

LEARNING MADE EASY

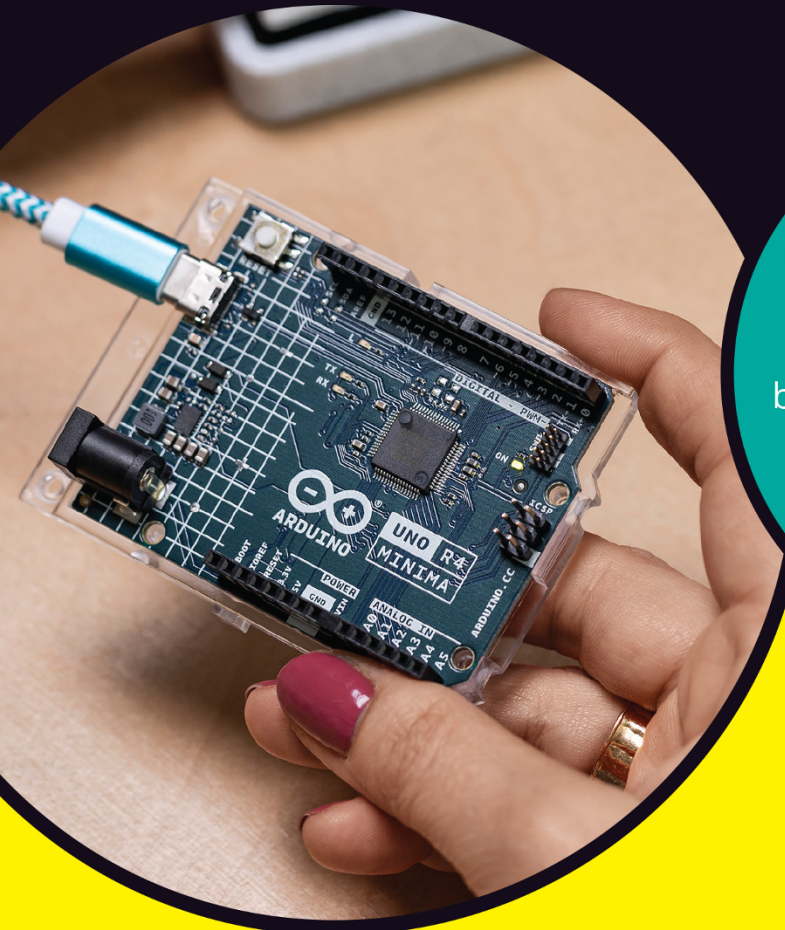


3rd Edition

# Arduino<sup>®</sup>

for  
**dummies<sup>®</sup>**

A Wiley Brand



Bring your creative,  
interactive inventions to life

Designed for complete  
beginners and curious minds

Start building with  
Arduino quickly

**John Nussey**

Hardware designer and educator,  
never far from a soldering iron



# Arduino®

for  
**dummies**®  
A Wiley Brand





# Arduino<sup>®</sup>

3rd Edition

**by John Nussey**

for  
**dummies**<sup>®</sup>  
A Wiley Brand

## Arduino® For Dummies®, 3rd Edition

Published by: **John Wiley & Sons, Inc.**, 111 River Street, Hoboken, NJ 07030-5774, [www.wiley.com](http://www.wiley.com)

Copyright © 2026 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial technologies or similar technologies.

Media and software compilation copyright © 2026 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial technologies or similar technologies.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at [www.copyright.com](http://www.copyright.com). Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at [www.wiley.com/go/permissions](http://www.wiley.com/go/permissions).

The manufacturer's authorized representative according to the EU General Product Safety Regulation is Wiley-VCH GmbH, Boschstr. 12, 69469 Weinheim, Germany, e-mail: [Product\\_Safety@wiley.com](mailto:Product_Safety@wiley.com).

**Trademarks:** Wiley, For Dummies, the Dummies Man logo, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and may not be used without written permission. Arduino is a registered trademark of Arduino SA. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit <https://hub.wiley.com/community/support/dummies>.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit [www.wiley.com](http://www.wiley.com).

Library of Congress Control Number is available from the publisher.

ISBN 978-1-394-36959-1 (pbk); ISBN 978-1-394-36961-4 (ebk); ISBN 978-1-394-36960-7 (ebk)

# Contents at a Glance

---

<b>Introduction</b> .....	1
<b>Part 1: Getting to Know Arduino</b> .....	5
CHAPTER 1: Discovering Arduino .....	7
CHAPTER 2: Finding Your Board and Your Way Around It .....	17
CHAPTER 3: Blinking an LED .....	39
<b>Part 2: Getting Physical with Arduino</b> .....	55
CHAPTER 4: Tools of the Trade .....	57
CHAPTER 5: A Primer on Electricity and Circuitry .....	69
CHAPTER 6: Basic Sketches: Inputs, Outputs, and Communication .....	85
CHAPTER 7: More Basic Sketches: Motion and Sound .....	115
<b>Part 3: Building on the Basics</b> .....	149
CHAPTER 8: Learning by Example .....	151
CHAPTER 9: Soldering On .....	163
CHAPTER 10: Getting Clever with Code .....	193
CHAPTER 11: Common Sense with Common Sensors .....	225
CHAPTER 12: Becoming a Specialist with Shields and Libraries .....	255
<b>Part 4: Sussing Out Software</b> .....	273
CHAPTER 13: Getting to Know Processing .....	275
CHAPTER 14: Processing the Physical World .....	293
<b>Part 5: The Part of Tens</b> .....	325
CHAPTER 15: Ten Places to Learn More about Arduino .....	327
CHAPTER 16: Ten Great Shops to Know .....	331
<b>Index</b> .....	335



# Table of Contents

<b>INTRODUCTION</b> .....	1
About This Book .....	1
Foolish Assumptions .....	2
Icons Used in This Book .....	2
Beyond the Book .....	3
Where to Go from Here .....	3
<b>PART 1: GETTING TO KNOW ARDUINO</b> .....	5
<b>CHAPTER 1: Discovering Arduino</b> .....	7
Where Did Arduino Come From? .....	8
Learning by Doing .....	11
Patching .....	11
Hacking .....	12
Circuit bending .....	13
Electronics .....	14
Inputs .....	14
Outputs .....	14
Open Source .....	15
<b>CHAPTER 2: Finding Your Board and Your Way Around It</b> .....	17
Getting to Know the Arduino Uno .....	18
The Brains: Renesas RA4M1 microcontroller .....	19
Header sockets .....	19
Special pin features .....	20
USB and Power .....	20
Built-in LEDs .....	21
Reset button .....	21
Discovering Other Arduino Boards .....	21
Official Arduino boards .....	22
Arduino-compatible boards .....	24
Shopping for Arduino .....	25
Arduino Store .....	25
Adafruit .....	25
SparkFun .....	26
Seeed Studio .....	26
Pimoroni .....	26
Watterott Electronic .....	26
RobotShop .....	26
Electronics distributors .....	27
Amazon .....	27
eBay .....	27

Kitted Out: Starting with a Beginner’s Kit . . . . .	27
Essentials . . . . .	28
Nice to have . . . . .	29
Preparing a Workspace . . . . .	30
Installing Arduino . . . . .	31
Using Arduino Cloud Editor . . . . .	32
Surveying the Arduino Environment . . . . .	32
Using AI Wisely in Your Arduino Projects . . . . .	34
How AI can help . . . . .	35
Where AI falls short . . . . .	36
A healthy way to use AI with Arduino . . . . .	37
Working smart with AI . . . . .	38
<b>CHAPTER 3: Blinking an LED . . . . .</b>	<b>39</b>
Working with Your First Arduino Sketch . . . . .	40
Finding the Blink sketch . . . . .	40
Identifying your board . . . . .	41
Configuring the software . . . . .	41
Uploading the sketch . . . . .	43
Congratulate yourself! . . . . .	45
What just happened? . . . . .	45
Looking Closer at the Sketch . . . . .	45
Comments . . . . .	47
Functions . . . . .	48
Setup . . . . .	49
Loop . . . . .	51
Blinking Brighter . . . . .	52
Tweaking the Sketch . . . . .	54
<b>PART 2: GETTING PHYSICAL WITH ARDUINO . . . . .</b>	<b>55</b>
<b>CHAPTER 4: Tools of the Trade . . . . .</b>	<b>57</b>
Finding the Right Tools for the Job . . . . .	57
Breadboard . . . . .	58
Jump wires . . . . .	60
Needle-nose pliers . . . . .	61
Multimeter . . . . .	62
Using a Multimeter to Measure Voltage, Current, and Resistance . . . . .	63
Measuring voltage in a circuit . . . . .	64
Measuring current in a circuit . . . . .	65
Measuring resistance of a resistor . . . . .	65
Measuring resistance of a variable resistor . . . . .	66
Checking the continuity (in beeps) of your circuit . . . . .	67

