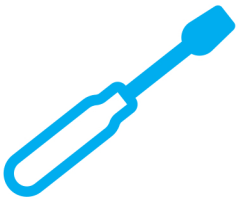
Practical methods for connecting physical objects.

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

# Making Things Talk

*Second Edition*

Tom Igoe

MAKER MEDIA
SEBASTOPOL, CA

# Making Things Talk

## by Tom Igoe



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



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

# Contents

# Preface

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

# 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.

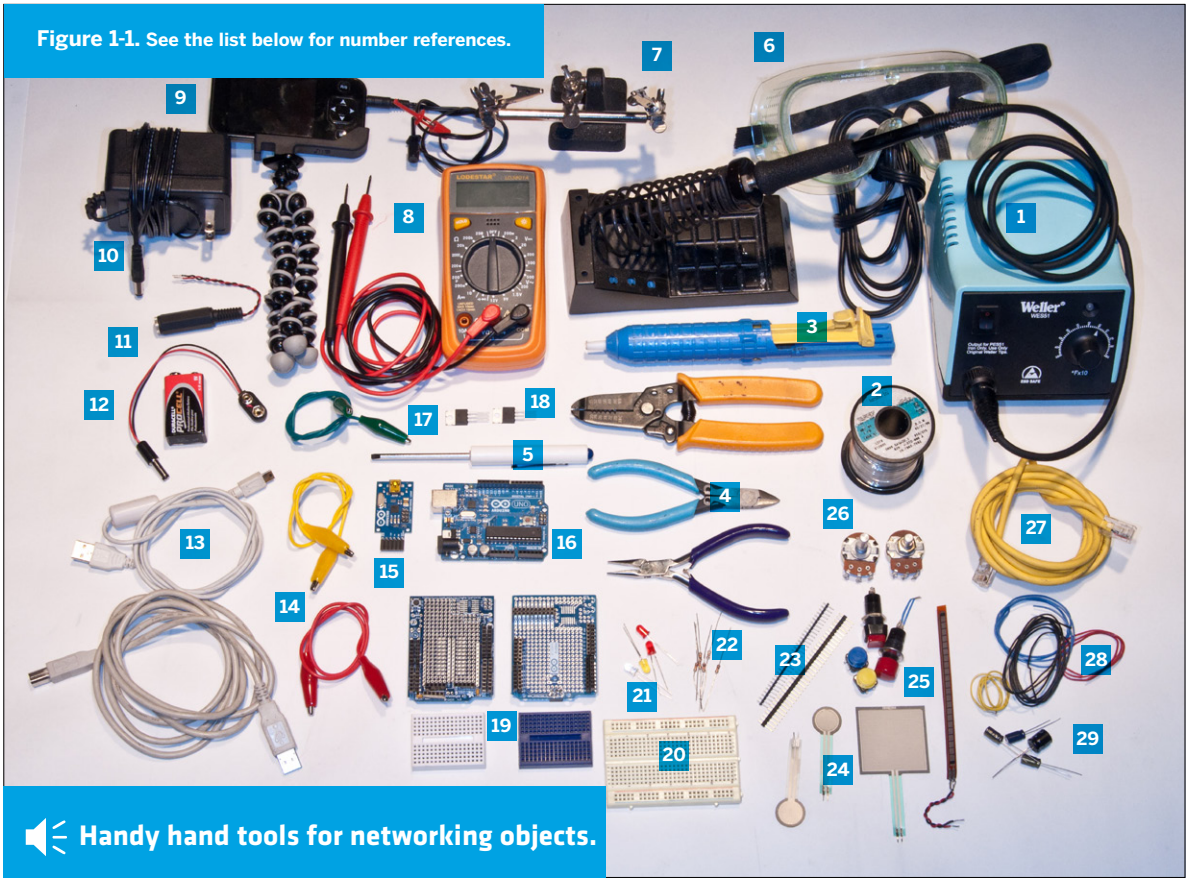
## Physical Tools

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

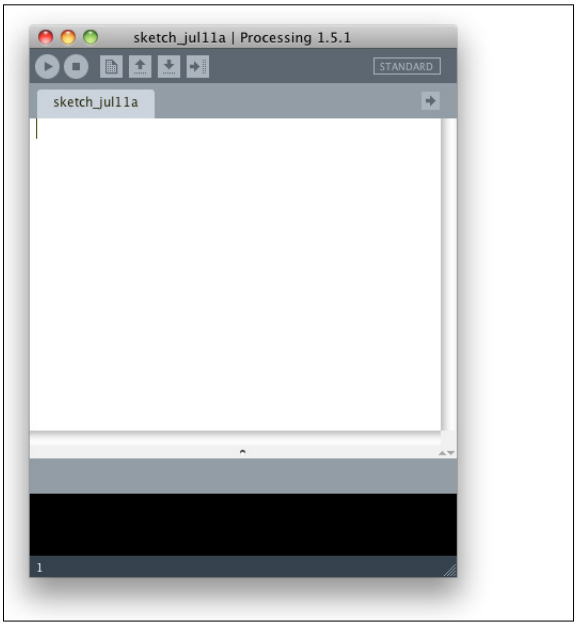



**Figure 1-2**
The Processing editor window.

