • 3 to 4 wires: Data+, Data-, Ground, Voltage (optional)
I2C/TWI • 3 wires: Data, Clock, Ground • Each slave gets a unique address, sent first	**TTL Serial** • Various voltage ranges, commonly 5V or 3.3V • 3 wires: Transmit, Receive, Ground • Logic: +V = logic 1, 0V = logic 0 • Distance: typically 1m or less	**USB** • Voltage range: 0 to 5V • Distance: 10m or less
SPI • 4 + n wires: Master In Slave Out (MISO), Master Out Slave In (MOSI), Clock, Ground, 1 Chip Select for each slave device • Addressing is done using chip select lines	**RS-232** • Voltage range: ±3 to ±15V • Logic: -V = logic 1, +V = logic 0 • 0V has no logical meaning • Distance: 300m or less	**RS-485 (DMX-512)** • Voltage range: –7 to +12V • Distance: 1200m or less

Plan the Physical System and Information Flow Early

When you know your physical communications protocol options, you can plan how things communicate before you begin to program or configure anything. Doing so will save you a lot of time.

Consider the CatCam web server in Chapter 10. It involved several devices:

- Microcontroller
- Ethernet controller
- SD card
- Temperature sensor
- Relay control
- IP-based camera

To plan a project like that, you can (and should) diagram the flow of information between all the elements, as you saw in Figures 10-3 and 10-4, so you know what each one needs from the others. Consider what pins of your microcontroller are available for the various functions you need. Decide where images and web pages are best served from, and what links and permissions are required to make it all happen. Consider which way data flows, and who needs to know whose address. Decide where you'll place the server and the camera physically, and how you will house them. That way, you can reduce your actual production workload, and concentrate on building only what you need in code, in circuits, and in materials.

Text or Binary?

One of the most confusing aspects of data communications is understanding the difference between a binary protocol and a text-encoded protocol. As you learned in Chapter 2, there is a common scheme for translating the data you send into alphanumeric characters. ASCII—and its more modern cousin, Unicode—allow you to convert any stream of text into binary information that a computer can read, and vice versa. Understanding the difference between a protocol that is text-based and one that's not, and why you'd make that choice, is essential to understanding how electronic things communicate.

Isn't All Data Binary?

Well, yes. After all, computers only operate on binary logic. So, even text-based protocols are, at their core, binary. However, it's often easier to write your messages in text and insist that the computers translate that text into bits themselves. For example, take the phrase "1-2-3, go". You can see it laid out in characters below, with the ASCII code for each character, and the bits that make up that code. It may seem like a lot of ones and zeroes for the computer to process, but when it's reading them at millions or billions of bits a second, it's no big deal. But what about when it's sending it to another computer? There are 80 bits there. Imagine you're sending it using TTL serial at 9600bps. If you add a stop bit and a start bit between each one—as TTL and RS-232 serial protocols do—that's 96 bits, meaning you could send this message in 1/100th of a second. That's still pretty fast.

Character	**1**	**-**	**2**	**-**	**3**	**,**		**g**	**o**	**!**
ASCII code	49	45	50	45	51	44	32	103	111	33
Binary	00110001	00101101	00110010	00101101	00110011	00101100	01000000	01100111	01101111	01000001

What about when the messages you have to send are not text-based? Imagine you're sending a string of RGB pixel values, each ranging from 0 to 255 like this:

```
102,198,255,127,127,212,255,155,127,
```

You know that each reading is eight bits, so in their most basic form, they take up three bytes per pixel. But if you're sending them in text form it's one byte per character, so the string above would be 36 bytes to send nine values. If you hadn't encoded it in text form, but just sent each value as a byte, you'd have only sent nine bytes. When you start to send millions of pixels, that's a big difference! In cases like this, where all the data you want to send is purely numeric (you don't need any text to describe the pixels because you know the first byte is always red, the second green, the third blue, and so on), sending data without encoding it as text makes sense. When you need to send text (for example, email or hypertext), encoding it as ASCII/Unicode makes sense. When the number of bytes you'd send is minimal, as with most of the sensor strings you've sent in this book, encoding it as ASCII/Unicode can help with debugging, because most serial or network terminal programs interpret all bytes that they receive as text, as you've seen. So:

- If there's a lot of data and it's all numeric, send as raw binary.
- If there's text, send as Unicode/ASCII.
- If it's a short data string, send as whichever makes you happy.

Interpreting a Binary Protocol

Since most of the protocols you've dealt with in this book have been text-based, you haven't had to do a lot of interpreting of the bits of a byte. Binary protocols often demand that you know which bit represents what, so it's useful to know a little about the architecture of a byte and how to manipulate it. Let's now talk for a bit about bits.

Binary protocols often show up in communications between chips in a complex device, particularly in synchronous serial protocols. Many SPI and I2C devices have small command sets. Their single-byte operational codes, or opcodes, are often combined with the parameters for the commands in the same byte. You saw this in action in Chapter 9 when you sent the opcode to read the firmware of the SM130 RFID reader.

These protocols are usually written out in hexadecimal notation, binary notation, or both. You've seen hexadecimal or base-16 notation already in this book. Just as hexadecimal numbers begin with 0x, binary numbers in Arduino—and by extension in C—begin with 0b, like so 0b10101010.

Which digit matters most in the number below?

```
$2,508
```

The 2, because it represents the largest amount, two thousands, or two groups of 10^3. It is the most significant digit. That number was in decimal, or base-10 notation. The same principle applies when you're writing in binary, or base-2. Which is the most significant bit:

```
0b10010110
```

The leftmost 1 is most significant because it represents 1 group of 128, or 2^7. Usually when you see bits written out, though, you care less about their decimal numeric values than their position in the byte. For example, in the project that follows, you're going to make music using MIDI, the Musical Instrument Digital Interface protocol. MIDI is a binary protocol in which all bytes with a 1 in the most significant bit are commands (verbs), and all bytes with a 0 in the most significant bit are data bytes (nouns, adjectives, or adverbs). A MIDI interpreter could look at the most significant bit (which happens to be the first to arrive serially) and know whether it's getting a command or not.

Bit Reading and Writing

Arduino offers you some commands for reading and writing the bits of a byte:

```
// to read the value of a bit:
myBit = bitRead(someByte, bitNumber);

// to write the value of a bit:
bitWrite(someByte, bitNumber, bitValue);
```

Giving a bit the value 1 is called setting the bit in general programming terms, and giving it the value 0 is known as clearing the bit. So you also have commands for these:

```
// to make a bit equal to 1:
bitSet(someByte, bitNumber);

// to make a bit equal to 0:
bitClear(someByte, bitNumber);
```

Bit Shifting

Sometimes it's easier to manipulate several bits at once. The shift left and shift right operators in Arduino, C, and Java allow you to just that. The shift left operator (<<) moves bits to the left in a byte, and the shift right operator (>>) moves them to the right:

```
0b00001111 << 2; // gives 0b00111100
0b10000000 >> 7; // gives 0b0000001
```

Bit shifting is useful when you need a particular value to be in a specific part of a byte, as you'll see shortly.

Bit Masking

The logical operators AND, OR, and XOR allow you to combine bits in some interesting ways.

AND (&): if the two bits are equal, the result is 1. Otherwise, it's 0:

```
1 & 1 = 1
0 & 0 = 0
1 & 0 = 0
0 & 1 = 0
```

OR (|): if either bit is 1, the result is 1. Otherwise, it's 0:

```
1 | 1 = 1
0 | 0 = 0
1 | 0 = 1
0 | 1 = 1
```

XOR (^): if the bits are not equal, the result is 1. Otherwise, it's 0:

```
1 ^ 1 = 0
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
```

Using the logical, or bitwise, operators, you can isolate one bit of a byte, like so:

```
// Check if bit 7 of someByte is 1:
if (someByte & 0b10000000) {}
```

In MIDI command bytes, for example, the command is the leftmost four bits, and the MIDI channel is the rightmost four bits. So, you could isolate the MIDI channel by using the AND operator:

```
channel = commandByte & 0b00001111;
```

Or, you could use bit shifting to get the command and lose the channel:

```
command = commandByte >> 4;
```

Combined, these bit-manipulation commands give you all the power you need to work with binary protocols.

Hex: What Is It Good For?

Since you can manipulate binary protocols bit by bit, you're probably wondering what hexadecimal notation is good for. Hex is useful when you're working in groups of 16, of course. For example, MIDI is grouped into banks of instruments, with 128 instruments per bank, and each instrument can play on up to 16 channels. So, those command bytes you were manipulating earlier could also be manipulated in hex. For example. 0x9n is a Note On command, where n is the channel number, from 0 to A in hex. 0x9A means note on, channel A (or 10 in decimal). Similarly, 0x8A means note off, channel A. Once you know that MIDI is organized in groups of 16, it makes a lot of sense to read and manipulate it in hexadecimal. In fact, many binary protocols can be grouped similarly, so knowing how to manipulate them in binary or hexadecimal is handy.

The next project puts these principles in action to control a MIDI synthesizer.

X

MIDI

The Musical Instrument Digital Interface (MIDI) protocol enables real-time communication between digital musical instruments. It's the granddaddy of digital synthesizer protocols.

Most music synthesizers, sequencers, samplers, keyboards, and workstations on the market today speak MIDI. If you plan to make music using hardware, you're going to run across it. MIDI is a comprehensive specification covering serial communication, hardware, and even the arrangement of banks of sounds in a synthesizer's memory. Because it's so comprehensive, it's enabled composers and musicians to work across a wide range of MIDI synthesizers, samplers, and controllers for decades. The full protocol is detailed on the MIDI Manufacturers' Association page at www.midi.org.

MIDI is a music description protocol. MIDI messages don't actually play notes—they describe what notes to play to a synthesizer. In that sense, MIDI is to music as HTML is to web pages. MIDI relies on a synthesizer to render music, just as HTML relies on a browser to render your page.

MIDI devices are loosely grouped into controllers, or things that generate MIDI messages, and playback devices, which receive messages and do something with them. Keyboards and other input devices fall into the first category. Synthesizers, samplers, and many other devices fall into the latter.

The MIDI serial protocol runs at 31,250 bps. There's a standard MIDI connector, a DIN5, that you'll find on all MIDI gear. All the connectors on the gear are female plugs, and the cables all have male connectors on both ends. All MIDI inputs to a device are supposed to be opto-isolated, so there is no direct electrical connection from one device to the next. An opto-isolator is a component containing an LED and a phototransistor. The input turns on the LED, and the LED triggers the phototransistor, which turns on the output. Figure 11-4 shows how to build a MIDI input and output circuit if you plan to build your own. This circuit would allow you to connect your microcontroller to any MIDI synthesizer or controller. There are several MIDI shields on the market if you don't want to build your own.

Microcontrollers are more frequently used as MIDI controllers than as MIDI input devices. There are so many good synthesizers and samplers on the market that just need a controller to make music, and controllers are fun and easy to build. For the project that follows, you'll build a very basic controller and use Spark Fun's Musical Instrument shield as your MIDI input device, which is a general MIDI synthesizer on a shield. The same code will work if you want to connect the microcontroller to any general MIDI synth, because the general MIDI specification covers not only the messages, but how synthesizers organize their banks of sounds.

MIDI messages are divided into three or more bytes. The first byte, a command byte, is always 128 (0x80) or greater in value. Its value depends on the command. The bytes that follow it are called status bytes. All status bytes have values of 127 (0x7F) or less.

There are a number of different MIDI commands. The most basic, note on and note off messages, control the playing of notes on 16 different channels of a synthesizer. Each note on or note off command contains two status bytes, specifying the pitch in a range from 0–127 (a seven-bit range) and the velocity (how hard the note should be struck), which is also seven bits. Pitch value 0x45 is defined as A above middle C (A-440) by the general MIDI specification. This specification also covers the instruments that you're likely to find on each channel. For a full listing of the General MIDI instrument specification, see www.midi.org/techspecs/gm1sound.php.

For more information on MIDI, see Paul D. Lehrman and Tim Tully's book **MIDI for the Professional** (Amsco).

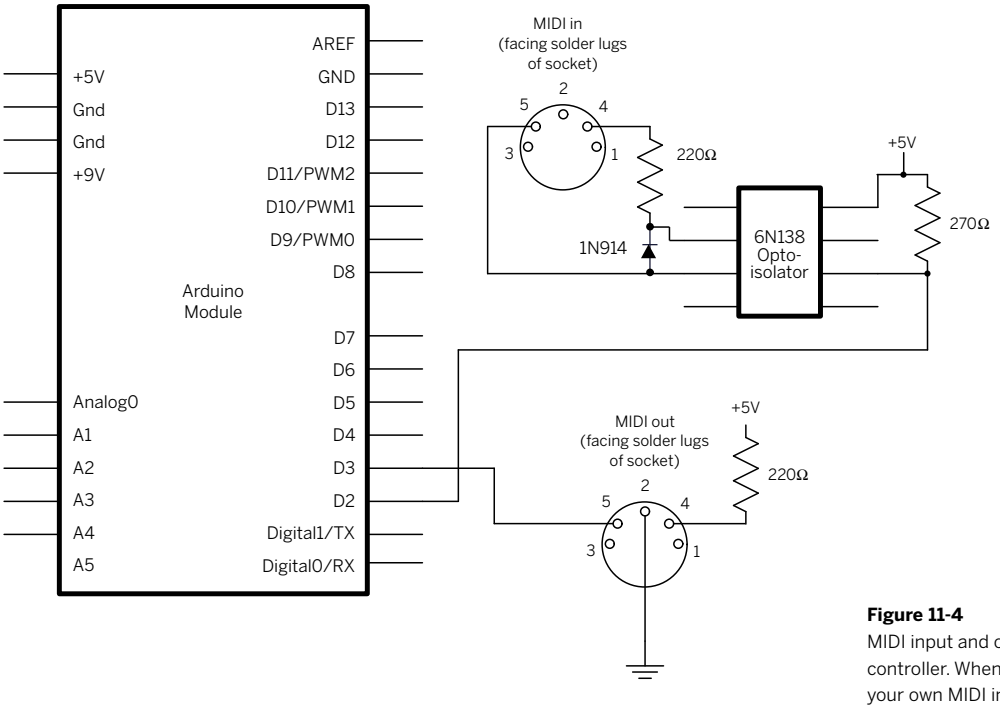
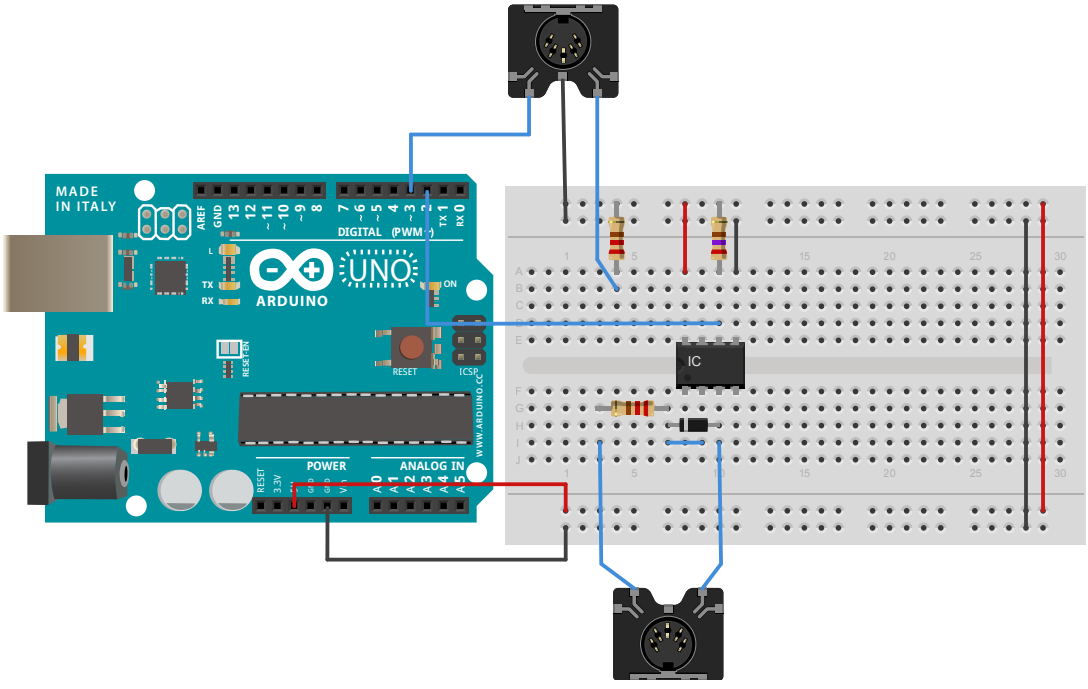


Figure 11-4
MIDI input and output from a microcontroller. When you have to build your own MIDI input or output, this is how you do it.



Project 32

Fun with MIDI

In this project, you'll see how a binary protocol is handled by running some basic MIDI exercises. Because of the wide range of MIDI controllers and input devices on the market, MIDI offers a very broad range of sonic possibilities. So, MIDI is a good protocol to be familiar with if you like sound projects.

The best way to get started with MIDI is to build a very simple controller, connect it to a synthesizer, and see what it can do. If you have your own synthesizer, you can use the circuit in Figure 11-4; if not, you can use the Musical Instrument shield as shown in Figure 11-5. It just stacks on top of an Arduino like other shields. The synth receives MIDI in from the Arduino's pin 3. To communicate with it, you'll use the SoftwareSerial library.

MATERIALS

- **1 Arduino**
- **MIDI Musical Instrument shield**
- **1 Sharp IR ranger**
- **1 100µF capacitor**
- **1 pair headphones or speakers**

The first sketch below is a test of the available instruments and channels. You'll notice that for each instrument, channel 10 is percussion—that's also part of the general MIDI instrument specification. This specification makes it possible for you to use a controller on the same channel and instrument with any MIDI synthesizer, and know more or less what kind of sound you'll get.

Play It

Start by setting up your pins, as usual. MIDI out will be on pin 3, and the Music Instrument shield reset pin is on pin 4. If you're using an external synth and the circuit in Figure 11-4, you won't need the reset connection.

setup() starts the serial and MIDI connection, and resets the shield. Then it sends a MIDI Control Change command on channel 0 (0xB0) to perform a Bank Selection. This sets the bank of instruments you'll play.