<b>CHAPTER 5: A Primer on Electricity and Circuitry</b> .....	69
Understanding Electricity .....	69
Using Equations to Build Your Circuits .....	71
Ohm's Law .....	71
Calculating power .....	74
Joule's Law .....	75
Working with Circuit Diagrams .....	76
A simple circuit diagram .....	76
Using a circuit diagram with an Arduino .....	78
Color-Coding .....	80
Datasheets .....	81
Resistor Color Charts .....	82
<b>CHAPTER 6: Basic Sketches: Inputs, Outputs, and Communication</b> .....	85
Uploading a Sketch .....	86
Using Pulse-Width Modulation (PWM) .....	87
The LED Fade Sketch .....	87
Understanding the Fade sketch .....	91
Declarations .....	92
Variables .....	92
Tweaking the Fade sketch .....	94
The Button Sketch .....	97
Understanding the Button sketch .....	100
Tweaking the Button sketch .....	101
The AnalogInput Sketch .....	102
Understanding the AnalogInput sketch .....	106
Tweaking the AnalogInput sketch .....	107
Talking Serial .....	108
The AnalogInOutSerial sketch .....	109
Understanding the AnalogInOutSerial sketch .....	112
<b>CHAPTER 7: More Basic Sketches: Motion and Sound</b> .....	115
Working with Electric Motors .....	115
Discovering Diodes .....	117
Spinning a DC Motor .....	117
The Motor sketch .....	118
Understanding the Motor sketch .....	122
Controlling the Speed of Your Motor .....	122
The MotorControl sketch .....	123
Understanding the MotorControl sketch .....	125
Tweaking the MotorControl sketch .....	125
Getting to Know Servomotors .....	126

Creating Sweeping Movements.....	127
The Sweep sketch .....	127
Understanding the Sweep sketch.....	130
Controlling Your Servo .....	131
The Knob sketch .....	132
Understanding the Knob sketch .....	134
Making Noises .....	136
Piezo buzzer .....	136
The toneMelody sketch .....	136
Understanding the sketch .....	140
Making an Instrument .....	143
The PitchFollower sketch .....	144
Understanding the sketch .....	146

**PART 3: BUILDING ON THE BASICS .....** 149

**CHAPTER 8: Learning by Example .....** 151

Chorus.....	151
How it works.....	152
Lessons learned.....	152
Build your own.....	153
Further reading .....	153
Push Snowboarding .....	153
How it works.....	153
Lessons learned.....	154
Build your own.....	154
Further reading .....	155
Baker Tweet .....	155
How it works.....	155
Lessons learned.....	156
Build your own.....	156
Further reading .....	156
Interactive Plan Chests and Compass Cards.....	156
How it works.....	157
Lessons learned.....	157
Build your own.....	158
Further reading .....	158
The Good Night Lamp.....	158
How it works.....	159
Lessons learned.....	159
Build your own.....	159
Further reading .....	160
Little Printer .....	160
How it works.....	161
Lessons learned.....	161

Build your own.....	161
Further reading.....	161
What These Projects Teach You.....	161
Your Turn.....	162
Where to Go Next.....	162
<b>CHAPTER 9: Soldering On.....</b>	<b>163</b>
Understanding Soldering.....	163
Gathering What You Need for Soldering.....	164
Creating a workspace.....	164
Choosing a soldering iron.....	165
Solder.....	170
Third hand (helping hand).....	171
Adhesive putty.....	172
Wire cutters.....	173
Wire strippers.....	173
Needle-nose pliers.....	174
Multimeter.....	174
Solder sucker.....	174
Solder wick.....	175
Equipment wire.....	176
Staying Safe while Soldering.....	177
Handling your soldering iron.....	177
Keeping your eyes protected.....	177
Working in a ventilated environment.....	178
Cleaning your iron.....	178
Don't eat the solder!.....	178
Assembling a Shield.....	178
Laying out all the pieces of the circuit.....	179
Assembly.....	181
Header pins.....	181
Acquiring Your Soldering Technique.....	182
Building Your Circuit.....	186
Knowing your circuit.....	186
Laying out your circuit.....	187
Preparing your wire.....	187
Soldering your circuit.....	187
Cleaning up.....	188
Testing your shield.....	188
Packaging Your Project.....	189
Enclosures.....	189
Wiring.....	190
Securing the board and other elements.....	191

<b>CHAPTER 10: Getting Clever with Code</b> .....	193
Blinking Better .....	193
Setting up the BlinkWithoutDelay sketch .....	195
Understanding the BlinkWithoutDelay sketch .....	198
Taking the Bounce Out of Your Button .....	200
Setting up the Debounce sketch .....	200
Understanding the Debounce sketch .....	204
Making a Better Button .....	206
Setting up the StateChangeDetection sketch .....	206
Understanding the StateChangeDetection sketch .....	209
Smoothing Your Sensors .....	212
Setting up the Smoothing sketch .....	213
Understanding the Smoothing sketch .....	216
Calibrating Your Inputs .....	218
Setting up the Calibration sketch .....	219
Understanding the Calibration sketch .....	222
<b>CHAPTER 11: Common Sense with Common Sensors</b> .....	225
Making Buttons Easier .....	226
Implementing the DigitalInputPullup sketch .....	227
Understanding the DigitalInputPullup sketch .....	230
Exploring Piezo Sensors .....	231
Implementing the Knock sketch .....	232
Understanding the Knock sketch .....	235
Utilizing Pressure, Force, and Load Sensors .....	237
Implementing the toneKeyboard sketch .....	239
Understanding the toneKeyboard sketch .....	242
Sensing with Style .....	243
Implementing the CapacitiveTouch sketch .....	245
Understanding the CapacitiveTouch sketch .....	247
Measuring Distance .....	249
Implementing the HC-SR04 sketch .....	250
Understanding the MaxSonar sketch .....	252
Taking Sensors Further .....	254
<b>CHAPTER 12: Becoming a Specialist with Shields and Libraries</b> .....	255
Looking at Shields .....	256
Considering combinations .....	256
Reviewing the field .....	257
Prototyping shields .....	258
Audio & display shields .....	259
Input shields .....	261
Motor & power shields, navigating and sensing shields .....	262

Legacy shields .....	263
Staying current.....	265
Browsing the Libraries .....	266
Reviewing the standard libraries.....	267
Installing additional libraries .....	269
Installing a library from a ZIP file.....	270
Further reading .....	271
<b>PART 4: SUSSING OUT SOFTWARE .....</b>	<b>273</b>
<b>CHAPTER 13: Getting to Know Processing .....</b>	<b>275</b>
Looking Under the Hood .....	276
Exploring Other Creative Coding Tools .....	277
Installing Processing .....	278
Taking a look at Processing .....	279
Trying Your First Processing Sketch .....	280
Drawing shapes .....	282
Changing color and opacity .....	287
Playing with interaction .....	290
<b>CHAPTER 14: Processing the Physical World .....</b>	<b>293</b>
Making a Virtual Button .....	293
Setting up the Arduino code .....	294
Setting up the Processing code .....	296
Understanding the Processing PhysicalPixel sketch .....	299
Understanding the Arduino PhysicalPixel sketch .....	303
Drawing a Graph .....	305
Setting up the Arduino code .....	306
Setting up the Processing code .....	307
Understanding the Arduino Graph sketch .....	310
Understanding the Processing Graph sketch .....	310
Sending Multiple Signals .....	314
Setting up the Arduino code .....	315
Setting up the Processing code .....	317
Understanding the Arduino SerialCallResponse sketch .....	320
Understanding the Processing SerialCallResponse sketch .....	321
<b>PART 5: THE PART OF TENS.....</b>	<b>325</b>
<b>CHAPTER 15: Ten Places to Learn More about Arduino .....</b>	<b>327</b>
Arduino Blog.....	327
Arduino Project Hub .....	328
Arduino Forum.....	328
Adafruit.....	328
SparkFun.....	328

	Hackaday .....	329
	YouTube .....	329
	Instructables.....	329
	Reddit and Discord Communities.....	329
	Maker Spaces, FabLabs, and Workshops.....	330
<b>CHAPTER 16:</b>	<b>Ten Great Shops to Know .....</b>	<b>331</b>
	Arduino Store.....	331
	Adafruit.....	332
	SparkFun.....	332
	Seeed Studio .....	332
	Pimoroni.....	332
	Digi-Key.....	333
	RS Components .....	333
	Tindie.....	333
	AliExpress .....	334
	Treasure Hunting.....	334
<b>INDEX.....</b>		<b>335</b>

# Introduction

---

**A**rduino is a tool, a community, and a way of thinking that is affecting how we use and understand technology. It has rekindled a love and understanding of electronics for many people, including myself, who felt that electronics was something that they had left behind at school.

Arduino is tiny circuit board that has huge potential. It can be used to blink a Morse-code signal using a single light-emitting diode (LED) or to control every light in a building, depending on how far you take it. Its capabilities are limited only by your imagination.

Arduino is also providing a new, practical approach to technical education, lowering the entry level for those wanting to use electronics to complete small projects and, I hope, encouraging you to read further to take on big ones.

A huge and ever-growing community of Arduinists has emerged — users and developers who learn from each other and contribute to the open-source philosophy by sharing the details of their projects. This open-source attitude is responsible for the huge popularity of Arduino.

Arduino is more than just a gadget; it's a tool. A piece of technology that makes understanding and using today's technology easier.

So if the prospect of understanding the limitless possibilities of technology doesn't sound interesting to you, please put this book down and back away.

Otherwise, read on!

## About This Book

---

This is a technical book, but it's not for technical people only. Arduino is designed to be usable by anyone, whether they're technical, creative, crafty, or just curious. All you need is an open mind or a problem to fix and you'll soon find ways that using Arduino can benefit you.

This book starts on the most basic level to get you started with using and understanding Arduino. At times throughout the book, I may refer to a number of technical things that will, like anything, take time to understand. I guide you through all the basics and then on to more advanced activities.