# 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

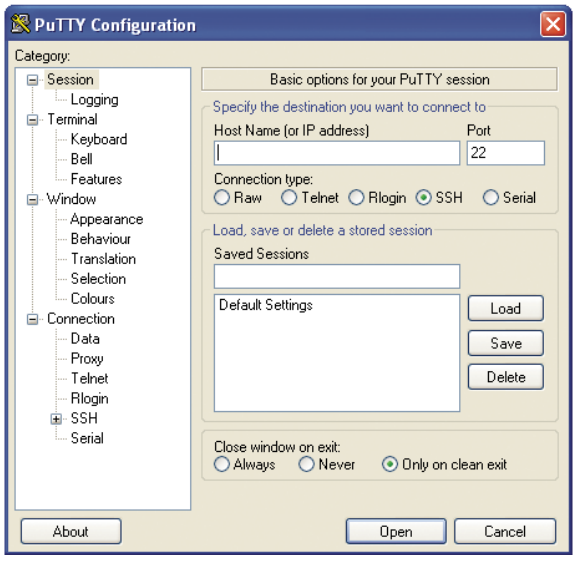



**Figure 1-3**
The main PuTTY window.

### 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
```

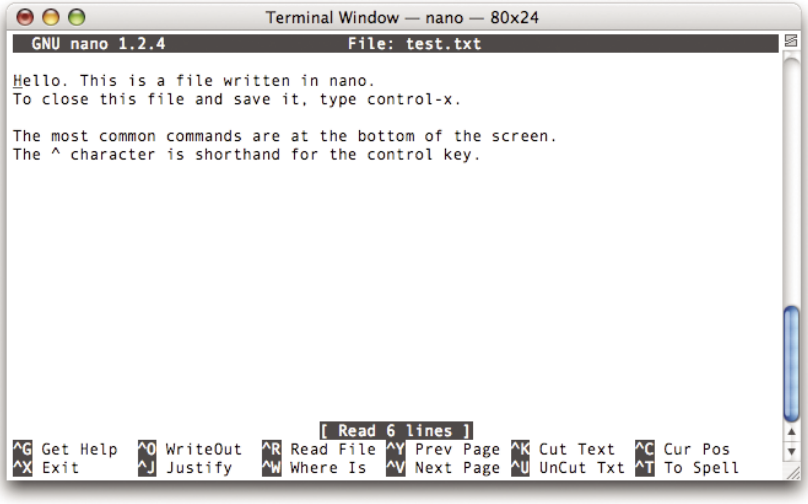



**Figure 1-4**
The nano text editor.

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

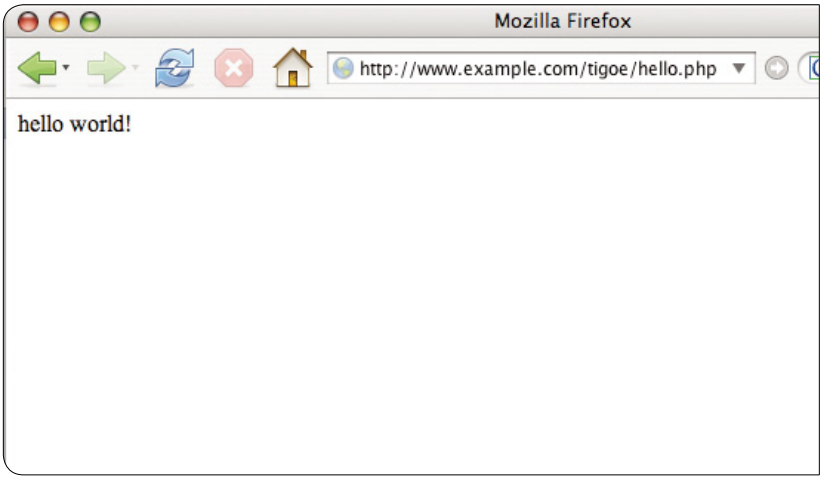



**Figure 1-5**
The results of your first PHP script, in a browser.

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

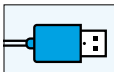
To get started, you'll need to know the serial port name. Click Start→Run (use the Search box on Windows 7), type devmgmt.msc, and press Enter to launch Device Manager. If you've got a serial device such as a Wiring or Arduino board