This example is based on Nathan Seidle's example sketches for the Musical Instrument Shield, available at www.sparkfun.com/products/10587.

```
/*
 MIDI general instrument demo
 Context: Arduino

 Plays all the instruments in a General MIDI instrument bank
 */
#include <SoftwareSerial.h>
// set up a software serial port to send MIDI:
SoftwareSerial midi(2,3);

const byte resetMIDI = 4; // Midi synth chip reset line

void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
  // initialize MIDI serial on the software serial pins:
  midi.begin(31250);

  //Reset the MIDI synth:
  pinMode(resetMIDI, OUTPUT);
  digitalWrite(resetMIDI, LOW);
  delay(20);
  digitalWrite(resetMIDI, HIGH);
  delay(20);
  // send a MIDI control change to change to the GM sound bank:
  sendMidi(0xB0, 0, 0);
}
```

» loop() cycles through all the instruments in the bank. For each instrument, it performs a MIDI Program change to select the instrument; then, it cycles through each of the 16 channels. For each channel, it plays notes from 21 (A0, the lowest note on an 88-key piano) to 109 (C8, the highest note).

```
void loop() {
  //Cycle through all the instruments in the bank:
  for(int instrument = 0 ; instrument < 127 ; instrument++) {
    Serial.print(" Instrument: ");
    Serial.println(instrument + 1);
    // Program select. Has only one status byte:
    sendMidi(0xC0, instrument, 0);
    // change channels within the instrument:
    for (int thisChannel = 0; thisChannel < 16; thisChannel++) {
      Serial.print("Channel: ") ;
      Serial.println(thisChannel + 1);
      for (int thisNote = 21; thisNote < 109; thisNote++) {
        // note on
        noteOn(thisChannel, thisNote, 127);
        delay(30);

        // note off
        noteOff(thisChannel, thisNote, 0);
        delay(30);
      }
    }
  }
}
```

» noteOn(), noteOff(), and sendMidi() are methods to send MIDI commands. The sendMidi() method just sends the three bytes you give it. The noteOn() and noteOff() methods use a bitwise OR operation to combine the command (0x80 or 0x90) with the channel into a single byte.

When you run this sketch, you should hear all the sounds your synth can make on the first bank of sounds.

```
//Send a MIDI note-on message.  Like pressing a piano key
//channel ranges from 0-15
void noteOn(byte channel, byte note, byte velocity) {
  sendMidi( (0x90 | channel), note, velocity);
}

//Send a MIDI note-off message.  Like releasing a piano key
void noteOff(byte channel, byte note, byte velocity) {
  sendMidi( (0x80 | channel), note, velocity);
}
void sendMidi(int cmd, int data1, int data2) {
  midi.write(cmd);
  midi.write(data1);
  midi.write(data2);
}
```

Wave at It

Of course, you can't be a self-respecting MIDI enthusiast until you've built your first theremin. It's like calling yourself a writer before you've written your auto-biography—it's just not done! Waving at a distance ranger to make notes is a rite of passage in MIDI-land. So, here's your first MIDI theremin program. This one's a bit different than most, as you'll see.

The global variables include a threshold setting for the infrared ranger, and a variable to hold the previous reading.

setup() opens communications and resets the shield.

```
/*
  Stinger player
  Context:  Arduino
*/

#include <SoftwareSerial.h>
// set up a software serial port to send MIDI:
SoftwareSerial midi(2, 3);

const int midiResetPin = 4;     // Musical instrument shield's reset pin
const int threshold = 100;      // sensor threshold
int lastReading = 0;            // last sensor reading

void setup() {
  // initialize hardware serial and MIDI serial:
  Serial.begin(9600);
  midi.begin(31250);

  // reset the musical instrument shield:
  resetMidi(midiResetPin);
}
```

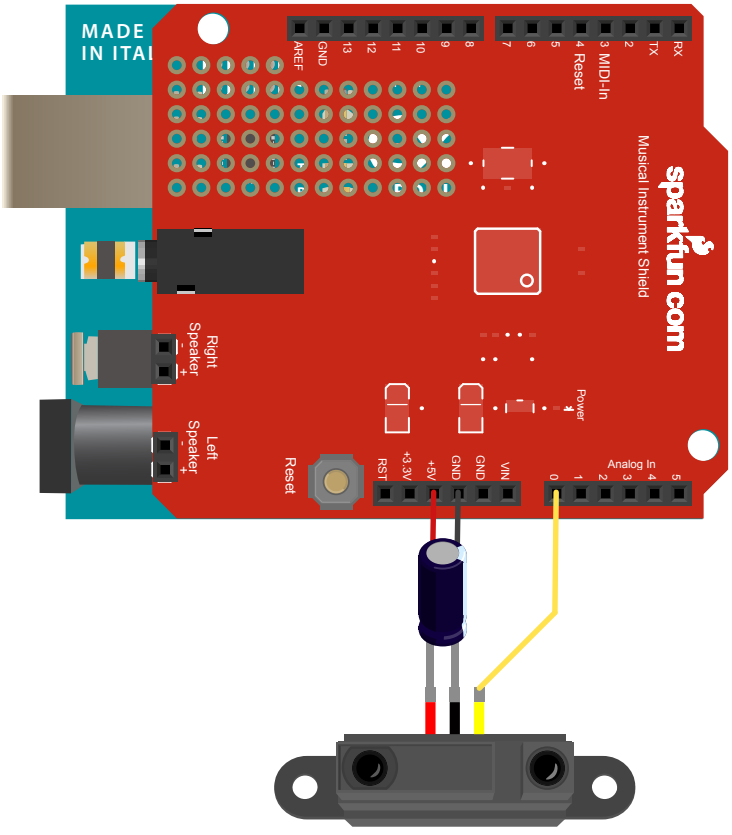


Figure 11-5
Spark Fun Musical Instrument shield attached to an Arduino and an infrared ranger. The 100µF capacitor smoothes the sensor's readings a bit, but it's optional if your sensor gives reliable readings without it.

The loop() reads the sensor, and when it crosses a threshold, it triggers a method called playStinger() that plays a sequence of notes.

```
void loop() {
  // read the sensor:
  int sensorReading = analogRead(A0);
  Serial.println(sensorReading);

  // if the sensor's higher than the threshold and
  // was lower than the threshold last time,
  // then play the stinger:
  if (sensorReading <= threshold
    && lastReading > threshold ) {
    playStinger();
  }
  // save the current reading for next time:
  lastReading = sensorReading;
}
```

playStinger() plays a sequence of notes with a sequence of pauses in between. To do this, it calls notes from an array using noteOn() and noteOff(), and then pauses, according to the rest times given by an array.

noteon(), noteOff(), and sendMidi() are the same as in the previous example, and you can copy them from there.

```
void playStinger() {
  int note[] = {43, 41, 49};
  int rest[] = {70, 180, 750};

  // loop over the three notes:
  for (int thisNote = 0; thisNote < 3; thisNote++) {
    // Turn on note:
    noteOn(9, note[thisNote], 60);
    delay(rest[thisNote]);
    //Turn off the note:
    noteOff(9, note[thisNote], 60);
    // a little pause after the second note:
    if (thisNote == 1) {
      delay(50);
    }
  }
}
//NOTE: the noteOn(), noteOff() and sendMidi() methods from the previous
//example go here.
```

resetMidi() takes the reset from the previous example out of setup() and puts it in its own method.

When you've uploaded this sketch, say the following:

"A guy walks into a bar and says 'OUCH!'"

Then wave your hand over the IR ranger. You're a comedian! And you've made your first MIDI instrument!

```
void resetMidi(int thisPin) {
  // Reset the Musical Instrument Shield:
  pinMode(thisPin, OUTPUT);
  digitalWrite(thisPin, LOW);
  delay(100);
  digitalWrite(thisPin, HIGH);
  delay(100);
}
```

DMX512

DMX512 is another binary protocol worth knowing about. It's a real-time serial protocol for communicating between stage-lighting control systems and lighting dimmers. It has been the industry standard for stage lighting and show control equipment for a couple of decades now. It's also used to control special-effects machines, moving lights, projection systems, and more.

DMX512 uses the RS-485 serial protocol as its physical layer. RS-485 is a bus-style serial protocol, meaning you can have several devices sharing the same transmit and receive lines, as long as each has a unique address. It uses differential signaling so that signal wires can be very long—up to 1200m. DMX512 devices generally use an XLR-style connector, either three pin or five pin, though many devices now use an RJ-45 Ethernet-style connector.

At 250kbps, DMX512 is fast for a serial protocol; it's fast enough that you can't just send it as regular TTL serial data from a microcontroller. However, there is a DMX library for Arduino called Simple DMX, and there are a few shields on the market that have DMX connectors on them. I've only tested two personally, both of which work well: the Tinker DMX shield is available from the Arduino Store at http://store.arduino.cc/ww/, and Daniel Hirschmann's Super DMX shield is available from http://www.hirschandmann.com/2011/super-dmx-shield-for-arduino/. The former lets you add your own connector; the latter has XLR, RJ-45 connectors, and terminals for you to add your own. Further notes can be found on the Arduino playground site at www.arduino.cc/playground/Learning/DMX. For more on DMX, see www.opendmx.net.

Like MIDI, DMX is an aging protocol. The lighting industry has started to develop its successor, Advanced Controller Network, or ACN, which can run over Ethernet. Over the next few years, you can expect to see it grow in prominence. For more on show control protocols in general, John Huntington's book **Control Systems for Live Entertainment** (Focal Press) can't be beat.
X

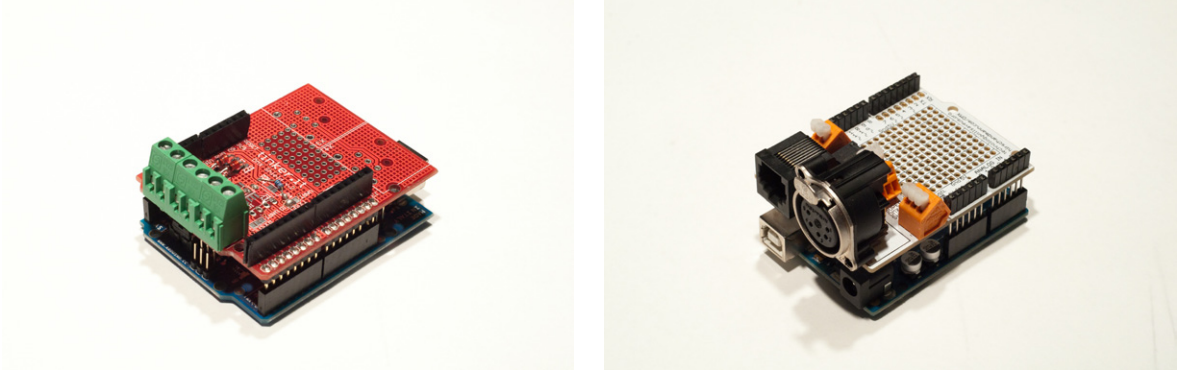


Figure 11-6
Tinker DMX shield, left, and Daniel Hirschmann's Super DMX shield, right. Both allow you to communicate with DMX512-based lighting and show control systems.

The Structure and Syntax of Text-Based Protocols

You've seen a number of text-based protocols throughout this book, and you've even written a few of your own. Protocols are basically ways of structuring data. Some protocols simply relay information, and others give commands—either explicitly or implicitly—through a request. It's worth reviewing some of their grammars so that when you run into similar ones, you'll know what to look for.

Simple Data Formats

The simplest was the comma-separated value string you wrote for Monski Pong. Comma-separated values (CSV) and tab-separated values (TSV) are ubiquitous, and it's common to end a CSV string with a newline, or a newline and carriage return. The NMEA-0183 protocol used by GPS receivers is a good example of CSV in practice.

You also saw a few examples of organizing data in name-value pairs. Any time you have a list of items, and each has a name associated with it, you've got a name-value pair. The environment variables in PHP—including $_GET, $_POST, $_REQUEST, and $_SERVER—give you name-value pairs. In programming languages, associative arrays are basically lists of name-value pairs.

You've also used a few Internet transfer protocols, like HyperText Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP). They share a common format as well: they open with a command like GET or POST, and then follow with properties of the transfer, each on its own line. Each line starts with the description of the property, like the content-length, separated from the actual value by a colon. The header of the transfer is separated from the body of the message by two newlines, and the message is usually closed with two newlines as well. Any data coming from the client was sent as name-value pair arrays, separated by a colon for information about the transfer, or by an ampersand for data items in the content of the transfer. Take a second look at this HTTP POST request from Chapter 2 as an example:

```
POST /age_checker.php HTTP/1.1
Host: example.com
Connection:Close
Content-Type: application/x-www-form-urlencoded
Content-length: 16

name=tom&age=14
```

Everything in the header is a parameter of the request: who you're requesting from, the connection type, the type of content to be exchanged, the content length. Everything in the body is a parameter of the content: name and age. Having a clear separation between the header formatting and the content formatting makes it easier to separate them when writing a program, as you've already seen.

Structured Data Formats

When you've got data to exchange that's more complex than a list of name-value pairs, you need a structure for it. For example, imagine an array of sensors spread around your house:

Address	Location	Last read	Value
1	kitchen	12:30:00 PM	60
2	living room	05:40:00 AM	54
3	bathroom	01:15:00 AM	23
4	bedroom	09:25:00 AM	18
5	hallway	06:20:00 AM	3

You've got more than just a list of single items, and more than just a few name-value pairs. In fact, each line of the table is a list of name-value pairs. This is where a structured data format comes in handy. JavaScript Object Notation, or JSON, is a popular notation format for structured data like this.

JSON represents each cell of the table as a name-value pair. Each line is a comma-separated list of pairs, enclosed in curly braces. The whole table is a comma-separated list of lists, like so:

```
[{"Address":1,"Location":"kitchen","Last read":"12:30:00 PM","value":60},
{"Address":2,"Location":"living room","Last read":"05:40:00 AM","value":54},
{"Address":3,"Location":"bathroom","Last read":"01:15:00 AM","value":23},
{"Address":4,"Location":"bedroom","Last read":"09:25:00 AM","value":18},
{"Address":5,"Location":"hallway","Last read":"06:20:00 AM","value":3}]
```

This way of formatting data is relatively simple. The punctuation that separates each element is just a single character, so you can scan through it one character at a time and know when you're done with each element. Because it's text, it's human-readable as well as machine-readable. Spaces, newlines, and tabs aren't considered part of the structure of JSON, so you can reformat it for easier reading like so:

```
[
   {
     "Address":1,
     "Location":"kitchen",
     "Last read":"12:30:00 PM",
     "value":60
   },
   {
     "Address":2,
     "Location":"living room",
     "Last read":"05:40:00 AM",
     "value":54
   },
   {
     "Address":3,
     "Location":"bathroom",
     "Last read":"01:15:00 AM",
     "value":23
   },
   {
     "Address":4,
     "Location":"bedroom",
     "Last read":"09:25:00 AM",
     "value":18
   },
   {
     "Address":5,
     "Location":"hallway",
     "Last read":"06:20:00 AM",
     "value":3
   }
]
```

The advantage of a data interchange format like JSON is that it's lightweight, meaning there aren't a lot of extra bytes needed to structure the information you want to send. It gives you more power than a simple list but is still efficient to send from a server to a client. You can send JSON as the body of an HTTP request, and as long as there's a program on the client that can parse it, you've got a quick way to exchange complex data.

When you've got something more complex to represent, you need a markup language. Markup languages provide a way to describe the structure of a text document in detail. Markup languages divide data into the content and the markup, a series of descriptive tags for organizing the content. The content is what you want to present; the markup describes how to present the content, what to do with it—and in some cases—how to interpret it.

Markup tags are typically strings of text separated from the content by a special pair of punctuation marks. You've seen them frequently in HTML:

```
<title> this is the document title</title>
```

If a tag has any attributes, they go inside the tag brackets as well:

```
<a href="www.example.com>A link</a>
```

Strict markup languages like XML always expect that every opening tag has a closing tag. Markup languages are strict in their formatting because they're designed to be machine-readable. A computer program can't assume where you meant to put a closing tag, so it will just keep looking until it finds the next one. If it finds a second opening tag before it gets to a closing tag, it gets confused.

Markup languages describe the structure of the document and how it's to be presented, but they don't tell you anything about what's in the document. They are to text documents what MIDI is to music: it describes the parameters of how to play a note without telling you anything about the actual timbre of the note.

You may have noticed that none of the transfer protocols you saw used the characters < and >. That is one reason why HTML uses these for tag markup. In parsing an HTTP response, you can know that you've reached the main text when you hit the first < character. Similarly, wiki markup typically uses square brackets, [and], to enclose tags. This helps distinguish it from HTML or XML markup. That way, the same document can contain both types of markup without causing conflict.

The bytes that form a web page can pass through a lot of programs before they reach the end, each using a different protocol. When each protocol is designed to take the others into account—as you can see with HTTP, HTML, and wiki markup languages—it's easy for each program to parse the bytes that matter to it and leave the rest alone.

Most of the time, you don't need all the information you're given in a typical information transfer. Knowing the protocol's structure and syntax not only helps you understand what's said, it also helps you determine what to ignore. Having to evaluate every chunk of data you receive can tax your system (and your patience) quite a bit. But when you know what you can ignore, you win. For example, consider the HTTP POST request that Twilio sends in Chapter 10. There's a huge amount of data there, more than 500 bytes. The only part you cared about, though, was the Digits property, so your sketch just scanned through the incoming stream of text, not saving anything until it saw the string "Digits". That kind of scanning and ignoring is central to good data parsing.

Markup vs. Programming Languages

Markup languages are primarily descriptive languages, not always active ones. They are interpreted by the program that displays them. For example, consider HTML, or HyperText Markup Language. HTML describes what a page should look like, but it doesn't have verbs to tell the browser to initiate action. It might describe an element to be displayed, such as a movie or audio file, and specify that the element be played on loading. There aren't HTML commands to make anything happen once the page is loaded. Markup languages usually rely on an outside actor, like the user, to prompt action.

In contrast, programming languages initiate action based on their own internal logic. Programming languages are sets of commands that make things happen. Your job as a programmer is to structure a set of commands to make things happen they way you want them to.

Programming languages have variables and arrays to manage simple data, but they generally don't dictate how you format complex data. The assumption is that you'll come up with a data format or protocol, and then use the programming language to read it byte-by-byte and interpret it, or to assemble it byte-by-byte. Useful programming languages come with libraries that can read and write common data formats, which you've seen throughout this book. Most of the libraries you used were designed to read or write a given data format. HTML5 represents a good combination of a markup language (HTML and CSS) and a programming language (JavaScript) that allows designers and developers to create full applications in a browser.

By now you've noticed that programming languages can be somewhat unforgiving when you leave out a comma or a semicolon, or get the capitalization of a word wrong. This is because the text of the programming language you write is actually a sort of markup language for the compiler. The compiler reads your text and translates it into machine language, so the text you write has to be machine-readable, just as markup languages and data formats do. The challenge to designing a good programming language is similar to designing a good data protocol: you want it to be efficient for the machine to read and interpret, yet still readable by humans.

The key to interpreting any data protocol—from the most simple comma-separated list to the most complex programming or markup language—is to start by asking what it's designed to do, and then look closely at its structure to see how it's organized. As you can see, many different protocols and languages share similar elements of punctuation or structure, so knowing one helps you to learn others.