Much of what is in this book is based on my learning and teaching experiences. I learned all about Arduino from scratch, but have always found that the best way to learn is in practice, by making your own projects. The key is to understand the basics that I cover in this book and then build on that knowledge by thinking about how you can apply it to solve problems, create things, or just entertain yourself.

## Foolish Assumptions

I assume nothing about your technical knowledge. Arduino is an easy-to-use platform for learning about electronics and programming. It is for people from all walks of life, whether you're a designer, an artist, or a hobbyist.

It can also be a great platform for people who are already technical. Maybe you've done a bit of coding but want to bring your projects into the physical world in some way. Or maybe you've worked with electronics and want to see what Arduino can bring to the table.

Whoever you are, you'll find that Arduino has great potential. It's really up to you to decide what to make of it.

## Icons Used in This Book

*Arduino For Dummies* uses icons to highlight important points for you. Keep an eye out for the following:



TIP

This icon highlights a bit of helpful information. That info may be a technique to help you complete a project more easily or the answer to a common problem.



WARNING

Arduinos aren't dangerous on their own; indeed, they're made to be extremely safe and easy to use. But if you use them in a circuit without proper planning as well as care and attention, they can damage your circuit, your computer, and yourself. When you see a Warning icon, please take special note.



REMEMBER

Often, you must consider certain points before proceeding with a task. I use Remember icons to remind you of such points.



TECHNICAL  
STUFF

Some information is more technical than others and is not for the faint-hearted. The joy of Arduino is that you don't need to fully understand the technical details immediately. You can skip anything that's marked with this icon if it's more complicated than you want to deal with at the moment; you can always return to it when you're ready.

## Beyond the Book

In addition to what you're reading right now, this product comes with a free access-anywhere Cheat Sheet that provides information on using resistors, getting the tools you'll need, and some system shortcuts. To get this Cheat Sheet, simply go to [www.dummies.com](http://www.dummies.com) and type *Arduino For Dummies 3rd Edition Cheat Sheet* in the Search box. I also provide a bonus chapter that teaches you all about using your Arduino to hack other hardware, such as games, controllers, and toys.

## Where to Go from Here

If you're uncertain about where to start, I suggest the beginning. By the end of Chapter 2, you'll have acquired a simple understanding of Arduino and will know where you can get a kit to continue learning.

If you've used Arduino before, you may want to jump straight to Chapter 4 to cover the basics again, or head straight to the area that interests you.



# 1

# Getting to Know Arduino

## **IN THIS PART . . .**

Find out all about the little blue circuit board.

Discover everything you need to get started with Arduino and where to get them.

Learn how to wield the awesome power of an LED, blinking in on command with a few simple lines of code.

- » Discovering what Arduino is
- » Learning where Arduino came from
- » Introducing the basic principles

## Chapter **1**

# Discovering Arduino

**A**rduino is made up of both hardware and software.

The Arduino board is a printed circuit board (PCB) designed to use a microcontroller chip as well as other inputs and outputs. The board has many other electronic components that are needed for the microcontroller to function or to extend its capabilities.

A *microcontroller* is a small computer contained in a single, integrated circuit or computer chip. Microcontrollers are an excellent way to program and control electronics. *Microcontroller boards* have a microcontroller chip and other useful connectors and components that allow a user to attach inputs and outputs. Some examples of devices with microcontroller boards are the Wiring board, the Basic Stamp, and PIC development boards.

You write code in the Arduino software to tell the microcontroller what to do. For example, by writing a line of code, you can tell a light-emitting diode (LED) to blink on and off. If you connect a pushbutton and add another line of code, you can tell the LED to turn on only when the button is pressed. Next, you may want to tell the LED to blink only when the pushbutton is held down. In this way, you can quickly build a behavior for a system that would be difficult to achieve without a microcontroller.

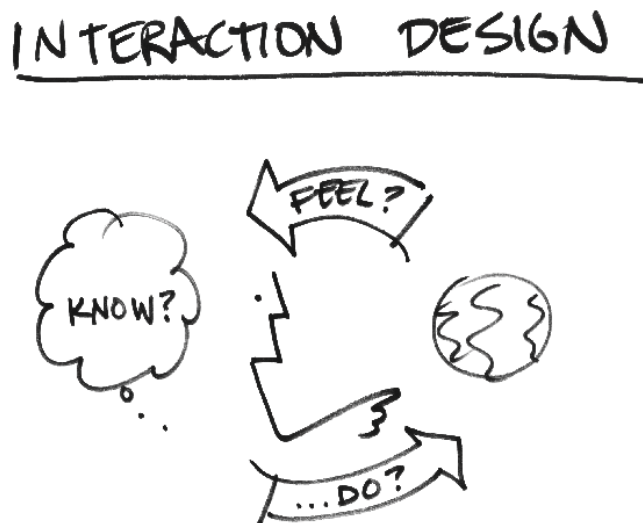
Similar to a conventional computer, an Arduino can perform a multitude of functions, but it's not much use on its own. It requires inputs or outputs to make it useful. These inputs and outputs allow a computer — and an Arduino — to sense objects in the world and to affect the world.

Before you move forward, it might help you to understand a bit of the history of Arduino.

## Where Did Arduino Come From?

Arduino started its life in Italy, at Interaction Design Institute Ivrea (IDII), a graduate school for interaction design that focuses on how people interact with digital products, systems, and environments and how they in turn influence us.

The term *interaction design* was coined by Bill Verplank and Bill Moggridge in the mid-1980s. The sketch in Figure 1-1 by Verplank illustrates the basic premise of interaction design: If you do something, you feel a change, and from that you can know something about the world.



**FIGURE 1-1:**  
The principle of interaction design, illustrated by Bill Verplank.

Courtesy of Bill Verplank

Although interaction design is a general principle, it more commonly refers specifically to how we interact with conventional computers by using peripherals (such as mice, keyboards, and touchscreens) to navigate a digital environment that is graphically displayed on a screen.

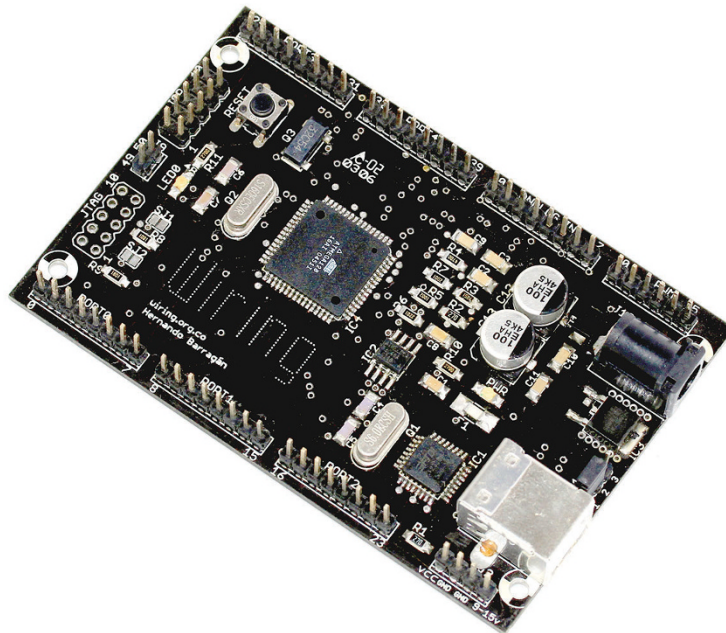
Another avenue, referred to as *physical computing*, is about extending the range of these computer programs, software, or systems through electronics. By using electronics, computers can sense more about the world and have a physical effect on the world themselves.

Both areas — interaction design and physical computing — require prototypes to fully understand and explore the interactions, which presented a hurdle for non-technical design students.

In 2001, a project called Processing, started by Casey Reas and Benjamin Fry, aimed to get non-programmers into programming by making it quick and easy to produce onscreen visualizations and graphics. The project gave the user a digital sketchbook on which to try ideas and experiment with a small investment of time. This project in turn inspired a similar project for experimenting in the physical world.

In 2003, building on the same principles as Processing, Hernando Barragán started developing a microcontroller board called Wiring. This board was the predecessor to Arduino.

In common with the Processing project, the Wiring project also aimed to involve artists, designers, and other non-technical people. However, Wiring was designed to get people into electronics as well as programming. The Wiring board (shown in Figure 1-2) was less expensive than some other microcontrollers, such as the PIC and the Basic Stamp, but it was still a sizable investment for students.



**FIGURE 1-2:**  
An early  
Wiring board.

In 2005, the Arduino project began in response to the need for affordable and easy-to-use devices for interaction design students to use in their projects. It is said that Massimo Banzi and David Cuartielles named the project after Arduin of Ivrea, an Italian king, but I've heard from reliable sources that it also happens to be the name of the local pub near the university, which may have been of more significance to the project.

The Arduino project drew from many of the experiences of both Wiring and Processing. For example, an obvious influence from Processing is the *graphic user interface* (GUI) in the Arduino software. This GUI was initially “borrowed” from Processing, and even though it still looks similar, it has since been refined to be more specific to Arduino. I cover the Arduino interface in more depth in Chapter 3.