attached, you'll see a listing for Ports (COM & LPT). Under that listing,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

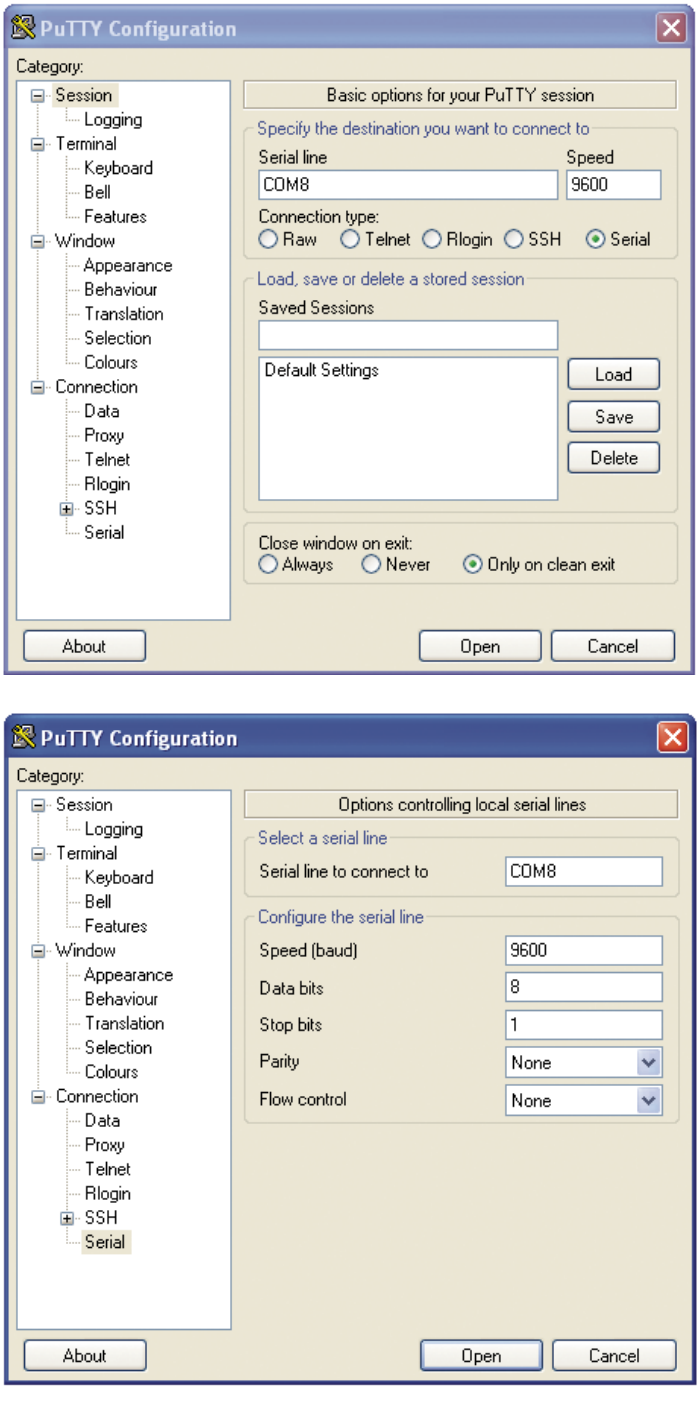



**Figure 1-6**
Configuring a serial connection in PuTTY.

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.

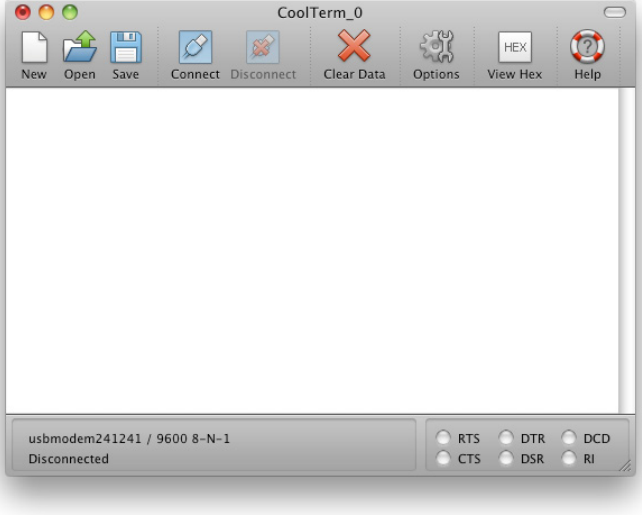



**Figure 1-7**
The CoolTerm serial terminal program.

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**

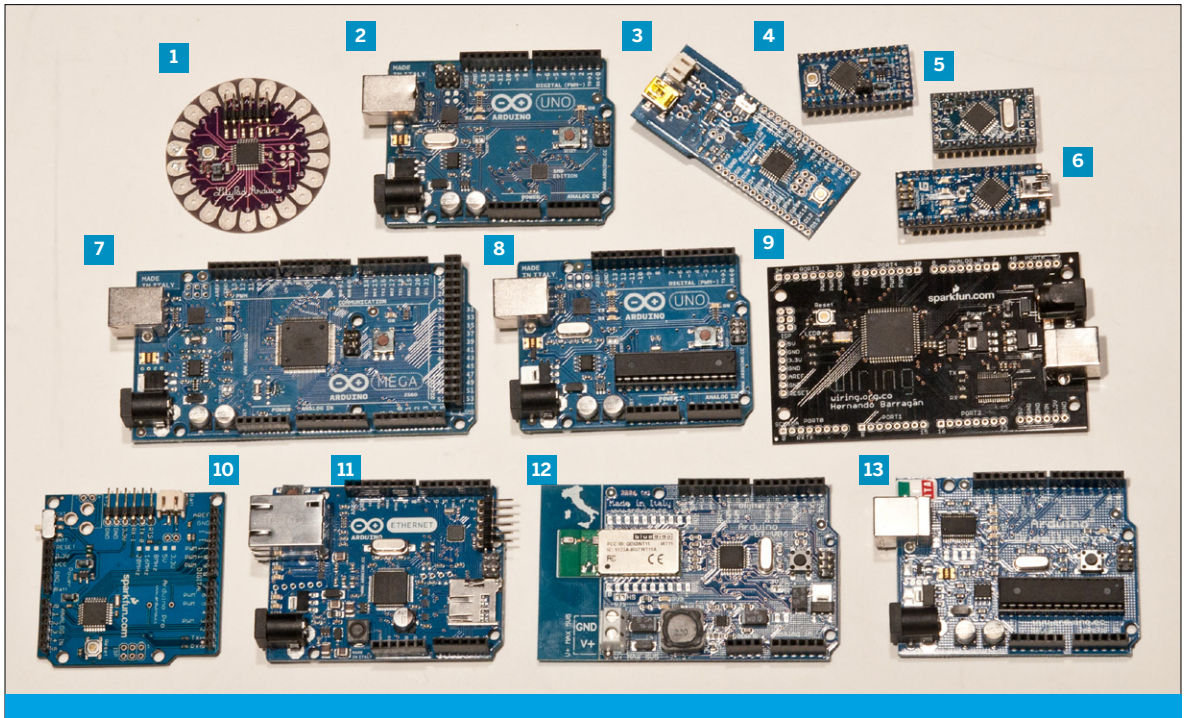



**Figure 1-8.** Varieties of Arduino, as well as a Wiring board: **1.** LilyPad Arduino **2.** Arduino Uno SMD **3.** Arduino Fio **4.** Arduino Pro Mini **5.** Arduino Mini **6.** Arduino Nano **7.** Arduino Mega 2560 **8.** Arduino Uno **9.** Wiring board **10.