X

Representational State Transfer

Underlying the HyperText Transfer Protocol (HTTP) is a principle known as Representational State Transfer, or REST. It's not a protocol—it's more of an architectural style for information exchange. Though it started with the Web, it has applications in other areas. Understanding a little about REST will not only help you understand other systems, it will design your own communications protocols as well. Once you know about it, you can never get enough REST in your life.

The basic idea of REST is this: there is a thing somewhere on a network. Maybe it's a database, or maybe it's a microcontroller controlling your household appliances, like you made in the previous chapter. You either want to know what state it's in, or you want to change its state. REST gives you a way to describe, or *represent*, the *state* of the thing on the Net, and to *transfer* that representation to a remote user. REST is an addressing scheme for any remote thing.

You've been using REST already—it's what HTTP is all about. You want to know about the thing, you make a request to GET that information or to POST changes to the thing. The thing responds to your request, either with a representation of the state of affairs (for example, the HTML page delivered by the air conditioner controller, which included representations of the temperature and the state of the thermostat), or by changing the state of affairs (for example, by updating the thermostat setting).

In RESTful thinking, URLs are nouns that describe things, and requests are the verbs that act on nouns. The properties of things are described in the URL, separated by slashes. For example, here's a RESTful description of one of an Arduino's pins:

```
/pin/A0/
```

If you want to know the state of that pin, you might say to the Arduino:

```
GET /pin/A0/
```

The Arduino would then reply with the state of pin A0, a number between 0 and 1023. Or, perhaps you want to change the state of a digital output pin. You might do this:

```
POST /pin/D2/1/
```

The Arduino would know that it should set digital pin 2 to be an output, and set it high (1 =HIGH). In this scenario, the Arduino is the server—serving you up representations of its state—and you're the client.

There are three important elements to this exchange:

1. The representation (called a resource) and its attributes are separated by slashes. You can think of the attributes like the properties of an object.
2. The verb is the request: whether you want to get some information or change the state of the thing, you start with the verb. The verbs are HTTP's verbs: GET, POST, PUT, and DELETE (the last two are used less often).
3. The description is technology-independent. There's no actual mention as to whether the resource is delivered to you as an HTML page or XML from a web server, or whether it's printed by a PHP or Ruby script on a server, a C/C++ program running on an Arduino, or black magic. It doesn't matter how the result is generated, you just want to know what the state of things is, and it's the server's job to do that.

The beauty of being technology-independent is that you can use REST for just about any control interface. For example, Open Sound Control, or OSC, a protocol for musical controllers that's designed to supersede MIDI, is RESTful. ACN, the Advanced Controller Networking protocol designed to replace DMX512, is also RESTful. It's also independent of the transport mechanism, and it can be sent over Ethernet, serial, or any other physical data transmission. REST provides a standard way of naming, or addressing, things so that you keep using that addressing scheme when you change environments, programming tools, or network technologies. It's simple enough that you can read it with a limited processor, and general enough to describe most anything you want information about or want to control.

There's another advantage to REST when designing websites: if you change the programming environment you use to run your site, you don't have to change the URLs. The client never sees the .php, .rb (Ruby), or .pl (Perl) file extension, so it doesn't know or care what programming environment powers your service. If you want to change languages, go ahead! The site structure can stay the same.

To design RESTfully on the Web, think of the site or device you're making in terms of its properties, and come up with an addressing scheme that gives clients access to view and change those properties appropriately. Practically, what it means is that all the URLs of your site end at the /. There's no index.html (or anything .html), and no GET query string. Query parameters are passed using POST, and they're generally the last item in the address, as you'll see below. Following are two examples: one is a traditional website and the other is a physical device on a network.

A Traditional Web Service

Perhaps you're making a social media site for runners to compare their daily runs. As part of it, each runner has a profile with various attributes for each day's run: the date (day, month, and year), the distance of the run, and the time of the run. For a run on 31 January 2012, the URLs to address those attributes might look like this.

To get the distance:

http://myrun.example.com/runnerName/31/1/2012/distance/

To set the distance to 12.56km:

http://myrun.example.com/runnerName/31/1/2012/distance/12.56

To set the time to 1 hour, 2 minutes, 34 seconds:

http://myrun.example.com/runnerName/31/1/2012/time/1:02:34

Notice how you can tell a call that gets the value of a parameter from one that sets the value: the former has nothing following the name of the parameter.

A Web-Based Device

Imagine you're building a device that controls the window blinds in your office. There are 12 windows, each with its own blind. The blinds can be positioned variably, in a range from 1 to 10. The client can get the state of any blind, or you can set it. The URLs to see the state of the windows might look like this.

To see them all at once:

http://mywindows.example.com/window/all

To see an individual window (window number 2):

http://mywindows.example.com/window/2

To set a window's blind halfway down:

http://mywindows.example.com/window/2/5

To close them all:

http://mywindows.example.com/window/all/0

If you don't like these particular addressing schemes, you could make your own, of course, as long as you followed the basic RESTful style of /object/attribute/ to get the value of a parameter, and /object/attribute/newValue to set it.

What the actual interface returned by the URL looks like is up to you. You might choose to make a graphic that shows images of the windows with blinds, or you might show the result in text only. RESTful architecture just determines where to find something, not what you'll find there.

RESTful addresses are designed to be easy for a computer to parse. All you have to do is to separate the incoming request into substrings on the slashes, look at each substring to see what it is, and act appropriately on the substrings that follow it. They're also easy for people to read. A URL like the ones above is much more comprehensible than the equivalent GET request:

http://myrun.example.com/?runnerName=George&day=31&month=1&year=2012&distance=12.56

The project that follows shows you how to set up a PHP-based server that can parse RESTful addresses.

Project 33

Fun with REST

If you've only done static HTML web design before, you're probably used to the idea that the slashes in a URL tell you in what directory on a server the final HTML file resides. However, it doesn't have to be that way. You can tell the server how to interpret the URL. In this project, you'll do just that.

Back in the "Network Identification" section of Chapter 9, you wrote a PHP script that printed out all of the environment variables returned by the server when you make an HTTP request. One of them was called REQUEST_URI. It gave you the string of text that follows the GET request from the client. That's the variable you want access to if you plan to build a RESTful web service. Each of the pieces of the request URI will be part of your interface, just like in the example URLs shown previously.

MATERIALS

» **Account on a web server**

It's most likely that your web server is running the Apache web server program. If so, you can create a special file inside any directory called **.htaccess** that determines the behavior of the server with respect to that directory. You can hide files and subdirectrories, password protect the directory, and more.

For this project, you'll set up a base directory for the project, and change the **.htaccess** file so that the server redirects anything that looks like a subdirectory to the script you're writing. Then you'll use the REQUEST_URI variable to make your own RESTful interface.

NOTE: URI stands for Uniform Resource Indicator. It's a synonym for Uniform Resource Locator, or URL, the term commonly used for web addresses.

Change the Access

Start by making a new directory on your web server. Call it **myservice**. Inside it, make a new file called **.htaccess**. Files that begin with a dot are invisible files on the Linux and Mac OS X operating systems. So it's best to make this file directly on the server, either through the command line or your favorite web-editing tool. Here's the text of the file.

```
RewriteEngine On
# put in the base path to your directory:
RewriteBase /~username/myservice
# redirect anything after the base directory to index.php:
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

This file tells the server to rewrite client HTTP requests starting with the name of this directory on your account. It takes any string after the directory name and replaces it with the index file, **index.php**.



In order for this to work, your web host needs to be running Apache's MOD_REWRITE module on its server. It's a pretty common module, so it's likely that it is, but if you find that this script doesn't work as described, check with your system administrator to see whether MOD_REWRITE is enabled.

Read It Here's the **index.php** file. It takes the $_REQUEST_URI variable and splits it up into an array of strings, using the slash (/) as the points on which to split.

Once you've saved this to the server, try entering the following URL in a browser:

http://www.myserver.com/myservice/this/is/a/list/of/restful/parameters/

You should get a response in the browser like this:

```
Item:
Item: myservice
Item: this
Item: is
Item: a
Item: list
Item: of
Item: restful
Item: parameters
```

Now that you've got anything that comes in from the client in an array, you can do whatever you want with it. This is where things get interesting. From here, you can write a program to look for and act on different elements in the list. The following is a brief example that would pull out the distance and time parameters based on the runners' website example above.

```
<?php
/*
          RESTful reader
          Context: PHP
*/
// split the URI string into a list:
$parameters = explode("/", $_SERVER['REQUEST_URI']);

// print out each of the elements of the list:
 foreach($parameters as $item) {
          echo "Item: ";
          echo $item."<br>";
 }
?>
```

Modify It Take out the foreach() block from the previous example and replace it as shown here. New lines are shown in blue.

```
<?php
// split the URI string into a list:
$$parameters = explode("/", $_SERVER['REQUEST_URI']);

$position = array_search('distance', $parameters);
if ($position) {
   $distance = $parameters[$position+1];
   echo "Your distance: ".$distance."<br />";
}


$position = array_search('time', $parameters);
if ($position) {
   $time = $parameters[$position+1];
   echo "Your time: ".$time."<br />";
}
?>
```

Type the URL as follows into your browser (change the hostname and path as needed):

http://www.example.com/myService/runnerName/31/1/2012/distance/12.45/time/0:45:34

When you type this into the browser, you'll get back:

```
Your distance: 12.45
Your time: 0:45:34
```

When you combine this approach with standard HTML forms using HTTP POST, you have two ways of passing in parameters, and your URLs reflect the state of the resource. It makes for a powerful and flexible way to build the architecture of a web service, resulting in URLs that are both machine-readable and easier for humans to understand.

Every server-side programming language has a variety of frameworks designed to make REST easier to implement. PHP frameworks like Cake and CodeIgniter, Ruby frameworks like Rails and Sinatra, and many others attempt to make it easier for you to build applications, and their application programming interfaces (APIs) are designed to encourage you to build in a RESTful way. With these, you can often build proxy applications that take data in a more complex format, and summarize it in a simpler format for the more limited devices in your system.

X

Conclusion

By definition, networked devices don't stand alone. If you're connecting a device to the Internet, you should take advantage of the power it offers. The more tools and protocols you know, the easier and more fun that is. Consider advantages offered by servers that have public addresses—and plenty of computing horsepower to spare. Consider combinations of wired and wireless protocols to give the things you build maximum freedom to respond to their physical environments.

As you can see by now, the physical interfaces that are the most flexible are usually the ones that are the most ephemeral on the network. They get turned on and off, they connect through private networks, and they use limited processors to save size, weight, and power. They're not always capable of doing everything that a dedicated web server can do, nor are they available at all times. However, these are not limitations when such devices are used in conversation with dedicated servers.

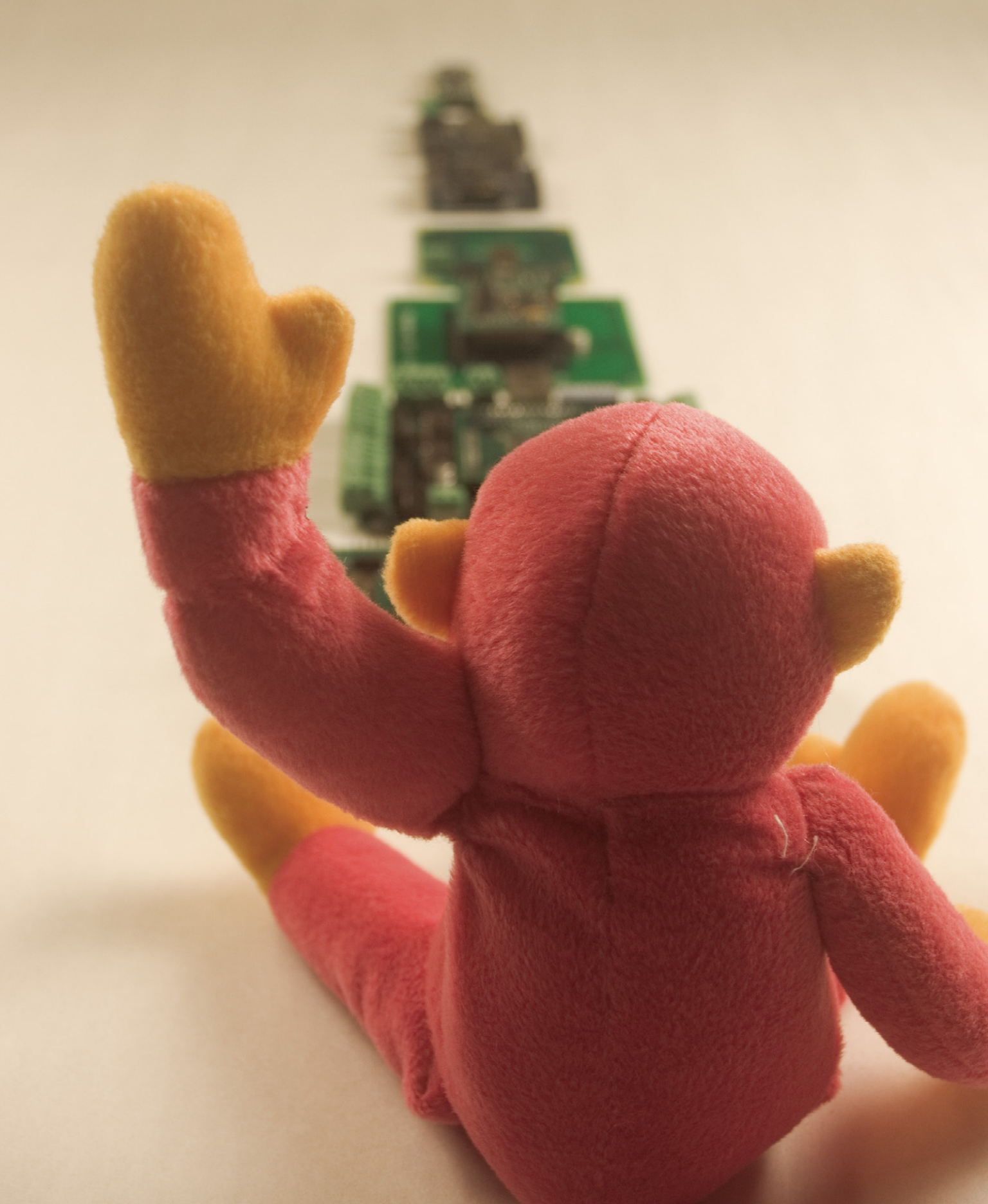
As hubs of your projects, use public, persistent servers that have public addresses and are always on the Net. They can store messages for physical devices that aren't online and deliver them later. They can act as proxies to take care of things for which simpler devices aren't designed, like complex data management or authentication. Don't be afraid to use tools and protocols in ways not thought of by their original designers. Make the technology fit the person's needs.

Making things talk to each other always comes back to two questions. Who are you serving in making what you're making? What does that person need, and how can you best give it to her?

To design a flexible network of physical things that's really useful to humans, you have to think about the person's needs and actions first. Then, it's all about the rules of love and networking:

- Listen More Than You Speak
- Never Assume
- Agree on How You Say Things
- Ask Politely for Clarification

The Internet of Things is useless if those things don't improve the quality of our lives and the ways in which we communicate with each other.




3.3V Zener Diode
TIP120 Transistors
2N2222 Transistors
green LEDs
red LEDs
yellow LEDs
Photo resistors
DC power connectors
DSUB9 female connectors for serial
nectors
Jumpers
straight headers
right angle headers



Where to Get Stuff

Many different hardware suppliers and software sources are mentioned in this book. This appendix provides a list of those parts and a summary of the vendors, along with a brief description of each. It's organized into three sections: Supplies, Hardware, and Software.

Supplies

Here's a shopping list of the parts used in this book. Each part lists the projects in which it's used.

If there are updates to this list, you'll find a link to them at http://oreilly.com/catalog/0636920010920.

DISTRIBUTOR KEY

- **A** Arduino Store (http://store.arduino.cc/ww/)
- **AF** Adafruit (www.adafruit.com)
- **CR** CoreRFID (www.rfidshop.com)
- **D** Digi-Key (www.digikey.com)
- **F** Farnell (www.farnell.com)
- **J** Jameco (http://jameco.com)
- **L** LessEMF (www.lessemf.com)
- **MS** Maker SHED (www.makershed.com)
- **P** Pololu (www.pololu.com)
- **RS** RS (www.rs-online.com)
- **SF** SparkFun (www.sparkfun.com)
- **SH** Smarthome (www.smarthome.com)
- **SS** Seeed Studio (www.seeedstudio.com)

Infrastructure

» **Personal computer** Used in all projects.
» **Ethernet connection to the Internet** Used in many projects.
» **Project enclosure** Used in most projects.
» **1/16-inch Mat board** Used in projects: 8, 26, 27 for project enclosures.
» **1 Bluetooth-enabled personal computer** Used in projects: 3, 4, 11, 18, 19. If your laptop doesn't have a Bluetooth radio, use a USB Bluetooth adapter. **SF** WRL-09434, **F** 1814756
» **Android device** Used in project: 31.

Microcontrollers, Shields, and Prototyping Boards

» **Arduino Uno module** Used in projects: 1, 2, 3, 9, 10, 11, 15, 16, 20, 21, 26, 32.
D 1050-1019-ND, **J** 2121105, **SF** DEV-09950, **A** A000046, **AF** 50, **F** 1848687, **RS** 715-4081, **SS** ARD132D2P, **MS** MKSP4
» **Arduino Fio module** Used in project: 10.
SF DEV-10116
» **Arduino Ethernet board** Used in projects: 6, 7, 8, 13, 14, 27, 29. Alternatively, an Uno-compatible board (above) with an Ethernet shield will work.
SF DEV-09026, **J** 2124242, **A** A000056, **AF** 201, **F** 1848680
» **LilyPad Arduino** Used in project: 31.
SF DEV-09266, **A** A000011
» **Arduino WiFi shield** Used in project: 12, but it can be used in any of the Ethernet projects.
A A000058
» **Arduino wireless shield** Used in projects: 10, 13, 14.
A A000064 or A000065. Alternative shields: **SF** WRL-09976, **AF** 126, **F** 1848697, **RS** 696-1670, **SS** WLS114A0P
» **RFID shield** Used in project: 27.
SF DEV-10406, **A** T040030 or T040031
» **Spark Fun Musical Instrument shield** Used in project: 32.
SF DEV-10587
» **Solderless breadboard** Used in projects: 2, 3, 5, 6, 7, 9, 10, 11, 12, 19, 20, 21, 27. Alternatively, you can use prototyping shields for Arduino.
D 438-1045-ND, **J** 20723 or 20601, **SF** PRT-00137, **F** 4692810, **AF** 64, **SS** STR101C2M or STR102C2M, **MS** MKKN2
» **Prototyping shields for Arduino** Used in project: 26. Also an alternative to solderless breadboards in all projects, but you'll still need a tiny breadboard with them.
AF 51, **A** A000024, **SF** DEV-07914, **MS** MSMS01
Breadboards for protoshields: **SF** PRT-08802, **AF** included with board, **D** 923273-ND
» **Perforated printed circuit board** Used in projects: 8, 27.
D V2018-ND, **J** 616673, **SS** STR125C2B, **F** 4903213, **RS** 159-5420

Communications Modules

» **FTDI-style USB-to-Serial adapter** Used in most projects.
SF DEV-09718 or DEV-09716, **AF** 70 or 284, **A** A000059, **MS** MKAD22, **D** TTL-232R-3V3 or TTL-232R-5V
» **Bluetooth Mate module** Used in projects: 3, 4, 11, 18, 19, 31.
SF WRL-09358 or WRL-10393
» **Digi XBee 802.15.4 RF modules** Used in projects: 10, 14, 17.
J 2113375, **SF** WRL-08664, **AF** 128, **F** 1546394, **SS** WLS113A4M, **MS** MKAD14
» **USB-XBee adapter** Used in projects: 10, 14, 17.
J 32400, **SF** WRL-08687, **AF** 247
» **Interface module: X10 One-Way Interface module** Used in project: 26.
SH 1134B
» **X10 modules** Used in project: 26.
SH 2002 or 2000.
» **RFID reader** Used in projects: 25, 26.
CR IDI003 or IDI004, **SF** SEN-08419
» **RFID tags** Used in projects: 25, 26.
CR WON002, **SF** COM-10169
» **1 SonMicro SM130 RFID read/write module** Used in project: 27.
SF SEN-10126
» **Mifare RFID read/write tags** Used in project: 27.
SF SEN-10127

- **13.56MHz antenna** Used in project: 27.
  A C000027
- **SD card reader that can read MicroSD** Available at any local electronics or office-supply store. Used in projects: 29, 30.
  MicroSD card Used in projects: 29, 30.
  IP-based camera Used in projects: 29, 30. D-Link DCS-930L

Breakout Boards and Connectors

- **LilyPad XBee** Used in project: 14.
  SF DEV-08937
- **XBee Explorer Regulated** Used in project: 14.
  SF WRL-09132
- **Gas sensor breakout board** Used in project: 13.
  SF BOB-08891, **P** 1479 or 1639
- **3-wire JST connector pigtail** Used in project: 15.
  SF SEN-08733
- **9V battery clip** Used in projects: 3, 14.
  D 2238K-ND, **J** 101470, **SF** PRT-09518, **F** 1650675
- **Female power plug, 2.1mm ID, 5.5mm OD** Used in project: 3.
  D CP-024A-ND, **J** 159506, **F** 1737256
- **Wire-wrapping wire** Used in project: 5.
  D K445-ND, **J** 22577, **SF** PRT-08031, **F** 150080
- **Wire-wrapping tool** Used in project: 5.
  J 242801, **F** 441089, **RSH** 276-1570, **S** TOL-00068
- **0.1-inch male header pins** Used in projects: 5, 10, 14, 15, 16, 19, 20, 25, 26, 27.
  D A26509-20-ND, **J** 103377, **SF** PRT-0011, **F** 1593411
- **2mm female header rows** Used in projects: 10, 14.
  J 2037747, **D** 3M9406-ND, **F** 1776193
- **2mm 5-pin socket** Used in projects: 25, 26.
  SF PRT-10519
- **2mm 7-pin socket** Used in projects: 25, 26.
  SF PRT-10518
- **6 pin stackable header** Used in project: 27.
  SF PRT-09280, **AF** 85
- **8-conductor wire** Used in project: 27.
  D AE08A-5-ND, **F** 1301013
- **Interface cable for GPS module** Used in project: 19.
  SF GPS-00465, **P** 28146
- **RFID breakout board** Used in projects: 25, 26.
  SF SEN-08423
- **4-wire phone cable with RJ-11 connector**
  Used in project: 26.
  D A1642R-07-ND, **J** 115617, **F** 1335141
- **2-pin Screw Terminal** Used in projects: 29, 30.
  SF PRT-08432, **D** 732-2030-ND, **F** 1792766, **RS** 189-5893
- **3-pin Screw Terminal** Used in projects: 29, 30.
  SF PRT-08235, **D** 732-2031-ND, **F** 1792767, **RS** 710-0166

Common Components

- **100-ohm resistor** Used in project: 8.
  D 100QBK-ND, **J** 690620, **F** 9337660, **RS** 707-8625
- **220-ohm resistor** Used in projects: 5, 8, 9, 11, 20, 26.
  D 220QBK-ND, **J** 690700, **F** 9337792, **RS** 707-8842
- **1-kilohm resistor** Used in projects: 5, 13, 14, 29, 30.
  D 1.0KQBK-ND, **J** 29663, **F** 1735061, **RS** 707-8669
- **10-kilohm resistors** Used in projects: 2, 6, 9, 11, 12, 13, 14, 20.
  D 10KQBK-ND, **J** 29911, **F** 9337687, **RS** 707-8906
- **4.7-kilohm resistor** Used in projects: 14, 27.
  D CF14JT4K70CT-ND, **J** 691024, **F** 735033, **RS** 707-8693
- **10-kilohm potentiometers** Used in projects: 10, 11, 13, 14, 27, 29, 30.
  J 29082, **SF** COM-09939, **F** 350072, **RS** 522-0625
- **100-kilohm resistors** Used in project: 14.
  D 100KQBK-ND, **J** 29997, **F** 9337695, **RS** 707-8940
- **270-kilohm resistor** Used in project: 31.
  J 691446, **D** P270KBACT-ND, **RS** 163-921, **F** 1565367
- **LEDs** Used in projects: 7, 8, 10, 11, 20, 26, 29, 30.
  D 160-1144-ND or 160-1665-ND, **J** 34761 or 94511, **F** 1015878, **RS** 247-1662 or 826-830, **SF** COM-09592 or COM-09590
- **RGB LED, common cathode** Used in project: 1.
  D 754-1492-ND, **J** 2125181, **SF** COM-00105, **F** 8738661, **RS** 713-4888
- **Infrared LED** Used in project: 9.
  J 106526, **SF** COM-09469, **F** 1716710, **RS** 577-538, **SS** MTR102A2B
- **5V regulator** Used in project: 13, 14, 19.
  J 51262, **D** LM7805CT-ND, **SF** COM-00107, **F** 1860277, **RS** 298-8514
- **3.3V regulator** Used in projects: 10, 14.
  J 242115, **D** 576-1134-ND, **SF** COM-00526, **F** 1703357, **RS** 534-3021
- **1µF capacitor** Used in projects: 10, 14.
  J 94161, **D** P10312-ND, **F** 8126933, **RS** 475-9009
- **10µF capacitor** Used in projects: 10, 14, 15.
  J 29891, **D** P11212-ND, **F** 1144605, **RS** 715-1638
- **100µF capacitor** Used in projects: 13, 32.
  J 158394, **D** P10269-ND, **F** 1144642, **RS** 715-1657

- **4700µF electrolytic capacitor** Used in project: 14.
  J 199006, **D** P10237-ND, **F** 1144683, **RS** 711-1526
- **TIP120 Darlington NPN transistor** Used in project: 13.
  D TIP120-ND, **J** 32993, **F** 9804005
- **1N4004 power diode** Used in project: 13.
  D 1N4004-E3 or 23GI-ND, **J** 35992, **F** 9556109, **RS** 628-9029
- **2N3906 PNP-type transistor** Used in projects: 14, 29, 30.
  J 178618, **D** 2N3906D26ZCT-ND, **SF** COM-00522, **F** 1459017, **RS** 294-328
- **9V battery** Used in projects: 3, 14.
- **9–12V DC power supply** Used in project: 13.
  J 170245, **SF** TOL-00298, **AF** 63, **F** 636363, **P** 1463
- **Lithium Polymer ion battery** Used in project: 31.
  SF PRT-00341, **AF** 258, **RS** 615-2472, **F** 1848660
- **USB LiPoly charger** Used in project: 31.
  A 259, **SF** PRT-10217

Specialty Components

- **Voltmeter** Used in project: 7.
  SF TOL-10285, **F** 4692810, **RS** 244-890
- **MAX8212 voltage monitor** Used in project: 14.
  D MAX8212CPA+-ND, **F** 1610130
- **solar cell** Used in project: 14.
  SF PRT-07840, **P** 1691
- **16x2 character LCD** Used in project: 27.
  SF LCD-00709
- **Relay Control PCB** Used in projects: 29, 30.
  SF COM-09096
- **Relay** Used in project: 29, 30.
  SF COM-00101, **D** T9AV1D12-12-ND, **F** 1629059
- **1N4148 diode** Used in projects: 29, 30.
  SF COM-08588, **F** 1081177, **D** 1N4148TACT-ND, **RS** 544-3480
- **PowerSwitch Tail** Alternative part used in projects: 29, 30.
  SF COM-09842, **AF** 268

Sensors

- **Flex Sensor resistors** Used in project: 2.
  D 905-1000-ND, **J** 150551, **SF** SEN-10264, **AF** 182, **RS** 708-1277, **MS** JM150551
- **Momentary switches or pushbuttons** Used in projects: 2, 8, 9, 11.
  D GH1344-ND, **J** 315432, **SF** COM-09337, **F** 1634684, **RS** 718-2213, **MS** JM315432
- **Force-sensing resistors, Interlink 400 series** Used in project: 5.
  D 1027-1000-ND, **J** 2128260, **SF** SEN-09673
- **web camera** Used in project: 5.
- **Photocells (light-dependent resistors)** Used in projects: 6, 12.
  D PDV-P9200-ND, **J** 202403, **SF** SEN-09088, **F** 7482280, **RS** 234-1050
- **2-axis joystick** Used in project: 8.
  J 2082855, **SF** COM-09032, **AF** 245, **F** 1428461
- **Accelerometer** Used in projects: 8, 21.
  J 28017, **SF** SEN-00692, **AF** 163, **RS** 726-3738, **P** 1247, **MS** MKPX7
- **Hanwei gas sensor** Used in project: 13.
  SF SEN-08880, SEN-09404, or SEN-09405, **P** 1480, 1634, 1633, 1481, 1482, or 1483
- **Sharp GP2Y0A21 infrared ranger** Used in project: 15.
  D 425-2063-ND, **SF** SEN-00242, **RS** 666-6570, **P** 136
- **MaxBotix LV-EZ1 ultrasonic ranger** Used in projects: 16, 32.
  SF SEN-00639, **AF** 172, **P** 726, **SS** SEN136B5B
- **EM-406A GPS receiver module** Used in project: 19.
  SF GPS-00465 **P** 28146 **AF** 99
- **ST Microelectronics LSM303DLH digital compass** Used in project: 20.
  SF SEN-09810, **RS** 717-3723, **P** 1250
- **LED tactile button** Used in project: 20.
  SF COM-10443 and BOB-10467
- **Temperature sensor** Used in projects: 29, 30.
  AF 165, **D** TMP36GT9Z-ND, **F** 1438760, **RS** 427-351

Miscellaneous

- **Ping-pong ball** Used in project: 1.
- **Small pink monkey** Used in projects: 2, 3.
- **cat** Used in projects: 5, 29, 30.
- **cat mat** Used in project: 5.
- **thick pieces of wood or thick cardboard, about the size of the cat mat** Used in project: 5
- **Lighting filters** Used in projects: 6, 12.
- **Triple-wall cardboard** Used in project: 8.
- **Velcro** Used in projects: 8, 31.
- **Cymbal monkey** Used in project: 13.
- **Conductive ribbon** Used in project: 31.
  SF DEV-10172
- **Conductive thread** Used in project: 31.
  SF DEV-10120, **L** A304
- **Shieldit Super Conductive Fabric** Used in project: 31.
  L A1220-14
- **Hoodie** Used in project: 31.
- **Embroidery thread** Used in project: 31.

Hardware

This list includes all vendors for current and past editions.

KEY

≈ Phone / * Toll free
Δ Fax
Mailing address

Abacom Technologies

Abacom sells a range of RF transmitters, receivers, and transceivers, and serial-to-Ethernet modules.
www.www.abacom-tech.com
email: abacom@abacom-tech.com
3210 Wharton Way
Mississauga ON L4X 2C1, Canada

Aboyd Company

The Aboyd Company sells art supplies, costumes, novelties, cardboard standups, home décor, and more. They're also a good source of Charley Chimp cymbal-playing monkeys.
www.aboyd.com
email: info@aboyd.com
≈ +1-888-458-2693
≈ +1-601-948-3477 *International*
Δ +1-601-948-3479
P.O. Box 4568
Jackson, MS 39296, USA

Acroname Robotics

Acroname sells a wide variety of sensors and actuators for robotics and electronics projects. They've got an excellent range of esoteric sensors like UV-flame sensors, cameras, and thermal-array sensors. They've got a lot of basic distance rangers as well. They also have a number of good tutorials on their site on how to use their parts.
www.acroname.com
email: info@acroname.com
≈ +1-720-564-0373
Δ +1-720-564-0376.
4822 Sterling Dr.
Boulder, CO 80301-2350, USA

Adafruit Industries

Adafruit makes a number of useful open source DIY electronics kits, including an AVR programmer, an MP3 player, and more.
www.adafruit.com
email: sales@adafruit.com

Arduino Store

The Arduino Store sells Arduino microcontroller boards and shields, as well as TinkerKit parts and accessories for Arduino selected by the Arduino team. It includes some of the Arduino team's favorite third-party accessories.
http://store.arduino.cc/ww/
GHEO SA
via soldini, 22
CH-6830 Chiasso, Switzerland

Atmel

Atmel makes the AVR microcontrollers that are at the heart of the Arduino, Wiring, and BX-24 modules. They also make the ARM microcontroller that runs the Make controller.
www.atmel.com
≈ +1-408-441-0311
2325 Orchard Parkway
San Jose, CA 95131, USA

CoreRFID

CoreRFID sells a variety of RFID readers, tags, and other RFID products.
www.rfidshop.com
e-mail: info@corerfid.com
≈ +44 (0) 845-071-0985
+44 (0) 845-071-0989
Dallam Court
Dallam Lane
Warrington, WA2 7LT, United Kingdom

D-Link

D-link makes a number of USB, Ethernet, and WiFi products, including the D-link IP-based WiFi camera used in Chapter 10.
www.dlink.com
e-mail: productinfo@dlink.com
≈ +1-800-326-1688

Devantech/Robot Electronics

Devantech makes ultrasonic ranger sensors, electronic compasses, LCD displays, motor drivers, relay controllers, and other useful add-ons for microcontroller projects.