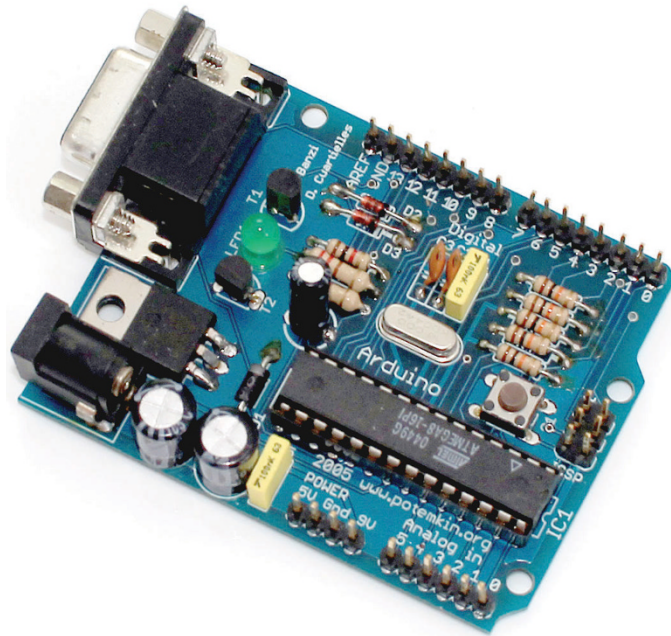
Arduino also kept the naming convention from Processing, calling its programs *sketches*. In the same way that Processing gives people a digital sketchbook to create and test programs quickly, Arduino gives people a way to sketch their hardware ideas as well. Throughout this book, I show many sketches that allow your Arduino to perform a huge variety of tasks. By using and editing the example sketches in this book, you can quickly build up your understanding of how they work. You'll be writing your own in no time. Each sketch is followed with a line-by-line explanation of how it works to ensure that no stone is left unturned.

The Arduino board, shown in Figure 1-3, was made to be more robust and forgiving than Wiring and other earlier microcontrollers. It was not uncommon for students, especially those from a design or arts background, to break their microcontroller within minutes of using it, simply by getting the wires the wrong way around. This fragility was a huge problem, not only financially but also for the success of the boards outside technical circles.

Another important difference between Arduino and other microcontroller boards is the cost. Back in 2006, another popular microcontroller, the Basic Stamp, cost nearly four times as much (\$119) as an Arduino (\$32). Today, an Arduino Uno costs just \$27.

In one of my first Arduino workshops, I was told that the price was intended to be affordable for students. The price of a nice meal and a glass of wine at that time was about \$42, so if you had a project deadline, you could choose to skip a nice meal that week and make your project instead.

The range of Arduino boards on the market is a lot bigger than it was back in 2006. In Chapter 2, you learn about just a few of the most useful Arduino and Arduino-compatible boards and how they differ to provide you with a variety of solutions for your own projects. Also, in Chapter 12, you learn all about a special type of circuit board called a shield, which can add useful, and in some cases phenomenal, features to your Arduino, turning it into a GPS (Global Positioning System) receiver, a mobile phone, or even a Geiger counter, to name just a few.



**FIGURE 1-3:**  
The original  
Arduino  
Serial board.

## Learning by Doing

People have used technology in many ways to achieve their own goals without needing to delve into the details of electronics. Following are just a few related schools of thought that have allowed people to play with electronics.

### Patching

*Patching* is a technique for experimenting with systems using wires. The earliest popular example of patching is in phone switchboards. For an operator to put you through to another line, he or she had to physically attach a cable.

This technique was also popular for synthesizing music, such as with the Moog synthesizer. When an electronic instrument generates a sound, it's really generating a voltage. Different collections of components in the instrument manipulate that voltage before it is outputted as an audible sound. The Moog synthesizer works by changing the path that that voltage takes, sending it through a number of different components to apply different effects.

Because so many combinations are possible, the musician proceeds largely through trial and error. But the simple interface means that this process is extremely quick and requires little preparation to get going.

# Hacking

*Hacking* is a term that typically refers to the subversive use of technology. More generally, though, it refers to exploring systems and making full use of them or repurposing them to suit your needs.

Hacking in this sense is possible in hardware as well as software. A great example of hardware hacking is a keyboard hack. Say that you want to use a big red button to move through a slideshow. Most software programs contain keyboard shortcuts, and most PDF viewers move to the next page in a slideshow when the user presses the spacebar. If you know this, you ideally want a keyboard with only a spacebar.

Today's keyboards have a small circuit board, a bit smaller than a credit card (see Figure 1-4), containing lots of contacts that are connected when you press different keys. If you can find the correct combination, you can connect two contacts by using a pushbutton. Now every time you press that button, you send a space to your computer.



**FIGURE 1-4:**  
The insides of a  
keyboard, ready  
to be hacked.

This technique is great for sidestepping the intricacies of hardware and getting the results you want. In the “Hacking Other Hardware” bonus chapter (<https://www.dummies.com/go/arduinoofd>), you learn more about the joy of hacking and

how you can weave hacked pieces of hardware into your Arduino project to control remote devices, cameras, and even computers with ease.

## Circuit bending

*Circuit bending* flies in the face of traditional education and is all about spontaneous experimentation. Children’s toys are the staple diet of circuit benders, but really any electronic device has the potential to be experimented with.

By opening a toy or device and revealing the circuitry, you can alter the path of the current to affect its behavior. Although this technique is similar to patching, it’s a lot more unpredictable. However, after you find a combination that produces a pleasing result, you can add or replace components, such as resistors or switches, to give the user more control over the instrument.

Most commonly, circuit bending is about sound, and the finished instrument becomes a rudimentary synthesizer or drum machine. Two of the most popular devices are the Speak & Spell (see Figure 1-5) and the Nintendo GameBoy. Musicians, such as the Modified Toy Orchestra ([en.wikipedia.org/wiki/Modified\\_Toy\\_Orchestra](http://en.wikipedia.org/wiki/Modified_Toy_Orchestra)), in their own words, “explore the hidden potential and surplus value latent inside redundant technology.” So think twice before putting your old toys on eBay!



**FIGURE 1-5:**  
A Modified Toy  
Orchestra Speak  
& Spell after  
circuit bending.

*Courtesy of Modified Toy Orchestra*

# Electronics

---

Although there are many ways to work around technology, eventually you'll want more of everything: more precision, more complexity, and more control.

If you learned about electronics at school, you were most likely taught how to build circuits using specific components. These circuits are based solely on the chemical properties of the components and need to be calculated in detail to make sure that the correct amount of current is going to the correct components.

These are the kind of circuits you find as kits at Radio Shack (or Maplin, in the United Kingdom) that do a specific job, such as an egg timer or a security buzzer that goes off when you open a cookie jar. These kits are good at their specific job, but they can't do much else.

This is where microcontrollers come in. When used with analog circuitry, microcontrollers can give that circuitry a more advanced behavior. They can also be reprogrammed to perform different functions as needed. Your Arduino is designed around one of these microcontrollers, and in Chapter 2, you look closely at an Arduino Uno to see exactly how it is designed and what it is capable of.

The microcontroller is the brains of a system, but it needs other electronic inputs and outputs to either sense or affect things in its environment.

## Inputs

*Inputs* are senses for your Arduino. They tell it what is going on in the world. At its most basic, an input could be a switch, such as a light switch in your home. At the other end of the spectrum, it could be a gyroscope, telling the Arduino the exact direction it's facing in three dimensions. You learn all about basic inputs in Chapter 6, and more about the variety of sensors and when to use them in Chapter 11.

## Outputs

*Outputs* allow your Arduino to affect the real world in some way. An output could be subtle and discreet, such as in the vibration of a cellphone, or it could be a huge visual display on the side of a building that can be seen for miles around. The first sketch in the book walks you through blinking an LED (see Chapter 3). From there, you can go on to controlling an electric motor (Chapter 7) and even controlling an LCD screen (Chapter 12).

# Open Source

---

Open source software, in particular Processing, has had a huge influence on Arduino development. In the world of computer software, *open source* is a philosophy in which people share the *source code*, the human-readable instructions that make up a program, and encourage others to use, remix, and redistribute it, as they like.

Just as the Processing software is open source, so are Arduino software and hardware. This means that the Arduino software and hardware are both released freely to be adapted as needed. You find the same spirit of openness also amongst the community on the Arduino forums.

On the official Arduino forums (<http://forum.arduino.cc/>) and many other ones around the world, people have shared their code, projects, and questions for an informal peer review. This sharing allows all sorts of people, including experienced engineers, talented developers, practiced designers, and innovative artists, to lend their expertise to novices in some or all of these areas. It also provides a means to gauge people's areas of interest, which occasionally filters into the official release of Arduino software or board design with refinements or additions. The Arduino website has an area known as the Playground ([playground.arduino.cc/](http://playground.arduino.cc/)), where people are free to upload their code for the community to use, share, and edit.

This kind of philosophy has encouraged the relatively small community to pool knowledge on forums, blogs, and websites, thereby creating a vast resource for new Arduinists to tap into.

Despite the open-source nature of Arduino, a huge loyalty to Arduino as a brand exists — so much so that there is an Arduino naming convention of adding *-duino* or *-ino* to the name of boards and accessories (much to the disgust of Italian members of the Arduino team)!



#### IN THIS CHAPTER

- » Inspecting the Arduino Uno R4 Minima
- » Discovering other Arduino boards
- » Knowing where to shop for Arduinos
- » Finding the right Arduino kit to get started
- » Setting up a workspace
- » Installing Arduino
- » Using AI in Arduino projects

## Chapter 2

# Finding Your Board and Your Way Around It

The name *Arduino* encompasses a host of concepts. It can refer to an Arduino board, the physical hardware, the Arduino environment — that is, a piece of software that runs on your computer — and, finally, Arduino as a subject in its own right, as in this book: how the hardware and software can be combined with related craft and electronics knowledge to create a tool kit for any situation.