** Arduino Pro **11.** Arduino Ethernet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

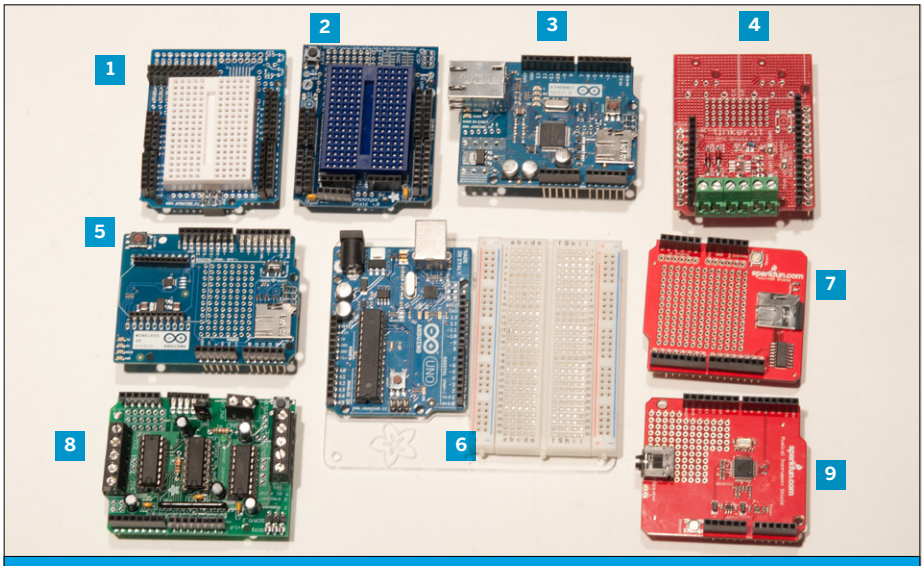



**Figure 1-9.** A sampling of shields for Arduino: **1.** Arduino prototyping shield **2.** Adafruit prototyping shield **3.** Arduino Ethernet shield **4.** TinkerKit DMX shield **5.** Arduino wireless shield **6.** Oomlout Arduino/breadboard mount, manufactured by Adafruit **7.** Spark Fun microSD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

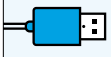
The growth of shields has been a major factor in the spread of Arduino, and the well-designed and documented ones make it possible to build many projects with no electronic experience whatsoever. You'll be using some shields in this book, and for other projects, building the actual circuit yourself.