http://robot-electronics.co.uk
e-mail: sales@robot-electronics.co.uk
≈ +44 (0)195-345-7387
Δ +44 (0)195-345-9793
Maurice Gaymer Road
Attleborough
Norfolk, NR17 2QZ, England

Digi

Digi makes XBee radios, radio modems, and Ethernet bridges.
www.digi.com
≈ +1-877-912-3444
Δ +1-952-912-4952
11001 Bren Road East
Minnetonka, MN 55343, USA

Digi-Key Electronics

Digi-Key is one of the U.S.'s largest retailers of electronics components. They're a staple source for things you use all the time—resistors, capacitors, connectors, some sensors, breadboards, wire, solder, and more.
www.digikey.com
≈ +1-800-344-4539 or
≈ +1-218-681-6674
Δ +1-218-681-3380
701 Brooks Avenue South
Thief River Falls, MN 56701, USA

ELFA

ELFA is one of Northern Europe's largest electronics components suppliers.
www.elfa.se
email: export@elfa.se
≈ +46 8-580-941-30
S-175 80 Järfälla, Sweden

Farnell

Farnell supplies electronics components for all of Europe. Their catalog part numbers are consistent with Newark in the U.S., so if you're working on both sides of the Atlantic, sourcing Farnell parts can be convenient.
http://uk.farnell.com
email: sales@farnell.co.uk
≈ +44-8701-200-200
Δ +44-8701-200-201
Canal Road,
Leeds, LS12 2TU, United Kingdom

Figaro USA, Inc.

Figaro Sensor sells a range of gas sensors, including volatile organic-compound sensors, carbon-monoxide sensors, oxygen sensors, and more.
www.figarosensor.com
email: figarousa@figarosensor.com
≈ +1-847-832-1701
Δ +1-847-832-1705
3703 West Lake Ave., Suite 203
Glenview, IL 60026, USA

Future Technology Devices International, Ltd. +(FTDI)

FTDI makes a range of USB-to-Serial adapter chips, including the FT232RL that's on many of the modules in this book.
www.ftdichip.com
email: admin1@ftdichip.com
≈ +44 (0) 141-429-2777
373 Scotland Street
Glasgow, G5 8QB, United Kingdom

Glolab

Glolab makes a range of electronic kits and modules, including several useful RF and IR transmitters, receivers, and transceivers.
www.glolab.com
email: lab@glolab.com

Gridconnect

Gridconnect distributes networking products, including those from Lantronix and Digi.
www.gridconnect.com
email: sales@gridconnect.com
≈ +1 630-245-1445
Δ +1 630-245-1717
≈* +1-800-975-GRID (4743)
1630 W. Diehl Road
Naperville, IL 60563, USA

Images SI, Inc.

Images SI sells robotics and electronics parts. They carry a range of RFID parts, force-sensing resistors, stretch sensors, gas sensors, electronic kits, speech-recognition kits, solar energy parts, and microcontrollers.
www.imagesco.com
email: imagesco@verizon.net
≈ +1-718-966-3694
Δ +1-718-966-3695
109 Woods of Arden Road
Staten Island, NY 10312, USA

Interlink Electronics

Interlink makes force-sensing resistors, touchpads, and other input devices.
www.interlinkelectronics.com
email: specialty@interlinkelectronics.com
≈ +1-805-484-8855
Δ +1-805-484-8989
546 Flynn Road
Camarillo, CA 93012, USA

IOGear

IOGear make computer adapters. Their USB-to-Serial adapters are good, and they carry Powerline Ethernet products.
www.iogear.com
email: sales@iogear.com
≈* +1-866-946-4327
≈ +1-949-453-8782
Δ +1-949-453-8785
23 Hubble Drive
Irvine, CA 92618, USA

Jameco Electronics

Jameco carries bulk and individual electronics components, cables, breadboards, tools, and other staples for the electronics hobbyist or professional.
http://jameco.com

email: domestic@jameco.com
international@jameco.com
custservice@jameco.com
≈ +1-800-831-4242
Toll-free 24-hour order line
≈ +1-650-592-8097
International order line
Δ +1-650-592-2503 *International*
Δ +1-800-237-6948* *Toll-free fax*
Δ +001-800-593-1449*
Mexico toll-free fax
Δ +1-803-015-237-6948*
Indonesia toll-free fax
1355 Shoreway Road
Belmont, CA 94002, USA

Keyspan

Keyspan makes computer adapters. Their USA-19*xx* series of USB-to-Serial adapters are very handy for microcontroller work.
www.keyspan.com
email: info@keyspan.com
≈ +1-510-222-0131 *Info/sales*
≈ +1-510-222-8802 *Support*
Δ +1-510-222-0323
4118 Lakeside Dr
Richmond, CA 94806, USA

Lantronix

Lantronix makes serial-to-Ethernet modules: the XPort, the WiPort, the WiMicro, the Micro, and many others.

www.lantronix.com
email: sales@lantronix.com
≈ +1-800-526-8766
≈ +1-949-453-3990
Δ +1-949-450-7249
5353 Barranca Parkway
Irvine, CA 92618, USA

Libelium

Libelium makes an XBee-based product and other wireless products.

www.libelium.com
email: info@libelium.com
Libelium Comunicaciones Distribuidas S.L.
Maria de Luna 11, Instalaciones
CEEIARAGON, C.P: 50018
Zaragoza, Spain

Linx Technologies

Linx makes a number of RF receivers, transmitters, and transceivers.

www.linxtechnologies.com
email: info@linxtechnologies.com
≈ +1-800-736-6677 *U.S.*
≈ +1-541-471-6256 *International*
Δ +1-541-471-6251
159 Ort Lane
Merlin, OR 97532, USA

Low Power Radio Solutions

LPRS makes a number of RF receivers, transmitters, and transceivers.

www.lprs.co.uk
email: info@lprs.co.uk
≈ +44-1993-709418
Δ +44-1993-708575
Two Rivers Industrial Estate
Station Lane, Witney
Oxon, OX28 4BH, United Kingdom

Maker SHED

Launched originally as a source for back issues of *MAKE Magazine*, the Maker SHED now has a variety of stuff for makers, crafters, and budding scientists.

www.makershed.com
email: help@makershed.com
≈+1-800-889-8969
1005 Gravenstein Hwy N
Sebastopol, CA 95472, USA

Making Things

Making Things makes the MAKE controller, and originated the now-discontinued Teleo controllers. They do custom hardware-engineering solutions.

www.makingthings.com
email: info@makingthings.com
Δ +1-415-255-9513
1020 Mariposa Street, #2
San Francisco, CA 94110, USA

Maxim Integrated Products

Maxim makes sensors, communications chips, power-management chips, and more. They also own Dallas Semiconductor. Together, they're one of the major sources for chips related to serial communication, temperature sensors, LCD control, and much more.

www.maxim-ic.com
email: info2@maxim-ic.com
≈ +1-408-737-7600
Δ +1-408-737-7194
120 San Gabriel Drive
Sunnyvale, CA 94086, USA

Microchip

Microchip makes the PIC family of microcontrollers. They have a very wide range of microcontrollers, for just about every conceivable purpose.

www.microchip.com
≈ +1-480-792-7200
2355 West Chandler Blvd.
Chandler, AZ, 85224-6199, USA

Mouser

Mouser is a large retailer of electronic components in the U.S. They stock most of the staple parts used in the projects in this book, such as resistors, capacitors, and some sensors. They also carry the FTDI USB-to-Serial cable.

www.mouser.com
email: help@mouser.com
1000 North Main Street
Mansfield, TX 76063, USA

NetMedia

NetMedia makes the BX-24 microcontroller module and the SitePlayer Ethernet module.

www.basicx.com
siteplayer.com
email: sales@netmedia.com
≈ +1-520-544-4567
Δ +1-520-544-0800
10940 N. Stallard Place
Tucson, AZ 85737, USA

Newark In One Electronics

Newark supplies electronics components in the U.S. Their catalog part numbers are consistent with Farnell in Europe, so if you're working on both sides of the Atlantic, sourcing parts from Farnell and Newark can be convenient.

www.newark.com
email: somewhere@something.com
≈ +1-773-784-5100
Δ +1-888-551-4801
4801 N. Ravenswood
Chicago, IL 60640-4496, USA

New Micros

New Micros sells a number of microcontroller modules. They also sell a USB-XBee dongle that allows you to connect Digi XBee radios to a computer really easily. Their dongles also have all the necessary pins connected for reflashing the XBee's firmware serially.
www.newmicros.com
email: nmisales@newmicros.com
≈ +1-214-339-2204

Parallax

Parallax makes the Basic Stamp family of microcontrollers. They also make the Propeller microcontroller, and a wide range of sensors, beginners' kits, robots, and other useful tools for people interested in electronics and microcontroller projects.
www.parallax.com
email: sales@parallax.com
≈* +1-888-512-1024 *Toll-free sales*
≈ +1-916-624-8333
Office/international
Δ +1-916-624-8003
599 Menlo Drive
Rocklin, California 95765, USA

Phidgets

Phidgets makes input and output modules that connect desktop and laptop computers to the physical world.
www.phidgets.com
email: sales@phidgets.com
≈ +1-403-282-7335
Δ +1-402-282-7332
2715A 16A NW
Calgary, Alberta T2M3R7, Canada

Pololu

Pololu makes a variety of electronic components and breakout boards for robotics and other projects.
www.pololu.com
email: www@pololu.com
≈ +1-702-262-6648
≈ +1-877-776-5658 U.S. only
Δ + 1-702-262-6894
3095 E. Patrick Ln. #12
Las Vegas, NV 89120, USA

RadioShack

Hooray! RadioShack has begun to realizedthat the DIY electronics market in the U.S. never went away, and they're carrying more parts again. By the time you read this, maybe they'll be carrying Arduinos as well. Check the website for part numbers, and call your local store first to see whether they've got what you need. It'll save you time.
www.radioshack.com

Reynolds Electronics

Reynolds Electronics makes a number of small kits and modules for RF and infrared communications, IR remote control, and other useful add-on functions for microcontroller projects.
www.rentron.com
email: sales@rentron.com
≈ +1-772-589-8510
Δ +1-772-589-8620
12300 Highway A1A
Vero Beach, Florida, 32963, USA

Roving Networks

Roving Networks makes and sells Bluetooth radio modules for electronics manufacturers. Their radios are at the heart of Spark Fun's Bluetooth Mate modules.
www.rovingnetworks.com
email: info@rovingnetworks.com
≈ +1-408-395-6539
Δ +1-603-843-7550
102 Cooper Court
Los Gatos, CA 95032, USA

RS Online

RS Online is one of Europe's largest electronics retailers. They sell worldwide.
www. rs-online.com
email: general@rs-components.com
≈ 0845-850-9900
Δ 01536-405678

Samtec

Samtec makes electronic connectors. They have a very wide range of connectors, so if you're looking for something odd, they probably make it.
www.samtec.com
email: info@samtec.com
≈ +1-800-SAMTEC-9

Seeed Studio

Seeed Studio makes a number of useful and inventive open source electronics parts.
www.seeedstudio.com
email: techsupport@seeedstudio.com
≈ +86-755-26407752
Room 0728, Bld 5,
Dong Hua Yuan,
NanHai Ave. NanShan dist.
Shenzhen 518054 China

SkyeTek

SkyeTek makes RFID readers, writers, and antennas.

www.skyetek.com

≈ +1-720-565-0441

Δ +1-720-565-8989

11030 Circle Point Road, Suite 300
Westminster, CO 80020, USA

Smarthome

Smarthome makes a wide variety of home-automation devices, including cameras, appliance controllers, X10, and INSTEON.

www.smarthome.com

email: custsvc@smarthome.com

≈ 1-800-762-7846

≈ + 1-800-871-5719 *Canada*

≈ +1-949-221-9200 *International*

16542 Millikan Avenue
Irvine, CA 92606, USA

Spark Fun Electronics

Spark Fun makes it easier to use all kinds of electronic components. They make breakout boards for sensors, radios, and power regulators, and they sell a variety of microcontroller platforms.

www.sparkfun.com

email: spark@sparkfun.com

2500 Central Avenue, Suite Q
Boulder, CO 80301, USA

Symmetry Electronics

Symmetry sells ZigBee and Bluetooth radios, serial-to-Ethernet modules, WiFi modules, cellular modems, and other electronic communications devices.

www.semiconductorstore.com

≈ +1-877-466-9722

≈ +1-310-643-3470 *International*

Δ +1-310-297-9719

5400 West Rosecrans Avenue
Hawthorne, CA 90250, USA

TI-RFID

TIRIS is Texas Instruments' RFID division. They make tags and readers for RFID in many bandwidths and protocols.

www.tiris.com

≈ +1-800-962-RFID (7343)

Δ +1-214-567-RFID (7343)

Radio Frequency Identification Systems
6550 Chase Oaks Blvd., MS 8470
Plano, TX 75023, USA

Trossen Robotics

Trossen Robotics sells a range of RFID supplies and robotics. They have a number of good sensors, including Interlink force-sensing resistors, linear actuators, Phidgets kits, RFID readers, and tags for most RFID ranges.

www.trossenrobotics.com

email: jenniej@trossenrobotics.com

≈ +1-877-898-1005

Δ +1-708-531-1614

1 Westbrook Co. Center, Suite 910
Westchester, IL 60154, USA

Uncommon Projects

Uncommon Projects makes the YBox, a text-overlay device that puts text from web feeds on your TV.

www.uncommonprojects.com

ybox.tv

email: info@uncommonprojects.com

68 Jay Street #206
Brooklyn, NY 11201, USA

Software

Most of the software listed in this book is open source. In the following listings, anything that's not open source is noted explicitly as a commercial application. If there's no note, you can assume it's open.

Arduino
Arduino is a programming environment for AVR microcontrollers. It's based on Processing's programming interface. It runs on Mac OS X, Linux, and Windows operating systems.
www.arduino.cc

Asterisk
Asterisk is a software private branch exchange (PBX) manager for telephony. It runs on Linux and Unix operating systems.
www.asterisk.org

AVRlib
AVRlib is a library of C functions for a variety of tasks using AVR processors. It runs on Mac OS X, Linux, and Windows operating systems as a library for the avr-gcc compiler.
hubbard.engr.scu.edu/avr/avrlib

avr-gcc
The GNU avr-gcc is a C compiler and assembler for AVR microcontrollers. It runs on Mac OS X, Linux, and Windows operating systems.
www.avrfreaks.net/AVRGCC

CCS C
CCS C is a commercial C compiler for the PIC microcontroller. It runs on Windows and Linux operating systems.
www.ccsinfo.com

CoolTerm
CoolTerm is a freeware (though not open source) serial terminal application for Mac OS X and Windows written by Roger Meier.
http://freeware.the-meiers.org

Dave's Telnet
Dave's Telnet is a telnet application for Windows.
http://dtelnet.sourceforge.net

Eclipse
Eclipse is an integrated development environment (IDE) for programming in many different languages. It's extensible through a plug-in architecture, and there are compiler links to most major programming languages. It runs on Mac OS X, Linux, and Windows.
www.eclipse.org

Evocam
Evocam is a commercial webcam application for Mac OS X.
http://evological.com

Exemplar
Exemplar is a tool for authoring sensor applications through behavior rather than through programming. It runs on Mac OS X, Linux, and Windows operating systems as a plug-in for Eclipse.
http://hci.stanford.edu/research/exemplar

Fwink
Fwink is a webcam application for Windows.
www.lundie.ca/fwink

Girder
Girder is a commercial home automation application for Windows.
www.girder.nl

GitHub
GitHub is a host for git, a version-control tool for programming source code of any language. Git and github are great tools to share your code.
http://git-scm.com
https://github.com

Java
Java is a programming language. It runs on Mac OS X, Linux, and Windows operating systems, and many embedded systems as well.
http://java.sun.com

Macam
Macam is a webcam driver for Mac OS X.
http://webcam-osx.sourceforge.net/

Max/MSP
Max is a commercial graphic data-flow authoring tool. It allows you to program by connecting graphic objects rather than writing text. Connected with Max are MSP, a real-time audio-signal processing library, and Jitter, a real-time video-signal processing library. It runs on Mac OS X and Windows operating systems.
www.cycling74.com

PEAR
PEAR is the PHP Extension and Application Repository. It hosts extension libraries for PHP scripting.
http://pear.php.net

PHP
PHP is a scripting language that is especially suited for web development and can be embedded into HTML. It runs on Mac OS X, Linux, and Windows operating systems.
www.php.net

PicBasic Pro
PicBasic Pro is a commercial BASIC compiler for PIC microcontrollers. It runs on Windows.
http://melabs.com

Processing
Processing is a programming language and environment designed for the nontechnical user who wants to program images, animation, and interaction. It runs on Mac OS X, Linux, and Windows.
www.processing.org

Puredata (PD)
Puredata (PD) is a graphic data-flow authoring tool. It allows you to program by connecting graphic objects rather than writing text. It's developed by one of the original developers of Max, Miller Puckette. It runs on Mac OS X, Linux, and Windows operating systems.
http://puredata.info

PuTTY SSH
PuTTY is a telnet/SSH/serial port client for Windows.
www.puttyssh.org

QR Code Library
QR Code Library is a set of libraries for encoding and decoding QR Code 2D barcodes. It runs on Mac OS X, Linux, and Windows as a library for Java.
http://qrcode.sourceforge.jp

Dan Shiffman's Processing Libraries
Dan Shiffman has written a number of useful libraries for Processing, including the pqrcode library used in this book (www.shiffman.net/p5/pqrcode). He's also got an SFTP library (www.shiffman.net/2007/06/04/sftp-with-java-processing) and a sudden-motion sensor library for Mac OS X (www.shiffman.net/2006/10/28/processingsms).

Sketchtools NADA
NADA is a proxy tool for connecting programming environments with hardware devices. Originally a commercial tool, it's since been open sourced.
code.google.com/p/nadamobile

TinkerProxy
TinkerProxy is a TCP-to-Serial proxy application.
code.google.com/p/tinkerit/wiki/TinkerProxy

Twilio
Twilio is a commercial IP telephony provider. They provide application programming interfaces that allow you to connect telephone calls to web applications.
www.twilio.com

UDP Library for Processing
Hypermedia's UDP library for Processing enables you to communicate via UDP from Processing. It runs on Mac OS X, Linux, and Windows as a library for Processing.
hypermedia.loeil.org/processing

Wiring
Wiring is a programming environment for AVR microcontrollers. It's based on Processing's programming interface. It runs on Mac OS X, Linux, and Windows operating systems.
www.wiring.org.co

Index

Symbols

$$$ command, 69, 194
$ symbol, 17
& (AND) logical operator, 423
< character, 85
> character, 85
+++ command, 68, 194, 230
| (OR) logical operator, 423
<< (shift left operator), 423
>> (shift right operator), 423
~ symbol, 13
^ (XOR) logical operator, 424

A

Abacom Technologies, 447
Aboyd Company, 447
absolute path, 13
accelerometers
  measuring rotations, 290, 292