This chapter provides an overview of what you need to get started with Arduino. You may be eager to dive in, so you may want to quickly scan through this chapter, stopping at any areas of uncertainty and referring to them later as needed.

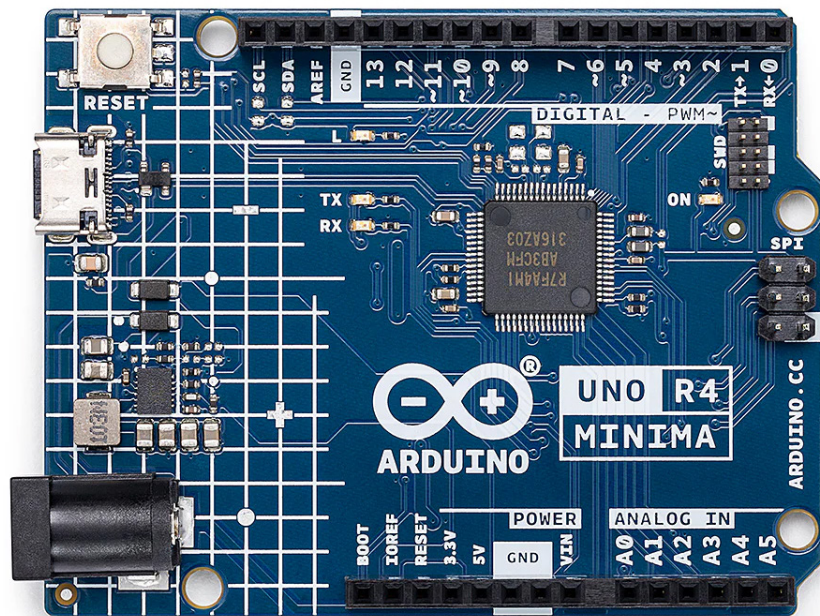
First, you learn about the components used on the Arduino Uno board, which is the starting point for most Arduinists. Beyond that, you get acquainted with the other available Arduino boards, how they differ, and what uses they have. The

chapter lists major suppliers that can equip you with all the parts you need and examines some of the starter kits that are ideal for beginners and for accompanying this book. When you have the kit and a workspace, you're ready to start.

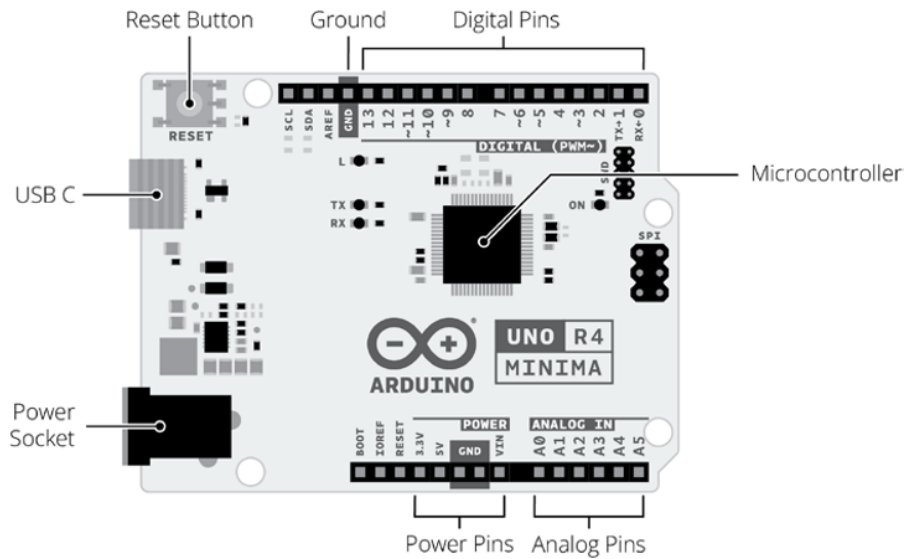
Then you find out where to obtain the software to control your Arduino. I walk you through the steps for downloading and installing the software, and give you a brief tour of the environment in which you develop your Arduino programs. Finally, I discuss the uses of AI in Arduino projects.

## Getting to Know the Arduino Uno

There are many types of Arduino boards, each designed for different applications. But the Arduino Uno is the classic starting point for most people. The latest version, released in 2023, is the Uno R4, available in two models: the Minima and the WiFi. Both are reliable, general-purpose boards, but if you're just starting out, the Uno R4 Minima is the best choice (see Figure 2-1). Take a moment to look at Figure 2-2, which labels the key parts of the board, such as the USB connector, headers, and pins, as you'll be referring to these often.



**FIGURE 2-1:**  
Arduino Uno  
R4 Minima.



**FIGURE 2-2:**  
An Arduino Uno R4 Minima with all the important parts labeled.

*Uno* is Italian for *one*, marking the release of Arduino 1.0 software. Earlier boards had names like *Serial*, *NG*, *Diecimila* (10,000 in Italian, to celebrate a sales milestone), and *Duemilanove* (2009 in Italian, for the release year). The Uno brought consistency to naming, and *R4* marks the fourth revision of the hardware.

## The Brains: Renesas RA4M1 microcontroller

At the heart of the Uno R4 Minima is the Renesas RA4M1 microcontroller — a 32-bit ARM Cortex-M4 chip that’s faster and more capable than the Uno’s previous ATmega328P. It’s the large black square in the center of the board, known as an integrated circuit (IC).



The RA4M1 is soldered directly to the board in a *surface-mount package*, unlike earlier Unos that used removable chips. This doesn’t affect performance — it just means the chip can’t be swapped out.

## Header sockets

The pins around the edge of the board are called header sockets. They connect the microcontroller to the outside world and are divided into three groups:

- » **Digital pins:** For sending or receiving on/off signals.
- » **Analog input pins:** For reading varying voltages from sensors.
- » **Power pins:** For supplying power to your circuit.

These pins make it easy to connect components on a *breadboard*, a reusable board for quickly building and testing simple circuits (see Chapter 6) or to plug in add-on boards called *shields*, which sit on top of the Arduino to add extra features such as motors, displays, or network connections (see Chapter 12).

On the Uno R4 Minima, most pins use 5V logic, although some modern boards use 3.3V, so always check compatibility when switching boards.



Inside every modern computer, signals are also moving between digital and analog forms — but at a level that’s hidden from you. Arduino is special because it bridges this gap directly: It can take signals from the physical world (like voltages from a sensor), convert them into digital data your computer can understand, and then send digital instructions back out as electrical signals. This is what allows you to write simple code that can control circuits and create complex behaviors.

## Special pin features

In addition to the basic digital, analog, and power groups, a few pins have special uses that you’ll often take advantage of.

Several digital pins are marked with a tilde (~), which indicates they support pulse-width modulation (PWM). PWM lets you create outputs that behave like analog signals, such as, for example, dimming an LED or controlling the speed of a motor.

You’ll also see a pin labeled *Vin*, short for *voltage in*. This allows you to power the board with an external voltage source such as a 12V adaptor or battery.

Alongside *Vin* are the regulated 5V and 3.3V pins, which provide clean power to sensors and circuits that require those levels. Finally, the GND pins (ground) are used to complete every circuit. Several ground pins are provided so you can connect multiple components at once.

## USB and Power

To upload programs and power the board from your computer, the Uno R4 Minima uses a USB-C socket. This modern, reversible connector has become the standard on many devices and replaces the older USB A-B cable used on earlier Unos.

There’s also a dedicated power socket next to the USB port. This accepts a 2.1 mm center-positive plug from an adaptor, battery pack, or even a solar panel.



WARNING

Always connect power with the correct polarity. If you accidentally plug in a *center-negative supply*, that's called reverse polarity. The Uno R4 has some protection to stop the voltage going the wrong way, but it can only do so much — and sometimes the protective parts sacrifice themselves in the process (usually with a whiff of burning plastic).

Be especially careful if you're powering the board directly through Vin, 5V, or 3.3V pins, because in those cases the protection is bypassed. A mistake here can destroy the board and the RA4M1 chip almost instantly. The recommended supply range is 6–24V. Too little voltage and your Arduino may not run reliably; too much and it risks overheating or permanent damage.

## Built-in LEDs

The Uno R4 Minima has four built-in LEDs that give you instant feedback about what the board is doing:

- » **ON:** This glows green whenever the board is powered.
- » **RX and TX:** These flash when data is being received or transmitted.
- » **L:** This is connected to digital pin 13, making it a handy built-in test light for your sketches.



TIP

If none of these LEDs light up when the board is connected, check that your USB cable is plugged in, the port is working, and the cable itself is good. If the board still won't power on, visit the Arduino support page at <https://www.arduino.cc/en/contact-us/>.

## Reset button

Next to the USB socket you'll find the reset button. Pressing this button restarts the program currently running on the Arduino. A quick press is enough to reset your sketch, whereas holding it down for longer will stop the program completely. You can also trigger the same effect by briefly connecting the reset pin to ground.

# Discovering Other Arduino Boards

The preceding section describes the standard USB Arduino board, but you should be aware that many others exist, all designed with different needs in mind. Some offer more functionality, and others are designed to be more minimal, but

generally they follow a design similar to that of the Arduino Uno R4 Minima. For this reason, all examples in this book are based on the Uno R4 Minima.