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.

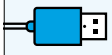
Though the microcontrollers themselves are cheap (between $1 and $10 apiece), getting all the tools set up for yourself can cost you some money. There's also a pretty significant time investment in getting set up, as the tools for programming these controllers from scratch assume a level of technical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

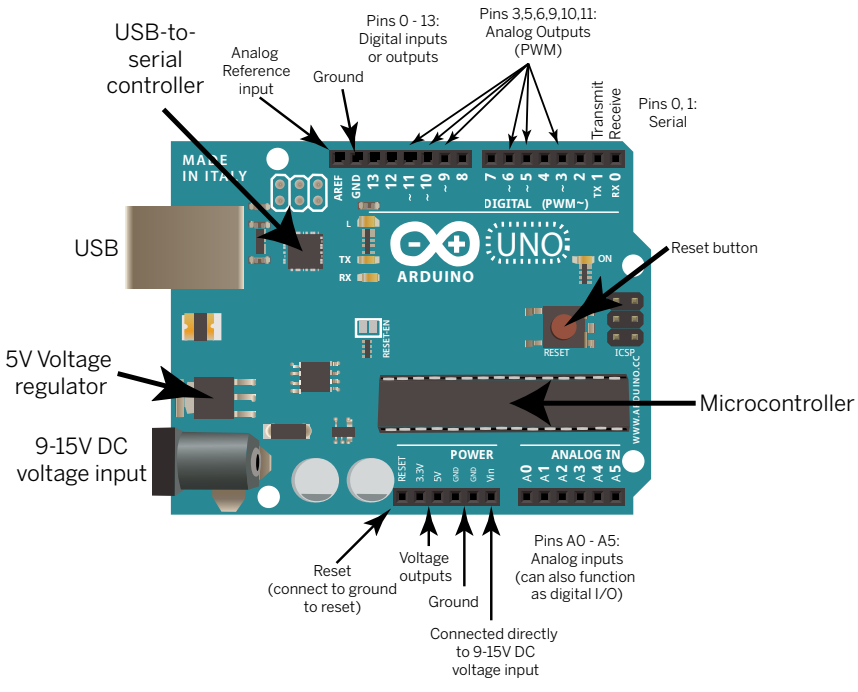



**Figure 1-10**
Functional parts of an Arduino. Most microcontrollers have the same or similar parts: power connections, digital and analog inputs, and serial communications.