  PROJECT 21: Determining Attitude Using an Accelerometer, 263, 290–298
ACN (Advanced Controller Networking), 431, 435
acquisition
  defined, 265
  distance ranging and, 267
Acroname Easier Robotics, 447
active distance ranging, 267, 272
active RFID systems, 315
Adafruit Industries
  about, 447
  accelerometers, 290
  Ethernet Shield, 119
  SD card breakout board, 376
  XBee Adapter Kit, 197
  XBee USB Adapter Board, 193
adapter boards, 39
Adaptive Design Association, 163–164
addressing schemes, 71, 79–81
ADH Tech, 395
Adobe Illustrator, 25
Advanced Controller Networking (ACN), 431, 435
AIM (AOL instant messenger), 151
AIRNow web page, 127–139
Air Quality Meter, Networked (project 7)
  project overview, 127–139
  supplies for, 117
Allan, Alasdair, 396
alligator clip test leads, 7
American Standards Association, 54
American Symbolic Code for Information Interchange. *See* ASCII character set
analog input circuits
  about, 30
  usage example, 33
analog radio transmission, 190
analogRead() method (Arduino), 268, 293
analog-to-digital converters, 238
analogWrite() method (Arduino), 48
AND (&) logical operator, 423
Andraos, Mouna, 261, 298
Android devices
  PROJECT 31: Personal Mobile Datalogger, 365, 401–414
  setting up Processing for, 396–400
  USB and, 414
anodes, defined, 46
antenna design, 190
AOL instant messenger (AIM, 151
Apache
  MOD_REWRITE module, 437
  PHP scripts and, 15
API (application programming interface), 387
App Inventor environment, 396
appliance control modules, 322
application layer
  about, 40
  TTL serial protocol, 42
  understanding by building a project, 46–49
  USB protocol, 42
application programming interface (API), 387
Arduino modules
  about, 20–21, 452
  changes to version 1.0, 27
  Ethernet library, 120–122
  flow control and, 62–63
  inputs and outputs for, 24, 25
  installation process, 24–26
  LEDs on, 27–28
  listening for incoming serial data, 29
  programming environment depicted, 27
  Serial Monitor, 31, 53, 230
  serial ports, 29
  shields for, 22
  solderless breadboards and, 30, 31
  USB-to-serial adapters and, 45, 119
  variations of, 21, 32
Arduino playground, 187
Arduino store site, 447
Area/Code site, 312
Arnall, Timo, 301, 316
ArrayList data type, 154
ASCII character set
  about, 54–55
  carriage return, 123, 194
  Hayes AT command protocol, 68
  linefeed, 123
  PHP support, 86
associative arrays, 432
Asterisk telephony server, 366, 386, 452
asynchronous serial communication
  about, 40, 41, 421
  Bluetooth and, 64
  flow control and, 62
ATCN command, 204
ATDH command, 196
ATDL command, 246
ATIS command, 235
Atmel microcontrollers, 23, 447
ATMY command, 231, 246
ATND command, 230
ATRE command, 234
attitude
  defined, 290
  PROJECT 21: Determining Attitude Using an Accelerometer, 263, 290–298
attributes, defined, 388
ATVR command, 231
available() function (Serial library), 121
avr-gcc, 452
AVRlib library, 452

B

Bağdatlı, Mustafa, 401, 402
balance boards, 163–166
Banzi, Massimo, 21
Barcia-Colombo, Gabriel, 72
bar code recognition
  about, 312
  PROJECT 24: 2D Bar Code Recognition Using Webcam, 302, 313–315
  scanned images and, 315
batteries and battery snap adapters
  common components, 8
  PROJECT 3: Wireless Monski Pong, 66
  purchasing, 7
baudrate, 69
Beagle Board, 118
begin() method
  Ethernet library, 139
  SD library, 375
beginPacket() method (UDP library), 229
Beim, Alex, 181
binary protocols, 422–424, 431
biometric tracking, 401–414
Bishop, Durrell, 309
bits
  masking, 423–424
  reading and writing, 423
  shifting, 423
bitwise operators, 423–424
blink() method (Processing), 135, 203, 212
Bluetooth protocol
  about, 64
  802.15.4 and, 226
  PROJECT 3: Wireless Monski Pong, 64–67
  PROJECT 4: Negotiating in Bluetooth, 39, 68–71
  PROJECT 11: Bluetooth Transceivers, 183, 206–215
  PROJECT 18: Reading Received Signal Strength Using Bluetooth Radios, 263, 276
  PROJECT 19: Reading GPS Serial Protocol, 278–285
  PROJECT 31: Personal Mobile Datalogger, 365, 401–414
  radio signal strength, 275
boolean data type, 11
Boxall, John, 395
breadboards, solderless. *See* solderless breadboards
breakout boards
  used in projects, 445
  XBee, 197, 200
Brevig, Alexander, 288
broadcast messages
  defined, 193
  querying for devices using UDP, 227–230
  querying for XBee radios using 802.15.4, 230–231
  UDP considerations, 226
BSD environment, 11
BtSerial library, 407, 410
Buechley, Leah, 402
Button library, 288
byte data type, 11

C

C language, 11
calibrating compasses, 287
call-and-response technique, 63
cameras
  face recognition algorithms, 309
  making infrared visible, 186
  network, 384–385
  PROJECT 9: Infrared Control of a Digital Camera, 182, 188–189
Canonical name (CNAME) record, 383
capacitors
  common components, 8
  purchasing, 7
captive portals, 219
carriage return character, 123, 194
carrier waves, 185
cascade files, 310
cascade() method (OpenCV library), 310
catcam
  PROJECT 5: Networked Cat, 99–100, 110–111
  PROJECT 29: Catcam Redux, 364, 369–383
cathode, defined, 46
CCS C compiler, 452
cd command, 13
chat servers
  defined, 151
  PROJECT 8: Networked Pong, 150, 153–155
Chip Select (CS) pin, 119
chmod command, 14
circuits
  about, 366
  analog input, 30, 33
  digital input, 30
classes
  constructor methods and, 168
  defined, 166
  instances of, 168
  instance variables and, 168
clearing the bit, 423
Clear to Send (CTS, 45
Client library
  connected() method, 121
  connect() method, 138, 142
client-server model
  defined, 82
  email and, 88
  PROJECT 5: Networked Cat, 105
  PROJECT 8: Networked Pong, 150, 155–177
  PROJECT 29: Catcam Redux, 364, 369–383
  web browsing and, 82–86
  writing test programs, 143–146
Clock pin, 119
close() function (SD library), 376
CNAME (Canonical name) record, 383
Cohen, Jonathan, 25, 309
color recognition
  about, 305
  answering machines and, 309
  lighting tricks, 308
  PROJECT 22: Color Recognition Using a Webcam, 302, 306–309
command-line interface
  about, 11–12
  controlling file access, 14
  creating files, 14
  deleting files, 15
  PHP scripts and, 15–17
  serial communication tools, 17–20
  troubleshooting messages, 81
  usage overview, 13–14
  viewing files, 14

command mode
about, 68
Hayes AT command protocol and, 68
PROJECT 4: Negotiating in Bluetooth, 69
switching modes, 68, 194
comma-separated values (CSV), 432
communications modules, 444–445
communications protocols. *See* protocols
compasses. *See* digital compasses
computers. *See* personal computers
conditional statements
about, 11
debugging, 141
if() statement, 61, 385
if-then statements, 11
while() statement, 379–380
connected() method (Client library), 121
connect() method
BtSerial library, 408, 410
Client library, 138, 142
constructor methods, 168
control connections, 31
Control panel modules, 322
converters, serial-to-USB, 7
Cooking Hacks site, 395
CoolTerm program
about, 18, 452
PROJECT 10: Duplex Radio Transmission, 194, 196
CoreRFID, 447
Cousot, Stephane, 227
Crites, Tom, 321
CS (Chip Select) pin, 119
CSS3 standard, 391
CSV (comma-separated values), 432
CTS (Clear to Send), 45
Culkin, Jody, 25

D

data formats, 432–434
datagrams
defined, 226
directed, 246
data layer
about, 40
RS-232 serial protocol, 43
TTL serial protocol, 42
USB protocol, 42
datalogging (project 31), 365, 401–414
data mode
about, 68
Hayes AT command protocol and, 68
PROJECT 4: Negotiating in Bluetooth, 69
switching modes, 68
data types, 11
date() function (PHP), 17
Dave's Telnet, 452
dBm (decibel-milliwatts), 275
DDNS (Dynamic DNS), 383
debounce technique, 162
debugging methods, 140–142
decibel-milliwatts (dBm), 275
decimal notation, 80
decodeImage() method (pqrcode library), 314
DELETE command (HTTP), 86
deleting files, 15
delimiters, defined, 56
desoldering pumps, 6
detect() method (OpenCV library), 311
Devantech/Robot Electronics, 447
Device Manager (Windows), 18, 26
DHCP (Dynamic Host Control Protocol), 81, 138
diagnostic tools
about, 140–146
control panel modules, 322
SMRFID Mifare v1.2, 336
Wi-Fi shield, 219
diagonal cutters, 6
Digi, 447
Digi-Key Electronics, 447
digital cameras
face recognition algorithms, 309
making infrared visible, 186
PROJECT 9: Infrared Control of a Digital Camera, 182, 188–189
digital compasses
calibrating, 287
PROJECT 20: Determining Heading Using a Digital Compass, 263, 286–289
digital input circuit, 30
digital radio transmission, 190
diodes, common components, 8
directed messages
about, 246–247
PROJECT 14: Relaying Solar Cell Data Wirelessly, 248–257
directionality
defined, 185
infrared, 186
directly connected networks, 77
distance ranging
about, 267
active, 267, 272
multipath effect, 276
passive, 267, 272
products supporting, 267
PROJECT 15: Infrared Distance Ranger Example, 262, 268–269
PROJECT 16: Ultrasonic Distance Ranger Example, 263, 270–271
PROJECT 17: Received Signal Strength Using XBee Radios, 263, 273–275
PROJECT 18: Reading Received Signal Strength Using Bluetooth Radios, 263, 276
RFID technology and, 316
triangulation, 277
trilateration, 267, 277
dist() function (Processing), 307
D-link, 447
DMX512 protocol, 421, 431
DNS (Domain Name System), 81, 138
dnServerIP() method (Ethernet library), 138
documentation, 25
Domain Name System (DNS), 81, 138
drawing tools, 25
draw() method (Processing), 11, 20
DSO Nano oscilloscope, 34
Duplex Radio Transmission (project 10)
communication between microcontrollers, 204
configuring XBee modules serially, 193–199
programming microcontrollers to user XBee module, 200–204
project overview, 193
supplies for, 182
duty cycle, defined, 235

Dynamic DNS (DDNS), 383
Dynamic Host Control Protocol (DHCP), 81, 138

E

EAN bar code symbology, 312
Echo Mode, 69
Eclipse IDE, 452
EEPROM library, 374
8-bit controllers, 23
802.15.4 standard
  Bluetooth and, 226
  querying for XBee radios, 230–231
electrical interfaces, 2
electrical layer
  about, 40
  RS-232 serial protocol, 43
  TTL serial protocol, 42
  USB protocol, 42
elements, defined, 387
ELFA, 448
email programs
  about, 88
  mobile phone support, 368
  PROJECT 5: Networked Cat, 95–99
  text messaging and, 393–394
embedded modules
  PROJECT 7: Networked Air Quality Meter, 117, 127–139
  troubleshooting, 140–146
end-of-transmission (ETX) byte, 318
endPacket() method (UDP library), 229, 253
Environmental Protection Agency, 127
environment variables
  HTTP support, 353–354
  mail, 357–359
  name-value pairs and, 432
  PHP support, 17
EPanorama site, 186
Ethernet cables, 7
Ethernet library
  about, 120–122
  begin() method, 139
  dnsServerIP() method, 138
  gatewayIP() method, 138
  localIP() method, 138
  PROJECT 6: Hello Internet! Daylight Color Web Server, 122
  PROJECT 13: Reporting Toxic Chemicals in the Shop, 243
  subnetMask() method, 138
Ethernet protocol, 79
Ethernet shield
  about, 22
  Adafruit, 119
  Arduino, 118
  debugging methods, 140
  PROJECT 6: Hello Internet! Daylight Color Web Server, 120–126
  PROJECT 7: Networked Air Quality Meter, 127–139
  PROJECT 13: Reporting Toxic Chemicals in the Shop, 239
  PROJECT 27: Tweets from RFID, 343
ETX (end-of-transmission) byte, 318
events
  about, 96
  PROJECT 5: Networked Cat, 96
Evocam webcam application, 452
Exemplar tool, 452
exists() method (SD library), 376
eXtensible Markup Language (XML), 387–391

F

face detection
  about, 309
  PROJECT 23: Face Detection Using a Webcam, 302, 310–311
falling edge of the clock, 289
Faludi, Robert, 195, 248, 249
Fan, Doria, 75
Faraday cage, 191
Faraday, Michael, 191
Farnell (vendor), 448
feedback loops
  interactive systems and, 151
  PROJECT 8: Networked Pong, 150, 153–177
fgetss() function (PHP), 131
Figaro USA, Inc., 448
file formats, 376
files
  controlling access to, 14
  creating, 14
  deleting, 15
  invisible, 13
  lock, 18
  uploading to servers, 101–102
  viewing, 14
$_FILES variable (PHP), 101
firmware
  reading, 335, 345
  SonMicro readers and, 333–334, 336
  upgrading on XBee radios, 231
float data types
  about, 11
  PROJECT 2: Monski Pong, 57
flow control
  animation and, 62–63
  PROJECT 2: Monski Pong, 50–61
  PROJECT 3: Wireless Monski Pong, 64–67
flush() function (SD library), 376
F() notation, 346
fopen() function (PHP), 132
force-sensing resistors (FSRs)
  about, 89
  common components, 8
  PROJECT 5: Networked Cat, 89–98
foreach() statement, 439
for-next loops, 11
Freescale site, 292
frequency division multiplexing, 192
Fritzing tool, 25
Fry, Ben, 11
FSRs (force-sensing resistors)
  about, 89
  common components, 8
  PROJECT 5: Networked Cat, 89–98
FTDI (Future Technology Devices International, Ltd.), 448
FTDI USB-to-serial adapter, 119
FTDI USB-to-TTL serial cable, 39, 43, 45
Fun with MIDI (project 32)
  project overview, 427–430
  supplies for, 418
Fun with REST (project 33), 437–439
Future Technology Devices International, Ltd. (FTDI), 448
Fwink webcam application, 452

G

Galvanic skin response, 401–414
game controllers, joystick, 156–162
gateway addresses, 122
gatewayIP() method (Ethernet library), 138
General MIDI instrument specification, 425

geocoding
about, 265
PROJECT 28: IP Geocoding, 355–361
GET command (HTTP)
about, 86–87
PROJECT 5: Networked Cat, 109
PROJECT 29: Catcam Redux, 379
$_GET environment variable, 17
Girder application, 452
GitHub site, 25, 321, 452
Global Positioning System. *See* GPS
global variables, 104, 107
Glolab, 448
GNU screen program, 18
Google Accessory Development Kit, 414
Google Voice, 366, 386
$GPGGA sentence, 280, 284
$GPGSV sentence, 280, 284
$GPRMC sentence, 280, 284
GPRS module, 395
GPS (Global Positioning System)
about, 267
distance ranging and, 272
multipath effect, 276
PROJECT 19: Reading GPS Serial Protocol, 263, 278–285
purchasing accessories, 285
trilateration and, 277
graphing results, 254–257
Gridconnect, 448
guard time, 194

H

handshake method (call-and-response), 63
hardware address
about, 79
composition of, 80
hardware vendors, 447–451. *See also* specific vendors
Hartman, Kate, 220
Hasson, Meredith, 415
Hayes AT command protocol
about, 68
GPRS support, 395
switching modes, 68
header pins
common components, 8
PROJECT 5: Networked Cat, 93
purchasing, 7
headers, defined, 56
Heading, Determining Using a Digital Compass (project 20)
project overview, 286–289
supplies for, 263
Headset Profile (Bluetooth), 64
Heathcote, Chris, 266, 267
heave translation, 290, 292
Hello Internet! Daylight Color Web Server (project 6)
project overview, 120–126
supplies for, 117
Hello, Wi-Fi! (project 12)
project overview, 217–218
supplies for, 183
helping hands, purchasing, 6
hexadecimal notation, 80, 424
HID (Human Interface Device) Profile, 64
Hirschmann, Daniel, 431
home automation
PROJECT 26: RFID Meets Home Automation, 303, 321–328
X10 protocol and, 322
home directory, 13
hook-up wire
common components, 8
purchasing, 7
HTML5 standard, 391–392
HTML (HyperText Markup Language)
about, 434
embedding PHP in, 16
HTTP 200 OK header, 380
HTTP 404 File Not Found header, 380
HTTP_ACCEPT_LANGUAGE environment variable, 354
HTTP (HyperText Transfer Protocol)
about, 84, 432
environment variables, 353–354
REST principle, 435–436
supported commands, 86–87
HTTP_USER_AGENT environment variable, 354
HTTP user agents, 355–356
hubs, defined, 78
Human Interface Device (HID) Profile, 64
Huntington, John, 431
Hypermedia UDP library for Processing, 453
HyperText Markup Language (HTML)
about, 434
embedding PHP in, 16
HyperText Transfer Protocol (HTTP)
about, 84, 432
environment variables, 353–354
REST principle, 435–436
supported commands, 86–87

I

I2C protocol
about, 119, 289, 421
PROJECT 27: Tweets from RFID, 345
SonMicro readers and, 333, 420
identification. *See also* physical identification; RFID technology
about, 301
network identification, 301, 353–359
PROJECT 22: Color Recognition Using a Webcam, 302, 306–309
PROJECT 23: Face Detection Using a Webcam, 302, 310–311
PROJECT 24: 2D Bar Code Recognition Using Webcam, 302, 313–315
PROJECT 25: Reading RFID tags in Processing, 302
PROJECT 27: Tweets from RFID, 303
PROJECT 28: IP Geocoding, 355–361
video identification, 305
IDEs (integrated development environments), 20
ID Innovations, 316, 318–320, 328
idle mode, 204
if() statement, 61, 385
if-then statements, 11
images
bar code recognition and, 315
capturing and uploading, 102–106
Images SI Inc., 448
IMAP (Internet Message Access Protocol), 88
induction property, 190

infrared communication
about, 185
data protocols, 186
how it works, 185–187
making it visible, 186
PROJECT 9: Infrared Control of a Digital Camera, 182, 188–189
PROJECT 15: Infrared Distance Ranger Example, 262, 268–269
sniffing IR signals, 187
troubleshooting, 186
Infrared Control of a Digital Camera (project 9)
project overview, 188–189
supplies for, 182
Infrared Distance Ranger Example (project 15)
project overview, 268–269
supplies for, 262
Inkscape tool, 25
instances, defined, 168
instance variables, 168
Institute for Interaction Design, 20
int data type
about, 11
scaling functions and, 57
integrated development environments (IDEs), 20
interactive systems
feedback loops and, 151
PROJECT 8: Networked Pong, 150, 153–177
Interactive Telecommunications Program (ITP), 248
interface modules, 322
Inter-Integrated Circuit. *See* I2C protocol
Interlink Electronics, 448
Internet Message Access Protocol (IMAP), 88
Internet Protocol addresses. *See* IP addresses
Internet Relay Chat (IRC), 151
internet transfer protocols, 432
intervalometers, 190
inverted logic, 43
invisible files, 13
IOGear, 43, 448
IP addresses
about, 79
constructing, 80
decimal notation and, 80
DHCP support, 138
DNS support, 81, 138
finding for hosts, 134–137
locating things and, 265
private, 81, 126
PROJECT 6: Hello Internet! Daylight Color Web Server, 122
PROJECT 28: IP Geocoding, 355–361
public, 81, 126
IP Geocoding (project 28)
mail environment variables, 357–359
project overview, 355–357
IRC (Internet Relay Chat), 151
IR communication. *See* infrared communication
IR LEDs, 185–187
is_bool() function (PHP), 17
is_int() function (PHP), 17
ISO 14443 standard, 317
ISO 15693 standard, 316