Previous revisions of the Uno should work without any changes, but if you're using an older or more specialized board, be sure to follow instructions specific to it. This section gives you a brief rundown of other available boards.

## Official Arduino boards

Although Arduino is open source, it is also a trademarked brand, so to guarantee the quality and consistency of its products, the Arduino team must properly approve new boards before they are officially recognized and can bear the name Arduino. You can recognize official boards first by the name — Arduino MKR, Arduino GIGA, or Arduino Nano, for example. Other nonofficial boards often include *Arduino compatible* or *for Arduino* in the name. The other way to recognize an official Arduino, made by the Arduino team, is by the branding (in the most recent versions): They are turquoise and display the infinity symbol somewhere on the board, along with a link to <https://www.arduino.cc/>. Some other companies also have their boards accepted as official boards, so you may find other company names printed on them, such as Adafruit Industries, SparkFun and Seeed Studio.

Because the schematics for the Arduino board are open source, unofficial Arduino boards have a lot of variation, which people have made for their own needs. These boards are usually based on the same microcontroller chips to remain compatible with the Arduino software, but they require extra consideration and reading to be sure that they will work as expected. The Seeeduino v4.3 (by Seeed Studio), for example, is based on the Arduino Uno and is 100 percent compatible but adds various extra connections, switches, and sockets, which may be of more use to you than an official Arduino board in certain situations.

Official boards are the safe option for beginners to choose because the majority of Arduino examples online are based on these boards. Because of this, official boards are more widely used, and because of *that*, any errors or bugs in the board design are likely to be remedied with the next revision or at least well documented. As well as the Arduino Uno form-factor, Arduino provides a variety of different board families to suit the varying demands of your project.

### Arduino Uno

The Uno is the classic Arduino board — the one most people start with. It's designed to be simple, robust, and beginner-friendly. The latest version, the Uno R4, keeps the same physical footprint and pin layout as previous Uno boards, so it

remains compatible with a huge ecosystem of shields (see Chapter 12). Under the hood, it's been completely modernized with a faster 32-bit processor, more memory, and a USB-C port for power and programming. Whether you're blinking your first LED or building a robot, an Uno board is a reliable and flexible starting point.

## **Arduino GIGA**

The GIGA board is for big ideas and bold projects. The GIGA R1 WiFi is the most powerful board Arduino has ever made, with a dual-core 32-bit processor, tons of memory, and built-in Wi-Fi and Bluetooth. It's perfect for demanding applications like robotics, multimedia, or real-time control systems. If the Uno is your city car, the GIGA is a turbocharged off-roader.

## **Arduino Nano**

Nano boards are small but mighty. Designed for compact projects, they're great for breadboarding or squeezing into tight spaces. The Nano ESP32 adds Wi-Fi, Bluetooth, and USB-C to the mix, making it a great choice for connected devices. Nano boards may be tiny, but they punch well above their weight.

## **Arduino MKR**

The MKR is all about Internet of Things (IoT). These slim boards combine 32-bit processors with built-in Wi-Fi, Bluetooth, LoRa, or even cellular connectivity. Many also include a battery connector, so you can take your project on the go. If you're building something smart and connected — like a weather station or a remote sensor — MKR boards are built for the job.

## **Arduino Portenta**

Portenta boards are professional-grade Arduinos. They feature powerful processors, advanced I/O, and support for real-time operating systems (RTOS). Some models include features like dual-core architecture, and even the ability to run Python or TensorFlow LiteRT, a lightweight version of TensorFlow designed to run machine-learning models efficiently on small devices. They're ideal for industrial or AI-powered applications — or for anyone ready to go beyond the basics.

## **Arduino Nicla**

Tiny but specialized, the Nicla boards are designed for smart sensing and machine learning at the edge. They pack serious tech — like cameras, microphones, IMUs (Inertial Measurement Units, which combine sensors like accelerometers and gyroscopes to detect movement and orientation), and environmental sensors —

into postage-stamp-sized boards. Nicla boards are perfect for wearables, motion detection, or AI projects that need to run on their own without a big processor behind them.

## Arduino-compatible boards

Alongside the official Arduino range, you'll find many Arduino-compatible boards made by other companies. These boards aren't official Arduinos, but they can be programmed in the Arduino software — the integrated development environment (IDE), which is the application you use to write, test, and upload your programs, and they use the same style of libraries and examples. Popular options include Wi-Fi/Bluetooth boards from Espressif (ESP32), the Raspberry Pi Pico (RP2040), and ecosystem families such as Adafruit Feather and SparkFun RedBoard.

Why choose one? They often add handy features (built-in wireless, battery charging, tiny form factors) or hit a lower price point. The trade-offs are that pin layouts, voltage levels (many are 3.3V), and available libraries can differ from an Uno, so you should always check the board's documentation before wiring or running a sketch.



WARNING

Before you can select a third-party board in the Arduino IDE, you may need to install its board core via Tools ⇄ Board ⇄ Boards Manager. The board's product page will usually give step-by-step instructions.

### ESP32

The ESP32 is one of the most popular Arduino-compatible boards on the market. Developed by Espressif, it offers built-in Wi-Fi and Bluetooth at a fraction of the cost of many official boards. It's widely available, often appearing in inexpensive kits, and has enough processing power for demanding projects like home automation, streaming sensors, or even lightweight AI tasks. Although it isn't an official Arduino, the ESP32 is supported by the Arduino IDE and enjoys a massive global community, making it a versatile choice if connectivity is a priority.

### Raspberry Pi Pico

The Raspberry Pi Pico is based on the RP2040 microcontroller, designed by the Raspberry Pi Foundation. It's inexpensive, compact, and very beginner-friendly, while still powerful enough for advanced projects. The Pico is especially good for education and quick prototyping because of its clear documentation and wide availability worldwide. Like the ESP32, it's not an official Arduino, but it can be programmed using the Arduino IDE as well as MicroPython and C/C++, giving you extra flexibility depending on your learning style or project needs.

## Adafruit Feather

The Adafruit Feather range is a family of slim, lightweight boards designed with portability and expandability in mind. Each Feather board shares the same compact footprint, making them easy to integrate into projects, and many include built-in features such as LiPo battery charging, Wi-Fi, Bluetooth, or LoRa connectivity. The Feather ecosystem also includes a wide variety of “FeatherWing” add-on boards that snap on to expand functionality, from displays to motor drivers. Fully compatible with the Arduino IDE, Feather boards strike a balance between professional polish and maker flexibility, making them an excellent option if you want a versatile platform that can grow with your project.

# Shopping for Arduino

When Arduino first launched, boards were only available from a handful of specialist hobby shops. Today, you have a wide range of options, from the official Arduino store to global distributors and online marketplaces. Later in this chapter, you’ll also see how beginner’s kits can be a convenient way to get started with the right mix of parts.

An up-to-date list of official distributors is available at <https://store.arduino.cc/pages/distributors>. The following are some of the most useful places to buy boards and components.

## Arduino Store

`store.arduino.cc`

The Arduino Store is the safest place to start. It always has the latest official Arduino boards, starter kits, and selected components. If you want to be sure you’re getting a genuine board, this is the place to go. Ships worldwide.

## Adafruit

`adafruit.com`

Founded in 2005 by MIT engineer Limor “Ladyada” Fried, Adafruit offers a huge range of products it designs and manufactures itself, alongside carefully selected third-party items. The site includes excellent tutorials, videos, and community support. The Adafruit Feather range is a highlight.

## SparkFun

[sparkfun.com](http://sparkfun.com)

SparkFun designs and sells its own boards, kits, and components, many of which are Arduino-compatible. Their website doubles as a shop and a learning platform, with detailed product pages, tutorials, and community discussions. Based in Boulder, Colorado.

## Seed Studio

<https://www.seeedstudio.com/>

Headquartered in Shenzhen, China, Seed Studio is an “open hardware facilitation” company. It manufactures and distributes Arduino-compatible products worldwide and offers rapid prototyping and small-batch production. Its online community votes on new product ideas.

## Pimoroni

<https://pimoroni.com>

Based in the UK, Pimoroni is known for colorful, beginner-friendly products and kits. A great choice for hobbyists in the UK and EU looking for quick shipping and playful designs.

## Watterott Electronic

<https://www.watterott.com/>

Founded by Stephan Watterott back in 2008, Watterott stocks a wide range of Arduino-compatible boards, sensors, and components. A solid option for EU customers.

## RobotShop

<https://robotshop.com>

A robotics-focused supplier with warehouses in North America, Europe, and Asia. Sells Arduino boards and accessories alongside a wide variety of robot kits and components.

## Electronics distributors

When you need reliable sourcing, bulk orders, or very specific parts, these global distributors are your best bet. Their catalogues are huge, so it helps to know the exact part or a close equivalent before you dive in.

- » **Digi-Key:** [digikey.com](https://www.digikey.com)
- » **RS Components:** <https://rs-online.com>
- » **Mouser:** <https://www.mouser.com/>
- » **Rapid:** <https://www.rapidonline.com/>
- » **Farnell:** <https://farnell.com>
- » **element14:** <https://element14.com>
- » **Newark:** <https://newark.com>



TIP

You can lose days searching through the extensive catalogues of components, so it's always a good idea to know the name of what you're looking for before you start! You can normally find this on the datasheet when you buy from more accessible hobbyist stores.