## 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).

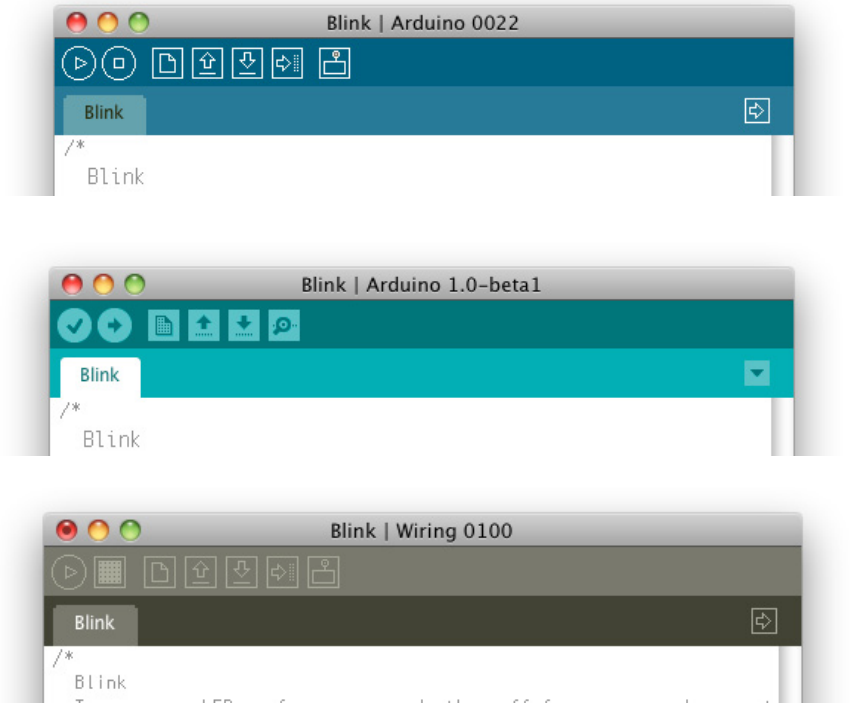



**Figure 1-11**
Toolbars for Arduino version 0022, Arduino 1.0, and Wiring 1.0.

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

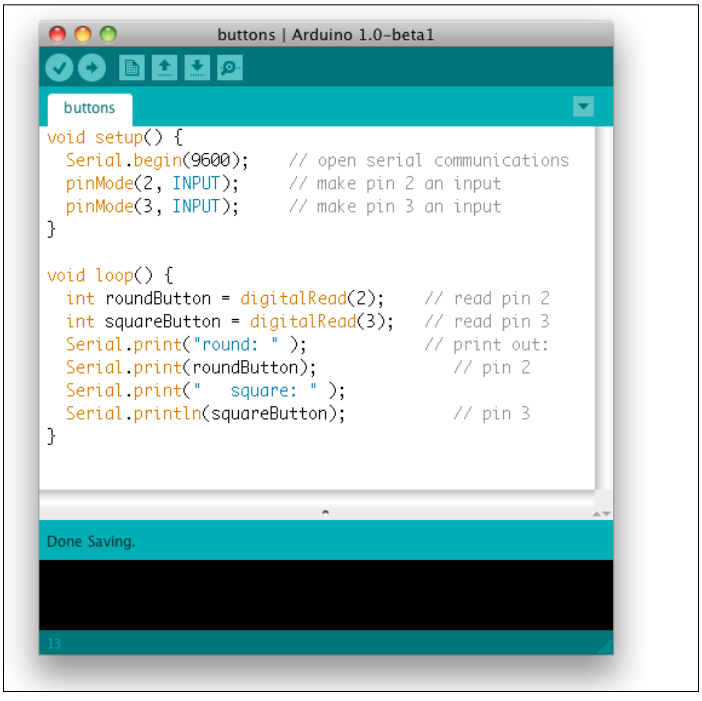



**Figure 1-12**
The Arduino programming environment. The Wiring environment looks similar, except the color is different.

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

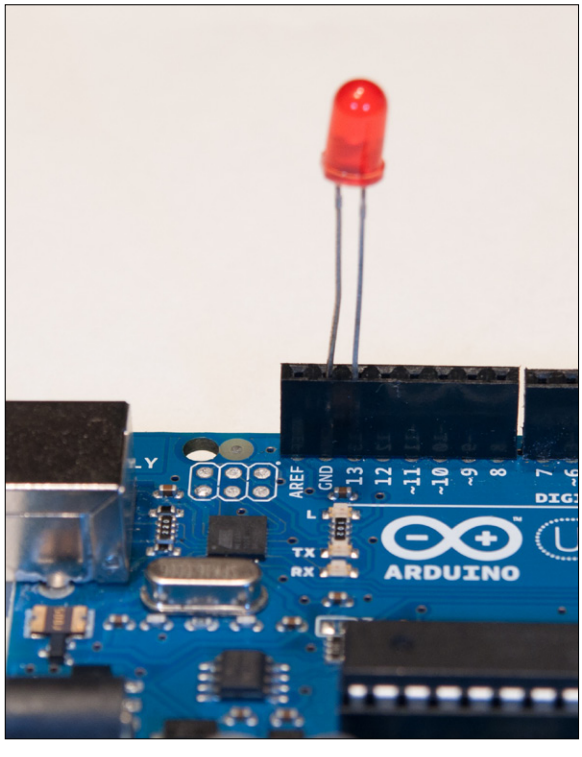



**Figure 1-13**
LED connected to pin 13 of an Arduino board. Add 220-ohm current-limiting resistor in series with this if you plan to run it for more than a few minutes.

## 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)
```