isset() function (PHP), 17
is_string() function (PHP), 17
ITP (Interactive Telecommunications Program), 248

J

Jameco Electronics, 448
JAN bar code symbology, 312
Java language
about, 452
classes and, 166
Processing language and, 11
JavaScript language, 391
JavaScript Object Notation (JSON), 432–433
Johansson, Sara, 301
joystick game controllers, 156–162
JSON (JavaScript Object Notation), 432–433

K

Kaufman, Jason, 75
Keyspan (vendor), 43, 448
Knörig, André, 25
Konsole program, 12

L

lamp control modules, 322, 325
Lantronix modules, 118, 449
LeafLabs Maple controller, 23
LEDs
on Arduino board, 27–28
common components, 8
IR, 185–187
opto-isolators and, 425
PROJECT 1: Type Brighter RGB LED Serial Control, 38, 46–49
PROJECT 7: Networked Air Quality Meter, 133
PROJECT 8: Networked Pong, 156–158, 161
PROJECT 10: Duplex Radio Transmission, 193
PROJECT 29: Catcam Redux, 377
purchasing, 7
PWM and, 127
troubleshooting support, 140
Lehrman, Paul D., 425
less editor, 14
Libelium, 395, 449
lighting filters, 120, 121
linefeed character, 123
Linux environment
Arduino/Wiring modules and, 21, 26
Bluetooth support, 65, 71
capturing and uploading images, 102
IP addressing and, 80
remote access applications for, 11, 12
screen programs and, 18
serial communication tools, 18
terminal emulation programs, 18
test chat server, 153
TextFinder library, 133
X-CTU software, 231
Linx Technologies, 449
LiquidCrystal library, 343
listen() method (UDP library), 229
list() function (Serial library), 296
loadStrings() method (Processing), 98, 412
localhost address, 82
localIP() method (Ethernet library), 138

location
about, 261
determining orientation, 286
network location, 264–266
physical location, 264–266
popular techniques, 266
PROJECT 15: Infrared Distance Ranger Example, 262, 268–269
PROJECT 16: Ultrasonic Distance Ranger Example, 263, 270–271
PROJECT 17: Received Signal Strength Using XBee Radios, 263
PROJECT 18: Reading Received Signal Strength Using Bluetooth Radios, 263, 276
PROJECT 19: Reading GPS Serial Protocol, 263, 278–285
PROJECT 20: Determining Heading Using a Digital Compass, 263, 286–289
PROJECT 21: Determining Attitude Using an Accelerometer, 263, 290–298
PROJECT 32: Fun with MIDI, 418, 427–430
PROJECT 33: Fun with REST, 437–439
lock files, 18
logical layer
about, 40
RS-232 serial protocol, 43
TTL serial protocol, 42
USB protocol, 42
logical operators, 423–424
logout command, 15
London, Kati, 220
loopback address, 82
loop() method (Processing), 20, 229
Lotan, Gilad, 248, 249, 258
LPRS (Low Power Radio Solutions), 449
ls command, 14

M

MAC addresses
about, 79
composition of, 80
PROJECT 6: Hello Internet! Daylight Color Web Server, 122
Macam webcam driver, 452
Mac OS X environment
Arduino/Wiring modules and, 21, 24
Bluetooth support, 65, 71
capturing and uploading images, 102
IP addressing and, 79
network settings panel, 79
ping tool, 82
remote access applications, 12
screen programs and, 18
serial communication tools, 18
terminal emulation programs, 18
test chat server, 153
TextFinder library, 133
USB protocol and, 43
X-CTU software, 231
Madsen, Leif, 386
Maker Notebooks, 25
Maker Shed, 449
Making Things, 449
man command, 15
map() function (Processing), 20, 57
marbles as physical tokens, 309
Margolis, Michael, 133
markup languages, 433–434
Master In, Slave Out (MISO) pin, 119, 376
Master Out, Slave In (MOSI) pin, 119, 376
MAX3232 chip, 44
Maxbotix, 420
Maxim Integrated Products, 449
Maxim Technologies, 44
Max tool, 452
Mazurov, Oleg, 414
Media Access Control addresses. *See* MAC addresses
Meier, Roger, 18
Melo, Mauricio, 75
mesh networks, 249
messages. *See* sending messages
Microchip microcontrollers, 23, 449
microcontrollers
about, 23
Arduino module, 20–21
Atmel AVR, 23
attaching to mobile phone networks, 395
common analog sensors, 8
common batter snap adapters, 8
common capacitors, 8
common diodes, 8
common header pins, 8
common LEDs, 8
common potentiometers, 8
common push buttons, 8
common resistors, 8
common solderless breadboards, 8
common transistors, 8
common voltage regulators, 8
communication between, 204
connecting, 215
defined, 3
8-bit controllers, 23
GPRS for, 395
hook-up wire, 8
making mobile, 205
Microchip PIC, 23
MIDI support, 425
mobile phones and, 366–367
physical interfaces, 3
physical tools, 5–9
process overview, 3
programming to use XBee module, 200–204
purchasing modules, 7
serial ports and, 17–18
32-bit controllers, 23
used in projects, 444
Wiring module, 20–21
XBee and, 197
microSD cards, 377
microwave range, 191
MIDI Manufacturers' Organization site, 425
MIDI (Musical Instrument Digital Interface)
about, 423, 425–426
command bytes, 424, 425
General MIDI instrument specification, 425
noteOff() method, 428
noteOn() method, 428
PROJECT 32: Fun with MIDI, 418, 427–430
resetMidi() method, 430
sendMidi() method, 428
status bytes, 425

Mifare protocol
about, 317
PROJECT 27: Tweets from RFID, 329
reading from tags, 343
writing to tags, 335
MISO (Master In, Slave Out) pin, 119, 376
mkdir command, 13
mkdir() method (SD library), 376
MMS (Multimedia Message Service), 393
mobile phones
about, 366–368
microcontrollers and, 366–367, 395
native applications for, 368, 396–400
PROJECT 29: Catcam Redux, 364, 369–383
PROJECT 30: Phoning the Thermostat, 365, 386–400
PROJECT 31: Personal Mobile Datalogger, 365, 401–414
modems
about, 64
Bluetooth support, 64
defined, 78
Hayes AT command protocol, 68
serial-to-Ethernet, 118
MOD_REWRITE module, 437
Mok, Jin-Yo, 149, 178
Monski Pong (project 2)
project overview, 50–60
supplies for, 39
Monski Pong, Wireless (project 3)
adjusting program, 65–66
project overview, 64–67
supplies for, 39
MOSI (Master Out, Slave In) pin, 119, 376
most significant bit, 423
most significant digit, 423
MoSync platform, 392
Mouser (vendor), 449
MSP library, 452
Mulligan, Ryan, 288
Multi Camera IR Control library, 188–189
Multimedia Message Service (MMS), 393
multimeters, 6
multipath effect, 276
multiplexing, 191
Murphy, Glenn, 280
Musical Instrument Digital Interface. *See* MIDI
Myers, Ryan, 72
mystery radio error, 184

N

NADA tool, 453
nameservers, 81
name-value pairs, 432
nano editor, 14
National Marine Electronics Association, 278
native applications for mobile phones, 368, 396–400
navigational terms, 290
nearfield communications (NFC), 329
needlenose pliers, 6
Negotiating in Bluetooth (project 4)
project overview, 68–71
supplies for, 39
netmask, 80, 122
NetMedia, 449
network cameras, 384–385
network communication
about, 37
layers of agreement, 40
PROJECT 1: Type Brighter RGB LED Serial Control, 38, 46–49
PROJECT 2: Monski Pong, 39, 50–60
PROJECT 3: Wireless Monski Pong, 39, 64–67
PROJECT 4: Negotiating in Bluetooth, 39, 68–71
PROJECT 5: Networked Cat, 76, 89–111
as session-based, 226
Networked Air Quality Meter (project 7)
project overview, 127–139
supplies for, 117
Networked Cat (project 5)
capturing and uploading images, 102–106
final assembly, 106–111
project overview, 89–98
sending mail from the cat, 98–99
supplies for, 76
uploading files to servers, 101–102
web page for cat cam, 99–100
web page for web cam, 110–111
Networked Pong (project 8)
anatomy of player object, 166–168
balance board client, 163–166
client-side overview, 155–156
joystick client, 156–162
main server program, 168–177
project overview, 153
server-side overview, 166
supplies for, 150
test chat server, 153–155
network identification
about, 301, 353–354
PROJECT 28: IP Geocoding, 355–361
network interface modules
about, 118–119
PROJECT 6: Hello Internet! Daylight Color Web Server, 117, 120–126
Network library
about, 11
PROJECT 5: Networked Cat, 104
serverEvent() method, 155, 168, 172, 174
network location, 264–266
network maps, 77–79
network protocols, 3
network servers, 3
network stack, 118
Nevarez, Ariel, 415
Newark In One Electronics, 449
New Micros (vendor), 450
New York University, 248
NFC (nearfield communications), 329
Nguyen, Tuan Anh T., 1
9-pin serial connector, 43
NMEA 0183 protocol, 278–280, 285
node discovery, 230
node identifier, 231
Nordberg, J., 261, 298
noteOff() method (MIDI), 428
noteOn() method (MIDI), 428
nslookup command, 134

O

objects
communication considerations, 2, 4
interface elements and, 2
physical tools for, 5–9
software tools for, 9–12

O'Brien, Derrick, 280
octets (IP addresses), 80
Olivero, Giorgio, 25
omnidirectional transmission, 185
one-dimensional bar codes, 312
OpenCV library
  about, 305
  bar code recognition, 313
  cascade() method, 310
  color recognition methods, 306
  detect() method, 311
  face detection methods, 310
OpenSSH program, 12
Open Systems Interconnect (OSI) model, 40
operational codes, 423
optical recognition
  defined, 304
  limitations, 315
opto-isolators, 425
O'Reilly Emerging Technology Conference, 266
orientation
  PROJECT 20: Determining Heading Using a Digital Compass, 286–289
  PROJECT 21: Determining Attitude Using an Accelerometer, 290–298
  types of sensors, 286
OR (|) logical operator, 423
oscilloscopes
  about, 34
  making infrared visible, 186
  purchasing, 6
  sniffing IR signals, 187
OSI (Open Systems Interconnect) model, 40
Oslo School of Architecture and Design, 301

P

Pablo, Angela, 248, 249, 258
packet length, 56, 240
packets, defined, 56
packet switching, 81–82
Paek, Joo Youn, 37
Pakhchan, Syuzi, 402
PAN (Personal Area Network) ID, 196, 234
Parallax
  about, 450
  Basic Stamp microcontroller, 23, 29
  RFID technology, 317
parent directory, 13
parsePacket() method (UDP library), 229, 255
passive distance ranging, 267, 272
passive RFID systems, 315
pattern recognition, 309
payloads, defined, 56
PBASIC language, 23
PD (Puredata) tool, 453
PEAR (PHP Extension and Application Repository), 453
Peek, Jerry, 15
peek() method (SD library), 376
perf boards, 156, 158
permissions, changing, 14, 358
Personal Area Network (PAN) ID, 196, 234
personal computers
  about, 3
  mobile phones and, 366
  physical interface, 3
  types of, 3–4
Personal Mobile Datalogger (project 31)
  circuits, 402
  coding, 405–414
  construction, 402–405
  project overview, 401
  supplies for, 365
Phidgets (vendor), 450
PhoneGap platform, 392
Phoning the Thermostat (project 30)
  project overview, 386
  supplies for, 365
  text messaging, 393–395
photocells, 120, 121
PHP Extension and Application Repository (PEAR), 453
PHP language
  about, 15, 453
  additional information, 17
  ASCII strings and, 86
  creating scripts, 16–17
  date() function, 17
  embedding in HTML pages, 16
  fgetss() function, 131
  fopen() function, 132
  handling variables, 17
  identifying installed version, 15
  is_bool() function, 17
  is_int() function, 17
  isset() function, 17
  is_string() function, 17
  preg_match function, 132
  Processing language and, 17
  reading web pages, 130–137
  sending email, 88
physical identification. *See also* RFID technology
  about, 301, 304
  bar code recognition, 312–315
  color recognition, 305–308
  face detection, 309–312
  marbles as physical tokens, 309
  pattern recognition, 309
  PROJECT 22: Color Recognition Using a Webcam, 302, 306–309
  PROJECT 23: Face Detection Using a Webcam, 302, 310–311
  PROJECT 24: 2D Bar Code Recognition Using Webcam, 302, 313–315
  shape recognition, 309
  video identification, 305
physical interfaces
  defined, 2
  physical tools for, 5–9
  types of computers and, 3
physical layer
  about, 40
  protocol considerations, 419–420
  RS-232 serial protocol, 43
  TTL serial protocol, 42
  USB protocol, 42
physical location
  distance ranging, 267
  network location and, 264–266
physical objects
  about, 2
  identifying, 304–305
  interface elements and, 2
physical tools
  about, 5–9
  debugging methods, 140
PICAXE environment, 23
PicBasic Pro, 29, 453
ping command, 81, 134
pitch rotation, 290, 292

Player object (project 8), 166–168
pliers, needlenose, 6
Pololu (vendor), 288, 450
POP (Post Office Protocol), 88
port numbers
  defined, 83
  private IP addresses and, 126
position() method (SD library), 376
POST command (HTTP)
  about, 86–87
  multipart, 106
  PROJECT 5: Networked Cat, 104, 110
  PROJECT 29: Catcam Redux, 379
$_POST environment variable, 17
POSTNET bar code symbology, 312
Post Office Protocol (POP), 88
potentiometers
  common components, 8
  defined, 30
  PROJECT 11: Bluetooth Transceivers, 209
  PROJECT 13: Reporting Toxic Chemicals in the Shop, 238
  PROJECT 27: Tweets from RFID, 343, 347
  purchasing, 7
  usage example, 33
power connectors, 6
power supplies, 6, 344
Power Switch Tail, 372
pqrcode library, 314
preg_match() function (PHP), 132
print() function
  SD library, 376
  Serial library, 54, 121
println() function
  SD library, 376
  Serial library, 121
private IP addresses, 81, 126
Processing language. *See also* specific methods
  about, 9–11, 453
  additional information, 11
  Android apps and, 396
  Arduino/Wiring modules and, 20
  classes and, 166
  flow control and, 62–63
  libraries supported, 227
  PHP language and, 17
profiles, defined, 64
programming languages, 434. *See also* specific languages
PROJECT 1: Type Brighter RGB LED Serial Control
  project overview, 46–49
  supplies for, 38
PROJECT 2: Monski Pong
  project overview, 50–60
  supplies for, 39
PROJECT 3: Wireless Monski Pong
  adjusting program, 65–66
  project overview, 64–67
  supplies for, 39
PROJECT 4: Negotiating in Bluetooth
  project overview, 68–71
  supplies for, 39
PROJECT 5: Networked Cat. *See also* PROJECT 5: Networked Cat: Catcam Redux
  capturing and uploading images, 102–106
  final assembly, 106–111
  project overview, 89–98
  sending mail from the cat, 98–99
  supplies for, 76
  uploading files to servers, 101–102
  web page for cat cam, 99–100
  web page for web cam, 110–111
PROJECT 6: Hello Internet! Daylight Color Web Server
  project overview, 120–126
  supplies for, 117
PROJECT 7: Networked Air Quality Meter
  project overview, 127–139
  supplies for, 117
PROJECT 8: Networked Pong
  anatomy of player object, 166–168
  balance board client, 163–166
  client-side overview, 155–156
  joystick client, 156–162
  main server program, 168–177
  project overview, 153
  server-side overview, 166
  supplies for, 150
  test chat server, 153–155
PROJECT 9: Infrared Control of a Digital Camera
  project overview, 188–189
  supplies for, 182
PROJECT 10: Duplex Radio Transmission
  communication between microcontrollers, 204
  configuring XBee modules serially, 193–199
  programming microcontrollers to user XBee module, 200–204
  project overview, 193
  supplies for, 182
PROJECT 11: Bluetooth Transceivers
  circuits, 206–207
  commands, 207–209
  connecting two microcontrollers, 215
  connecting two radios, 209–215
  project overview, 206
  supplies for, 183
PROJECT 12: Hello, Wi-Fi!
  project overview, 217–218
  supplies for, 183
PROJECT 13: Reporting Toxic Chemicals in the Shop
  circuits, 235–238
  common problems, 242
  project overview, 232–234
  radio settings, 234–235
  reading XBee protocol, 238–245
  supplies for, 224
PROJECT 14: Relaying Solar Cell Data Wirelessly
  circuits, 248–253
  graphing results, 254–257
  project overvew, 248
  radio settings, 248
  supplies for, 225
PROJECT 15: Infrared Distance Ranger Example
  project overview, 268–269
  supplies for, 262
PROJECT 16: Ultrasonic Distance Ranger Example
  project overview, 270–271
  supplies for, 263
PROJECT 17: Received Signal Strength Using XBee Radios
  project overview, 273–275
  supplies for, 263

PROJECT 18: Reading Received Signal Strength Using Bluetooth Radios
project overview, 276
supplies for, 263
PROJECT 19: Reading GPS Serial Protocol
project overview, 278–285
supplies for, 263
PROJECT 20: Determining Heading Using a Digital Compass
project overview, 286–289
supplies for, 263
PROJECT 21: Determining Attitude Using an Accelerometer, 290–298
PROJECT 22: Color Recognition Using a Webcam
project overview, 306–309
supplies for, 302
PROJECT 23: Face Detection Using a Webcam
project overview, 310–311
supplies for, 302
PROJECT 24: 2D Bar Code Recognition Using Webcam
project overview, 313–315
supplies for, 302
PROJECT 25: Reading RFID tags in Processing
project overview, 318–320
supplies for, 302
PROJECT 26: RFID Meets Home Automation
project overview, 321–328
supplies for, 303
PROJECT 27: Tweets from RFID
circuits, 329–332, 343–346
construction, 351–352
project overview, 329
saving program memory, 346–351
SonMicro communications protocol, 333–335
supplies for, 303
troubleshooting, 350
writing Mifare tags, 335
PROJECT 28: IP Geocoding
mail environment variables, 357–359
project overview, 355–357
PROJECT 29: Catcam Redux. *See also* Processing language: Networked Cat
circuits, 372–373
coding, 373–383
project overview, 369–371
supplies for, 364
PROJECT 30: Phoning the Thermostat
project overview, 386
supplies for, 365
text messaging, 393–395
PROJECT 31: Personal Mobile Datalogger
circuits, 402
coding, 405–414
construction, 402–405
project overview, 401
supplies for, 365
PROJECT 32: Fun with MIDI
project overview, 427–430
supplies for, 418
PROJECT 33: Fun with REST, 437–439
protocols. *See also* specific protocols
binary, 422–424, 431
defined, 2–3
good habits for, 5
making connections, 419–421
network, 3
planning physical system, 421
PROJECT 1: Type Brighter RGB LED Serial Control, 46–49
PROJECT 2: Monski Pong, 50–60
PROJECT 32: Fun with MIDI, 418, 427–430
PROJECT 33: Fun with REST, 437–439
REST principle, 435–436
serial, 3
text, 422–424, 432–434
prototyping boards
about, 156
depicted, 158
used in projects, 444
prototyping shields, 7
PSTN (public switched telephone network), 386
public IP addresses, 81, 126
public switched telephone network (PSTN), 386
pull-down resistors, 30
pull-up resistors, 30
pulses
carrier waves and, 185
communication protocols and, 2
pulse width modulation (PWM), 127, 234
pulse width ratio, 235
Puredata (PD) tool, 453
push buttons
common components, 8
PROJECT 11: Bluetooth Transceivers, 209
purchasing, 7
PUT command (HTTP), 86
PuTTY program
about, 18, 453
configuring serial connection, 19
disconnecting connection, 18
downloading, 12, 84