## Amazon

Arduino boards and starter kits are widely available on Amazon, alongside a huge variety of components. Convenience and fast shipping are a plus, but genuine products can be harder to find among unrelated listings. Always check that the seller is an official distributor or that the board carries the Arduino brand.

## eBay

Once known mainly for second-hand parts, eBay is now a marketplace for many retailers and wholesalers. You'll find official boards, clones, and custom variants side by side. Some clones work fine, but build quality and safety can vary. If you're just starting out, stick with certified distributors or clearly branded Arduino boards.

## Kitted Out: Starting with a Beginner's Kit

By this point, you probably know a bit about the Arduino board, but no board is an island; you need other components to make use of it. In the same way that a computer isn't much use without a mouse and keyboard, an Arduino isn't as much fun without extra parts to connect and experiment with.

Every new Arduinist should follow a few basic examples to learn the fundamentals of Arduino (covered in Chapters 3 through 7). These examples can all be built with a handful of common components. To save you the time and effort of sourcing these separately, many companies put together starter kits that let you begin experimenting straight away.

Many different starter kits are available, each reflecting the experiences and preferences of the people or companies who designed them. You'll often find the same type of component in slightly different shapes or packages, but they all serve the same purpose once you know what to look for.

Instead of listing every possible part, it's most helpful to think of a kit in terms of the core components you'll use all the time and the extra parts that give you more to explore.

## Essentials

- » **Arduino Uno board:** The star of the show. Most kits come with an official Arduino Uno, often the R4 Minima, which is reliable and beginner-friendly.
- » **USB-C cable:** Used to power the board and upload your programs. This is the lifeline between your computer and Arduino.
- » **LEDs (plus resistors):** Little lights that glow when powered. They're perfect for testing your projects, giving you instant feedback. Resistors are included to keep the LEDs from burning out.
- » **Pushbuttons:** Simple switches you press to send signals to the Arduino — like turning something on or off with a tap.
- » **Potentiometer:** A dial that lets you vary resistance. Great for adjusting brightness, speed, or volume in your projects.
- » **Light sensor (LDR):** Changes resistance depending on how much light shines on it. You can use it to detect day and night or to build a simple light meter.
- » **Temperature sensor:** Measures the surrounding temperature and lets your Arduino respond to changes in the environment.
- » **Servomotor:** A small motor with built-in control that can move to specific angles. It's commonly used for precise motion, like moving a robot arm or steering a model car.

## Nice to have

Many kits also include these parts, which add variety and open up new project ideas:

- » **Piezo buzzer:** Makes beeps, tones, or simple tunes. You can also use it to detect vibration if you attach it to a surface.
- » **Diodes and transistors:** Tiny parts that help you control the flow of electricity. They're handy for switching circuits or amplifying signals.
- » **Relays:** Small electromechanical switches that let your Arduino control higher-powered devices, such as lamps or motors, without frying the board.
- » **DC motor:** A basic motor that spins continuously when powered — good for fans, wheels, or simple moving gadgets.
- » **Additional sensors and modules:** Some kits include bonus parts like ultrasonic distance sensors, accelerometers, or tiny displays, which give you even more ways to experiment.



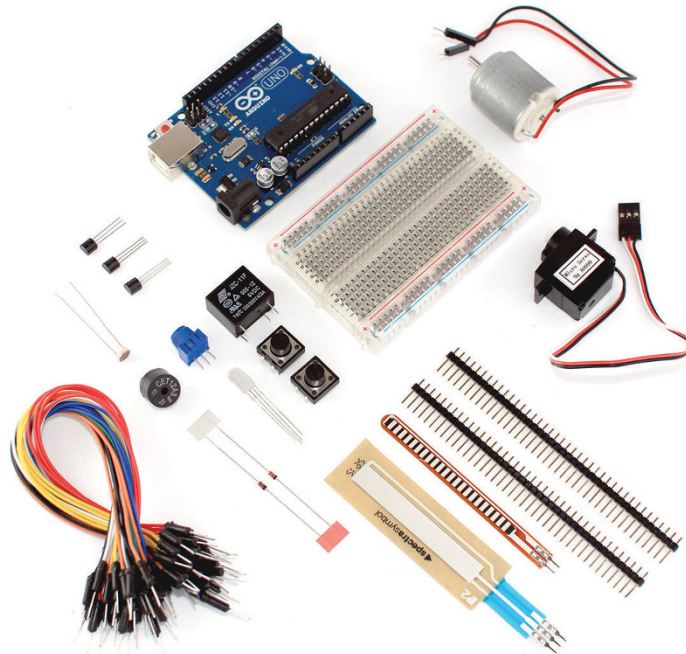
TIP

Don't worry if your kit looks a little different. The important thing is that you have a balance of *inputs* (sensors, buttons) and *outputs* (LEDs, motors, buzzers) so you can start building interactive projects right away.

Here are a few of the better-known kits. They include all the components in the preceding list, and any will be an excellent companion for the examples in this book:

- » **Arduino Starter Kit** — The official kit from Arduino. It comes with high-quality components and a printed project book that walks you through 15 experiments. Available at <https://store.arduino.cc/arduino-starter-kit>, usually around US\$90–100.
- » **SparkFun Inventor's Kit (v5.0)** — A well-curated kit from SparkFun that includes a large variety of parts and a detailed guidebook. It's widely used in classrooms and workshops. Available at <https://sparkfun.com/products/15631>, usually around US\$110.
- » **Adafruit Arduino Starter Pack** — A solid kit from Adafruit with an Uno, breadboard, sensors, motors, and LEDs. Adafruit's online tutorials are an excellent companion to the pack. Available at <https://adafruit.com/product/68>, usually around US\$100.
- » **Budget kits (Amazon/eBay/AliExpress)** — Inexpensive alternatives that typically cost US\$30–50. These include a wide selection of parts, often with clone boards. They're great for experimenting, but quality can vary, so beginners may prefer an official kit for reliability.

You can create all the basic examples in this book with any kit in the preceding list, although a slight variation may occur in the number and type of components. See an example Arduino kit in Figure 2-3. Sometimes the same component can take many different forms, so be sure to carefully read the parts list to make sure that you can identify each of the components before you start. Cheaper kits are available, but these will likely not include some components, such as motors or the variety of sensors.



**FIGURE 2-3:**  
An example of an  
Arduino kit with  
an Arduino Uno  
and a good range  
of components.

## Preparing a Workspace

When working on an Arduino project, you could sit on your sofa or be at the top of a ladder (I've tried both). But just because it's possible doesn't mean that it's sensible or advisable. You'll have a much easier time if you set up a proper workspace before diving in, especially when you're just starting out.

Working with electronics is a fiddly business. You're dealing with lots of tiny, delicate, and very sensitive components, so you need great precision and patience when assembling your circuit. If you're in a dimly lit room trying to balance things on your lap, you'll quickly go through your supply of components by either losing or destroying them.

It's always good to make life easy for yourself, and the best way to do this when working on Arduino projects is to prepare your workspace. The ideal workspace has the following:

- » Large, uncluttered desk or table
- » Good work lamp
- » Comfortable chair
- » Small containers or a parts box to keep components organized
- » Cup of tea or coffee (highly recommended)



TIP

Try to avoid working on soft furnishings such as carpets or sofas, as static electricity can sometimes harm sensitive parts.

When you're comfortable, you're ready to learn how to set up the Arduino IDE. Versions of Arduino software are available for Windows, macOS, Linux, and Arduino Cloud Editor.

## Installing Arduino

This section talks you through installing the *Arduino IDE* on your platform of choice. These instructions are specifically for installation using an Arduino Uno R4 Minima, but they work just as well for earlier boards.

The Arduino IDE is free to download from [arduino.cc/en/software](https://arduino.cc/en/software). It is supported on macOS, Windows and Linux (32-bit, 64-bit and ARM). At the time of this writing, Arduino was version 2.x.

To install Arduino IDE 2.x:

- » **On macOS:** Download the .zip file. It will usually unzip automatically, giving you the Arduino app, which you can drag to your Applications folder. You may need to grant permission the first time you open it.
- » **On Windows:** Download the installer .exe file and double-click to run it. The installer guides you through setup, including creating shortcuts and installing USB drivers if needed.
- » **On Linux:** Download the package for your distribution. The IDE is distributed as a compressed folder (arduino-ide\_x.x.x) or as an .AppImage file that runs directly once made executable. For advanced users, the .deb and .rpm packages integrate with system package managers.



TIP

If you prefer not to install any software, you can use the Arduino Cloud Editor, which runs in your web browser and stores your sketches online.

## Using Arduino Cloud Editor

Arduino Cloud Editor is a cloud-based version of the downloadable IDE and a great way to get started quickly.

*Cloud-based* means that Cloud Editor runs in your browser and saves your project files to your online account instead of on your computer. Apart from those differences, Cloud Editor has all the core features of the downloadable IDE but is more flexible, working across platforms without the need to install any files. In Figure 2-4, you can see how the layout differs.