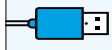
Once the sketch is uploaded, the LED you wired to the output pin will begin to blink. That's the microcontroller equivalent of "Hello World!"

**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.

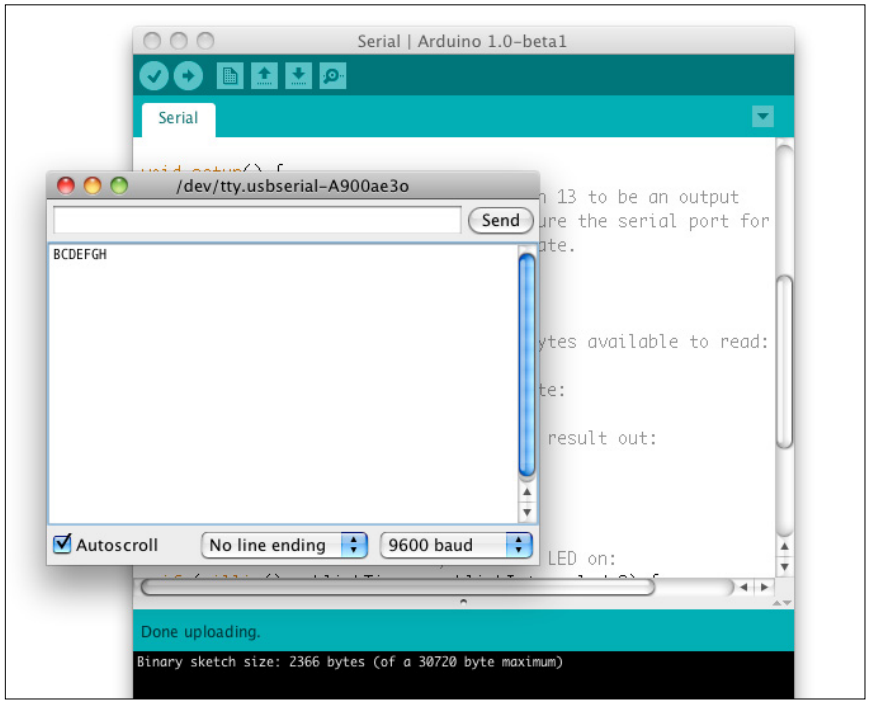



**Figure 1-14**
The Serial monitor in Arduino, running the previous sketch. The user typed BCDEFGH.

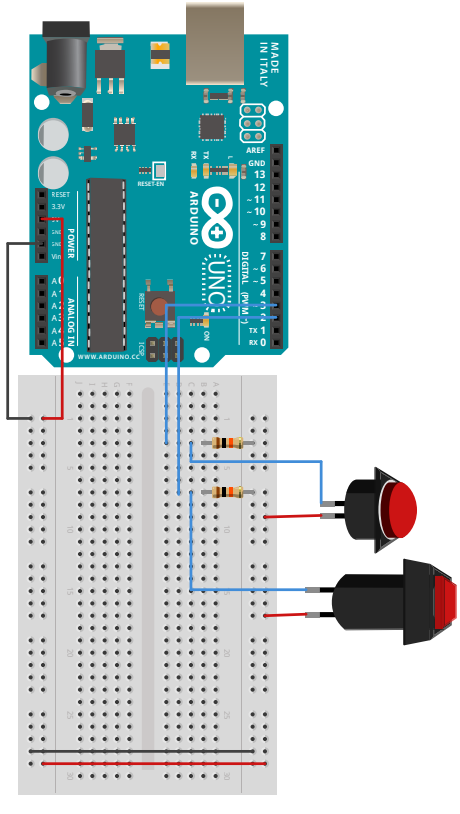


**Figure 1-15**
Arduino connected to a breadboard. +5V and ground run from the module to the long rows of the board. This way, all sensors and actuators can share the +5V and ground connections of the board. Control or signal connections from each sensor or actuator run to the appropriate I/O pins. In 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.**

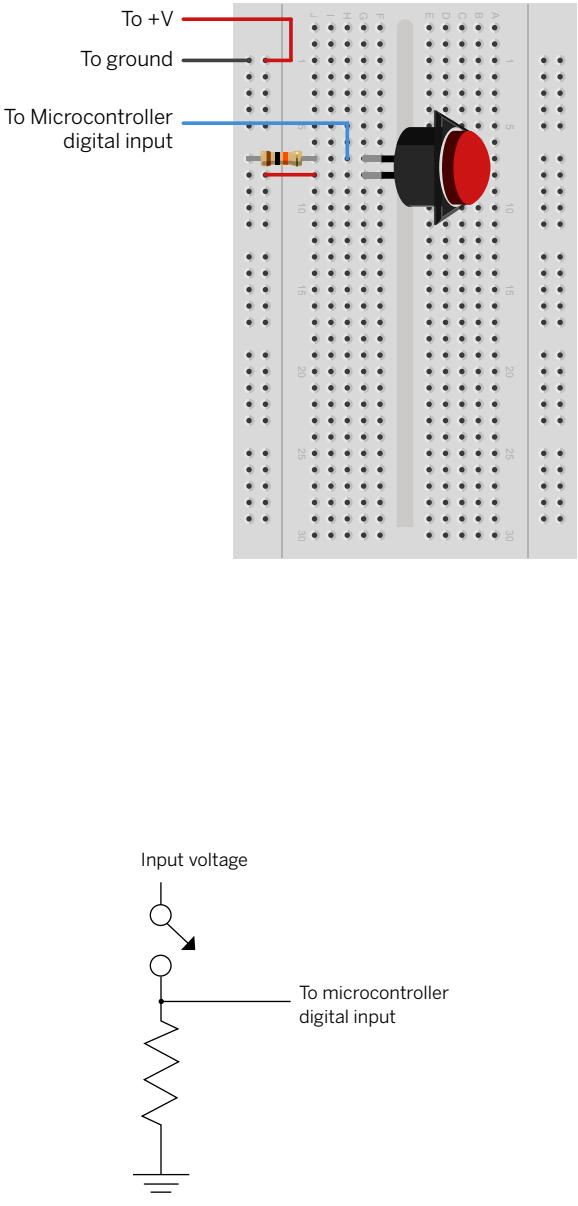
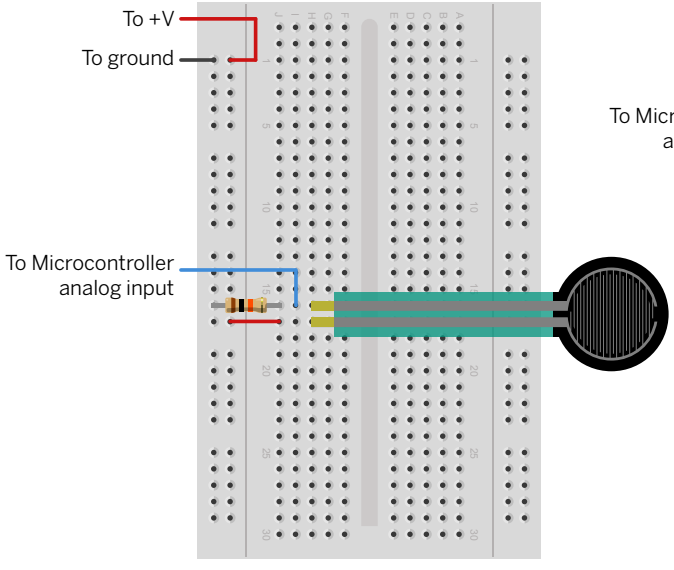