PWM (pulse width modulation), 127, 234
Python language, 395

Q

QR code
about, 305, 312
PROJECT 24: 2D Bar Code Recognition Using Webcam, 313–315
QRcode library, 453
Quantified Self meetups, 401
querying for devices
UDP support, 227–230
using 802.15.4, 230–231
Quicktime program, 102

R

RabbitCore processors, 118
radio frequency (RF) shields, 191
Radio Shack, 450
radios, purchasing, 216
radio transceivers. *See* transceivers
radio transmission. *See also* signal strength
about, 185, 190
digital and analog, 190
node discovery, 230
node identifier, 231
PROJECT 10: Duplex Radio Transmission, 182, 193–205
PROJECT 17: Received Signal Strength Using XBee Radios, 263, 273–275

PROJECT 18: Reading Received Signal Strength Using Bluetooth Radios, 263, 276
read() function (Serial library), 121
Reading GPS Serial Protocol (project 19)
project overview, 278–285
supplies for, 263
Reading Received Signal Strength Using Bluetooth Radios (project 18)
project overview, 276
supplies for, 263
Reading RFID tags in Processing (project 25)
project overview, 318–320
supplies for, 302
Reas, Casey, 11
received signal strength (RSSI), 273
Received Signal Strength Using XBee Radios (project 17)
project overview, 273–275
supplies for, 263
receive() method (UDP library), 228
receive pin (RX), 45, 215, 240
receivers
defined, 185
distance ranging and, 272
Rectangle object (Java), 310
relative path, 13
Relaying Solar Cell Data Wirelessly (project 14)
circuits, 248–253
graphing results, 254–257
project overvew, 248
radio settings, 248
supplies for, 225
REMOTE_ADDR environment variable, 354
remove() method (SD library), 376
Reporting Toxic Chemicals in the Shop (project 13)
circuits, 235–238
common problems, 242
project overview, 232–234
radio settings, 234–235
reading XBee protocol, 238–245
supplies for, 224
Representational State Transfer (REST)
about, 435–436
PROJECT 33: Fun with REST, 437–439
$_REQUEST environment variable, 17, 85, 406
Request to Send (RTS), 45
resetMidi() method (MIDI), 430
resistors
common components, 8
PROJECT 5: Networked Cat, 89
pull-down, 30
pull-up, 30
purchasing, 7
REST (Representational State Transfer)
about, 435–436
PROJECT 33: Fun with REST, 437–439
Revolution Education, 23
Reynolds Electronics, 186, 450
RFID technology
about, 304, 315–317
bar code recognition and, 315
PROJECT 25: Reading RFID tags in Processing, 302, 318–320
PROJECT 26: RFID Meets Home Automation, 303, 321–328
PROJECT 27: Tweets from RFID, 329–350
testing circuits, 324
RF (radio frequency) shields, 191
ring networks, 77
rising edge of the clock, 289
RJ-11 jacks, 321
RJ-45 connectors, 431
RMC data, 280
rm command, 15
rmdir command, 14
rmdir() method (SD library), 376
Rogue Robotics, 22
roll rotation, 290, 292
root directory, 13
rotate() method (Processing), 297
rotations
defined, 290
depicted in three dimensions, 292
measuring, 290, 292
routers
defined, 78
gateway addresses, 122
port numbers and, 126
Roving Networks, 69, 206, 450
RS-232 serial protocol, 43, 421
RS-485 protocol, 421, 431
RS Online, 450
RSSI (received signal strength), 273
RTS (Request to Send), 45
RX (receive pin), 45, 215, 240
rxvt program, 12

S

safety goggles, 6
Samtec (vendor), 450
scalar variables, 17
Schelling, Nahana, 415
Schneider, Andrew, 223
screen programs
closing down, 18
defined, 18
screwdrivers, 6
SD cards
best practices, 376–377
reading from, 375
writing to, 376
SD library
about, 376
adding, 375
begin() method, 375
close() function, 376
exists() method, 376
flush() function, 376
mkdir() method, 376
peek() method, 376
position() method, 376
print() function, 376
println() function, 376
remove() method, 376
rmdir() method, 376
seek() method, 376
size() method, 376
write() function, 376
Seeed Studio, 34, 450
seek() method (SD library), 376
Seidle, Nathan, 372
send() method (UDP library), 227
sending messages
broadcast messages, 193, 226–231
directed messages, 246–257
good habits for, 4
HTTP commands, 86–87
packet switching and, 81–82
PROJECT 5: Networked Cat, 95–99
PROJECT 10: Duplex Radio Transmission, 193
sessions versus, 226
troubleshooting, 81
sendMidi() method (MIDI), 428

sendPacket() method (UDP library), 229
sensors
  common components, 8
  determining distance, 267
  feedback loops and, 151
  PhoneGap platform, 392
  purchasing, 7
  used in projects, 446
sentences (NMEA protocol), 279
serial buffer, 62
serial clock pin, 289
serial communication. *See also* asynchronous serial communication; synchronous serial communication
  about, 3, 17, 28–30
  debugging methods, 140–142
  defined, 40
  Linux environment, 18
  Mac OS X environment, 18
  NMEA 0183 protocol, 278
  picking protocols, 420–421
  Windows environment, 18
serial data pin, 289
Serial library
  about, 11
  available() function, 121
  list() function, 296
  print() function, 54, 121
  println() function, 121
  read() function, 121
  write() function, 54, 121
Serial Monitor (Arduino), 31, 53, 230
Serial Peripheral Interface. *See* SPI
Serial Port Profile (SPP), 64, 401
serial ports
  about, 17
  Arduino/Wiring modules, 29
  closing, 18, 57
  releasing, 18, 57
  usage considerations, 18
  Windows environment, 18
serial-to-Ethernet modems, 118
serial-to-USB converters, 7
serverEvent() method (Network library), 155, 168, 172, 174
servers
  defined, 82
  making public, 383
  PROJECT 8: Networked Pong, 166–177
  uploading files to, 101–102
  web browsing and, 83
  writing test programs for, 146
server-side scripts, 15
Service Discovery Protocol, 64
Session Initiation Protocol (SIP), 386–387
sessionless networks
  about, 223
  broadcast messages, 227–231
  PROJECT 13: Reporting Toxic Chemicals in the Shop, 224, 232–245
  PROJECT 14: Relaying Solar Cell Data Wirelessly, 225, 248–257
  sessions versus messages, 226
sessions
  defined, 152, 226
  messages versus, 226
setting the bit, 423
setup() method (Processing), 11, 20
Setz, Sebastian, 188
SFTP library, 453
shape recognition, 309
shields. *See also* Ethernet shield
  compatibility considerations, 22
  defined, 22
  depicted for Arduino module, 22
  making, 22
  used in projects, 444
  Wireless, 199
Shiffman, Daniel, 11, 313, 453
shift left operator (<<), 423
shift right operator (>>), 423
Short Messaging Service (SMS)
  about, 393–394
  mobile phone support, 368
signal connections, 31
signal strength
  distance ranging and, 267, 272
  measuring, 275
  PROJECT 17: Received Signal Strength Using XBee Radios, 263, 273–275
  PROJECT 18: Reading Received Signal Strength Using Bluetooth Radios, 263, 276
  RFID systems and, 316
SIM cards, 395
SimpleDmx library, 431
Simple Mail Transfer Protocol (SMTP), 88, 432
SIP (Session Initiation Protocol), 386–387
size() method (SD library), 376
Sjaastad, Mosse, 301
sketches
  defined, 9
  flow control and, 62
Sketchtools NADA tool, 453
Sklar, David, 17
Skyetek, 451
Skyhook site, 267
Slave Select (SS) pin, 119
Slavin, Kevin, 312
SM13X FU tool, 336
Smarthome (vendor), 322, 451
smartphones. *See* mobile phones
Smith, Jared, 386
SMRFID Mifare v1.2 diagnostic software, 336
SMS (Short Messaging Service)
  about, 393–394
  mobile phone support, 368
SMTP (Simple Mail Transfer Protocol), 88, 432
sockets
  defined, 152
  PROJECT 8: Networked Pong, 150, 153–177
software interfaces, 2, 4
software objects
  about, 2
  interface considerations, 2, 4
software oscilloscopes, 34
SoftwareSerial library, 321, 324
software tools. *See also* specific software
  list of, 452–453
  Processing tool, 9–11
  remote access applications, 11–12
Solaris environment, 11
soldering irons and solder, 6
solderless breadboards
  Arduino/Wiring modules and, 30, 31
  common components, 8
  perf boards and, 156
  purchasing, 7
sonMicroEvent() method (SonMicroReader library), 339
SonMicroReader library, 335, 339

SonMicro readers
about, 317, 420
PROJECT 27: Tweets from RFID, 303, 329–356
Spark Fun Electronics
about, 451
Bluetooth Mate radio, 32, 39, 68–71, 206–215, 278–285
Breakout Board, 194
GPRS support, 395
GPS receivers, 285
musical instrument shield, 418, 425
music instrument shield, 429
RFID technology, 316, 319, 330
SD card shield, 376
XBee Explorer, 193, 201
SPI library
about, 119
PROJECT 6: Hello Internet! Daylight Color Web Server, 122
PROJECT 13: Reporting Toxic Chemicals in the Shop, 243
SPI (Serial Peripheral Interface)
about, 119, 421
connections supported, 119
PROJECT 6: Hello Internet! Daylight Color Web Server, 122
SD cards and, 376
SPP (Serial Port Profile), 64, 401
Spreadtrum Technologies, 395
Sridhar, Sonali, 261, 298
Sriskandarajah, Sai, 220
ssh program, 12
SSID, 217
SS (Slave Select) pin, 119
star networks, 77
start-of-transmission (STX) byte, 318
state machines, 346
Strang, John, 15
String data type, 11
STX (start-of-transmission) byte, 318
subnetMask() method (Ethernet library), 138
subnet masks, 80, 122
subnets, defined, 80
supplies. *See also* under specific projects
breakout boards, 445
common components, 445–446
communications modules, 444–445
connectors, 445
infrastructure, 444
microcontrollers, 444
miscellaneous, 446
prototyping boards, 444
sensors, 446
shields, 444
specialty components, 446
surge translation, 290, 292
sway translation, 290, 292
switches, defined, 78
Symmetry Electronics, 451
synchronous serial communication
about, 40, 41, 421
SPI and, 119
X10 protocol and, 322

T

tab-separated values (TSV), 432
tags, defined, 387
tails, defined, 56
TCP/IP stack, defined, 118
TCP (Transmission Control Protocol)
about, 152, 226
PROJECT 8: Networked Pong, 150, 153–177
UDP and, 226
telephone answering machine, 309
telnet
escape key combination, 153
network modules and, 118
reliability of, 12
Windows limitations, 84
terminal emulation programs
about, 17–18
OpenSSH and, 12
PROJECT 4: Negotiating in Bluetooth, 69
Terminal program, 12
test leads, alligator clip, 7
test programs
for clients, 143–145
for servers, 146
Texas Instruments, 23, 316
TextFinder library
PROJECT 7: Networked Air Quality Meter, 133
PROJECT 11: Bluetooth Transceivers, 209
PROJECT 29: Catcam Redux, 378
text messaging, 393–395
text protocols, 422–424, 432–434
thermostats
PROJECT 29: Catcam Redux, 364, 369–383
PROJECT 30: Phoning the Thermostat, 365, 386–400
32-bit microcontrollers, 23
tilt() method (Processing), 296, 297
time-division multiplexing, 191
Tinker DMX shield, 431
TinkerKit RFID shield, 330
TinkerProxy application, 453
TI-RFID, 451
Todino-Gonguet, Grace, 15
tools
diagnostic, 140–146, 322, 336
physical, 5–9
serial communication, 17–20
software, 9–12
transceivers
about, 3, 185, 192
PROJECT 10: Duplex Radio Transmission, 193
PROJECT 11: Bluetooth Transceivers, 183, 206–215
transfer protocols, 432
transistors
common components, 8
purchasing, 7
translate() method (Processing), 297
translations
depicted, 290
depicted in three dimensions, 292
Transmission Control Protocol. *See* TCP
transmit pin (TX), 45, 215, 240
transmitters, defined, 185
triangulation, 277
trilateration
about, 267, 277
PROJECT 19: Reading GPS Serial Protocol, 263, 278–285
Trossen Robotics, 451
troubleshooting
embedded modules, 140–146
IR projects, 186
PROJECT 13: Reporting Toxic Chemicals in the Shop, 242
PROJECT 27: Tweets from RFID, 350
sending messages, 81
X10 projects, 325
TSV (tab-separated values), 432

TTL serial protocol
about, 421
defined, 42
GPRS support, 395
RFID technology and, 317
RS-232 adapters and, 43
SonMicro readers and, 420
TTL-to-RS-232 converters, 44
Tully, Tim, 425
Tweets from RFID (project 27)
circuits, 329–332, 343–346
construction, 351–352
project overview, 329
saving program memory, 346–351
SonMicro communications protocol, 333–335
supplies for, 303
troubleshooting, 350
writing Mifare tags, 335
Twilio
about, 366, 386, 453
PROJECT 30: Phoning the Thermostat, 365, 386–400
TwiML markup schema, 388–390
Twitter-related project. *See* PROJECT 27: Tweets from RFID
TWI (Two-Wire Interface), 119, 421
two-dimensional bar codes, 312, 315
Two-Wire Interface (TWI), 119, 421
TX (transmit pin), 45, 215, 240
Type Brighter RGB LED Serial Control (project 1)
project overview, 46–49
supplies for, 38

U

UART (Universal Asynchronous Receiver-Transmitter), 324, 419
Ubuntu environment
Bluetooth support, 65, 71
capturing and uploading images, 102
IP addressing and, 80
Software Update tool, 26
TextFinder library, 133
UDP library
about, 227, 453
beginPacket() method, 229
endPacket() method, 229, 253
listen() method, 229
parsePacket() method, 229, 255
receive() method, 228
send() method, 227
sendPacket() method, 229
UDP (User Datagram Protocol)
about, 152, 226
datagram support, 226, 246
querying for devices, 227–230
TCP and, 226
Ultrasonic Distance Ranger Example (project 16)
project overview, 270–271
supplies for, 263
Uncommon Projects, 451
Universal Asynchronous Receiver-Transmitter (UART), 324, 419
Universal Product Code (UPC), 312
Universal Serial Bus (USB) protocol
about, 42, 43, 421
Android devices and, 414
Unix environment
command-line interface and, 11
command user manual, 15
invisible files, 13
UPC (Universal Product Code), 312
upgrading firmware on XBee radios, 231
USB-A-to-Mini-B cable, 39
USB cables, 7
USB-to-RS-232 adapters, 43
USB-to-serial adapters
Arduino modules and, 43, 45, 119
depicted, 39
PROJECT 4: Negotiating in Bluetooth, 69
PROJECT 6: Hello Internet! Daylight Color Web Server, 120
PROJECT 11: Bluetooth Transceivers, 207
PROJECT 27: Tweets from RFID, 330
XBee radios and, 230
USB-to-TTL serial cable, 43, 45
USB-to-XBee adapters
PROJECT 17: Received Signal Strength Using XBee Radios, 273
purchasing, 197
USB (Universal Serial Bus) protocol
about, 42, 43, 421
Android devices and, 414
User Datagram Protocol. *See* UDP

V

Van Meggelen, Jim, 386
variables
environment, 17, 353–354, 432
global, 104, 107
instance, 168
PHP considerations, 17
scalar, 17
Verify function (Arduino), 26
video identification, 305
viewing files, 14
Virtual Terrain Project, 265
voice communications
GPRS and, 395
mobile phone support, 368
VoIP (Voice over IP), 386–387
voltage dividers
defined, 30
usage example, 33
voltage regulators
common components, 8
PROJECT 13: Reporting Toxic Chemicals in the Shop, 236
purchasing, 7
variations of, 32
voltage triggers, 248

W

web browsing
about, 82–86
mobile phones and, 368
PROJECT 5: Networked Cat, 89–111
webcams
PROJECT 5: Networked Cat, 99–100, 110–111
PROJECT 22: Color Recognition Using a Webcam, 302, 306–309
PROJECT 23: Face Detection Using a Webcam, 302, 310–311
PROJECT 24: 2D Bar Code Recognition Using Webcam, 302, 313–315
PROJECT 29: Catcam Redux, 364, 369–383
web interfaces, 118
web scrapers
about, 127
PROJECT 7: Networked Air Quality Meter, 117, 127–139

web servers
PROJECT 6: Hello Internet! Daylight Color Web Server, 117, 120–126
TCP support, 152
WEP keys, 217, 218
while() statement, 379–380
Windows environment
Arduino/Wiring modules and, 21, 26
Bluetooth support, 65, 71
capturing and uploading images, 102
IP addressing and, 79
network settings panel, 79
ping tool, 82
remote access applications, 12
serial communication tools, 18
telnet limitations, 84
terminal emulation programs, 18
test chat server, 153
TextFinder library, 133
X-CTU software, 231
wire, hook-up
common components, 8
purchasing, 7
wireless communication
about, 184
diagnostics, 219
PROJECT 3: Wireless Monski Pong, 39, 64–67
PROJECT 9: Infrared Control of a Digital Camera, 182, 188–189
PROJECT 10: Duplex Radio Transmission, 182, 193–205
PROJECT 11: Bluetooth Transceivers, 183, 206–215
PROJECT 12: Hello, Wi-Fi!, 183, 217–218
PROJECT 14: Relaying Solar Cell Data Wirelessly, 225, 248–257
purchasing modules, 216
radio signal strength, 275
types of, 185
Wireless Monski Pong (project 3)
adjusting program, 65–66
project overview, 64–67
supplies for, 39
Wireless shields, 199
Wire library, 288, 289
wire strippers, 6
Wiring module
about, 20–21, 453
depicted, 21
inputs and outputs for, 24
installation process, 24–26
programming environment depicted, 27
serial ports, 29
solderless breadboards and, 30
WizNet module, 119
Wordpress blogs, 25
Worldkit, 265
WPA2 encryption, 217
WPA encryption, 217, 218
write() function
SD library, 376
Serial library, 54, 121

X

X10 protocol
about, 321–322
synchronization problems, 325
testing output, 325
unit codes, 324
XBee radios
about, 32
configuring serially, 193–199
factory default settings, 234
mesh networking and, 249
mounting on breakout board, 194
programming microcontrollers to use, 200–204
PROJECT 13: Reporting Toxic Chemicals in the Shop, 224, 232–245
PROJECT 17: Received Signal Strength Using XBee Radios, 263, 273–275
purchasing, 195
purchasing accessories, 197
querying for using 802.15.4, 230–231
signal strength, 275
upgrading firmware on, 231
XBee-to-USB serial adapters, 193
X-CTU software, 231
X-Mailer field, 359
XML (eXtensible Markup Language), 387–391
XOR (^) logical operator, 424
xterm program, 12

Y

yaw rotation, 290, 292

Z

ZigBee protocol, 249

Four hot Arduino kits from the makers behind MAKE and Maker Faire!

Ultimate Microcontroller Pack



The Ultimate Microcontroller Pack includes everything you need to dive right into the world of microcontollers. The 100+ components allow you to complete nearly any online tutorial without having to source individual parts. Everyone from beginners to advanced users will appreciate all that the Ultimate Microcontroller Pack has to offer.

Mintronics: Survival Pack



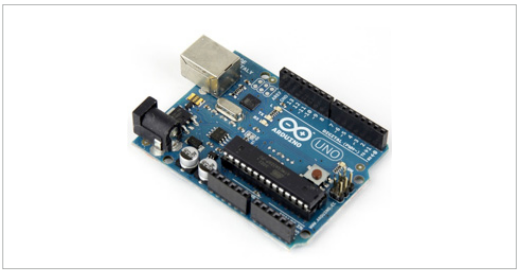
You never know when you'll find yourself in a MacGyver moment. It can happen anywhere, at any time. Wouldn't it be easier to hack those electronics with some real components instead of a rubber band and a paper clip? The Mintronics: Survival Pack from the Maker Shed contains over 60 useful components for making, hacking, and modifying electronic circuits and repairs on the go.

MakerShield Kit



The ultimate open source prototyping shield for Arduino & compatible microcontrollers. Create your circuits the way you want, and easily changes without having to solder.

Arduino Uno



Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer!

Make:
Maker SHED
DIY KITS + TOOLS + BOOKS + FUN
makershed.com