**Figure 1-16**
Digital input to a microcontroller.
*Top*: breadboard view.
*Bottom*: schematic view.



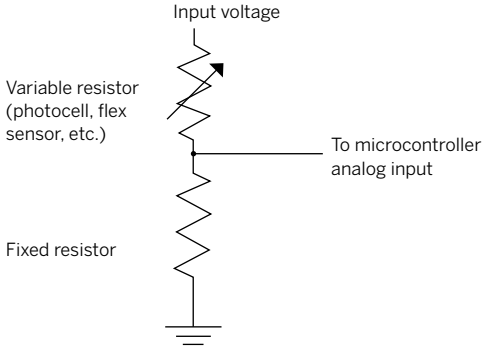
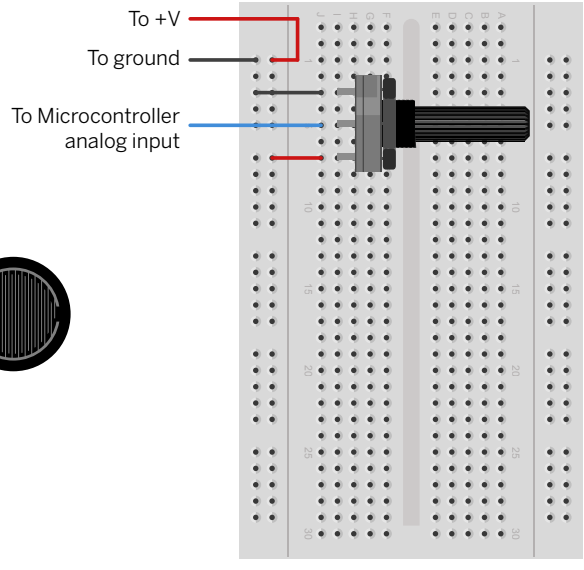




**Figure 1-17**
Voltage divider used as analog input to a microcontroller.
*Top*: breadboard view.
*Bottom*: schematic view.



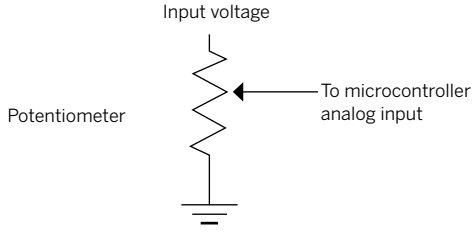




**Figure 1-18**
Potentiometer used as analog input to a microcontroller.
*Top*: breadboard view.
*Bottom*: schematic view.

# 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.

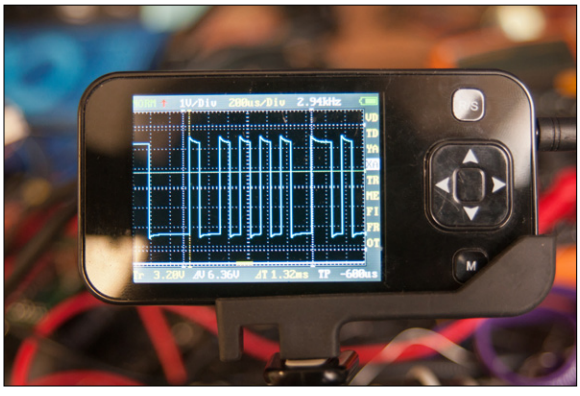


**Figure 1-19**
DSO Nano oscilloscope reading a serial data stream.

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**