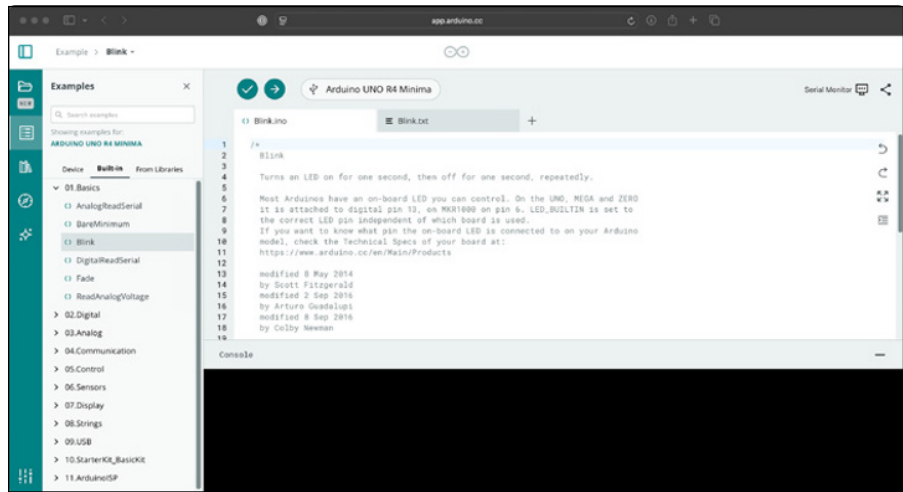
## Surveying the Arduino Environment

Programs written for Arduino are known as *sketches*. This naming convention was passed down from Processing, which allowed users to create programs quickly, in the same way that you would scribble an idea in a sketchbook.

Before you write your first sketch, take a moment to explore the Arduino IDE. The IDE is presented as a graphical user interface (GUI), which simply means you work with menus, buttons, and windows instead of typing long command-line instructions in a terminal.

In the Arduino IDE, the main window is divided into several key areas (see Figure 2-5):

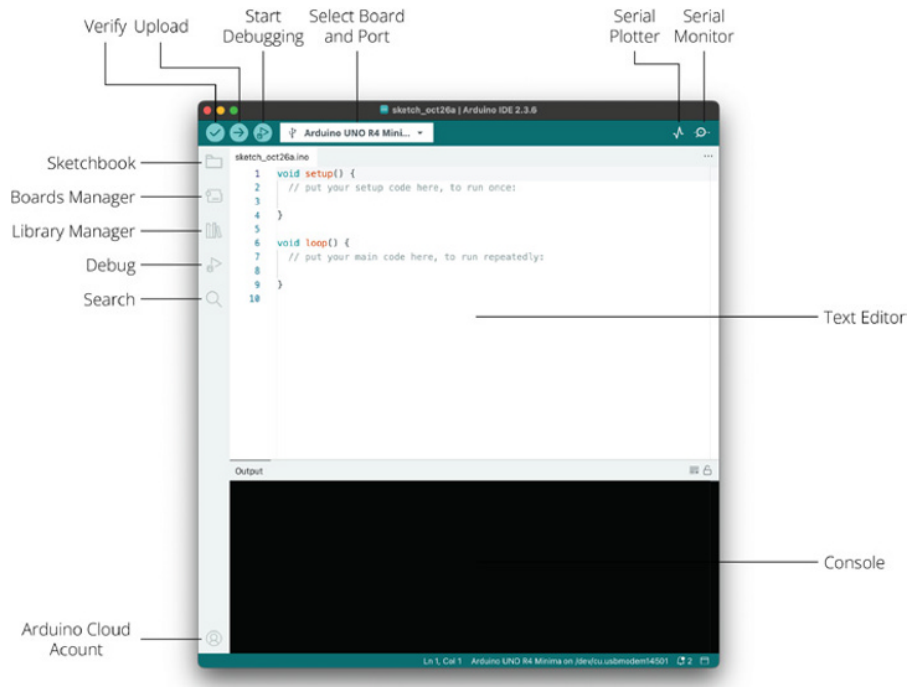
- » **Menu bar (top of the window/system menu):** As with most apps, this contains drop-down menus for File, Edit, Sketch, Tools, and Help. These give you access to everything from board settings to library management. On macOS, it appears at the top of the screen; on Windows and Linux, it sits at the top of the IDE window.
- » **Toolbar (icons under the menu bar):** Quick-access buttons for the most common tasks:
  - *Verify (✓):* Checks your code for errors (a process called compiling). It's like a spelling and grammar checker for your sketch — it won't guarantee the program works, but it catches obvious mistakes.



**FIGURE 2-4:**  
The Arduino  
Cloud Editor GUI.

- **Upload (→):** Compiles and then uploads your sketch to the connected board.
- **Debug (bug icon):** Starts a live debugging session (only available on supported boards).
- **Board/Port selector (drop-down):** Lets you quickly choose your board type and USB port.
- **Serial Plotter (waveform icon):** Allows you to view data that is being sent to or received by your Arduino board.
- **Serial Monitor (magnifying glass icon):** Opens a window to view text data sent to or received from your Arduino.
- » **Sidebar (icons on the left edge):** A new addition in IDE 2.x, the sidebar gives you quick access to tools like:
  - *Sketchbook (your saved sketches).*
  - *Library Manager (to add libraries).*
  - *Board Manager (to install support for different boards).*
  - *Debugger (for boards that support it).*
- » **Editor pane (center):** This is where you write your sketch. The editor highlights keywords in color, indents your code neatly, and shows line numbers. By default, you see the two standard functions:
  - *setup() runs once at the start.*
  - *loop() repeats continuously.*

- » **Tabs (top of editor pane):** Each sketch file appears as a tab, so you can work with multiple files in a project.
- » **Message area (bottom bar):** Shows status information such as “Compiling sketch. . .” or “Upload complete.” It also displays error messages if something goes wrong. On the right, it shows cursor position (line/column) and board connection status (currently: “No board selected”).



**FIGURE 2-5:**  
The areas  
of the GUI.

## Using AI Wisely in Your Arduino Projects

AI tools have become incredibly useful companions when you’re learning Arduino. Whether you’re using Arduino’s own AI Assistant in the Arduino Cloud or general-purpose AI tools like ChatGPT, Claude, or Gemini, they can help you get unstuck, discover new ideas, and even write chunks of code. But because Arduino lives in the real world — with wires, voltages, sensors, and the occasional puff of smoke — AI has its limits. Used well, it can speed you up. Used blindly, it can send you down the wrong rabbit hole.

It may seem a little odd to read a book about learning electronics at a time when AI can happily generate sketches for you. But if you’re holding this book, chances

are you're not here just to solve a problem and move on. You're here because you enjoy understanding how things work and want the satisfaction of building something yourself. AI can help you explore ideas more quickly and clear up confusing concepts, but it can't replace the curiosity that brought you here. That part is still very much your own.

## How AI can help

AI isn't just about generating code. When you're learning Arduino, it can help explain unfamiliar concepts, guide you through problems, and suggest useful starting points when you're not sure where to begin. The sections that follow explore some of the most common and effective ways AI can support your Arduino learning, while still keeping you firmly in control of your project.

### A tutor on demand

AI is excellent at explaining concepts in simple terms. If you're not sure what PWM is, why your button is "floating," or how the I2C address for your sensor works, you can ask an AI for a clear, personalized explanation. Arduino's own AI Assistant is especially good at tailoring help to your sketch and board because it's aware of the Arduino ecosystem, though general GPT-style tools can give broader or more detailed examples.

This makes AI feel a bit like having an experienced maker sitting beside you who never gets tired of your questions.

### A debugging partner

AI can spot missing semicolons or mismatched brackets, but it can also help you understand error messages, restructure a loop, or suggest ways to print diagnostics to the Serial Monitor. It can walk through your logic and tell you where things aren't doing what you expect. When used in the Arduino Cloud, the AI Assistant can even reason about your sketch in context.

For beginners, it can dramatically shorten those "why won't this compile?" moments.

### A research assistant

One of the biggest strengths of AI is surfacing things you didn't know existed:

- » Libraries for unusual sensors
- » Example sketches buried in forum threads

- » Different approaches to the same problem
- » Compatibility notes you might miss at first glance
- » Niche hardware components you may never have searched for directly, but which turn out to be exactly what your project needs

In my experience, this is where AI shines: finding the starting point, suggesting the right library, or generating boilerplate code that gets you exploring your idea quickly.

## Rapid prototyping and creative exploration

If you want to quickly test an idea — a touch sensor, a small display, a servo animation, or a sequence of LEDs — AI can generate a sample sketch in seconds. You can ask for multiple variations, compare approaches, and get a feel for how your project might work before you commit to refining it.

This makes early prototyping feel faster and more playful.

## Where AI falls short

AI can be a powerful helper, but it isn't infallible. When your project moves from code on a screen to components on a breadboard, there are plenty of ways things can go wrong that AI simply can't detect. The sections that follow look at where AI guidance is most likely to fall short, and why hands-on testing is always essential.

## Electronics are physical, and AI can't see them

AI can suggest wiring diagrams, but it can't see that:

- » You connected a sensor to 5V when it needs 3.3V
- » Your ground wire popped out
- » Your LED strip is drawing too much current
- » Your breadboard track is broken

Real hardware needs real testing, and no AI can protect you from basic electrical mistakes.

## AI often assumes perfect components

AI might suggest code that compiles beautifully but doesn't match your specific sensor, library version, or board revision. It may:

- » Mix up pin numbers
- » Reference library functions that don't exist
- » Propose timing that isn't realistic
- » Suggest circuits that quietly exceed safe limits

Because it generates patterns, not tested circuits, you still need to double-check its technical details.

## It sometimes “hallucinates”

AI may invent functions, misunderstand datasheets, or recommend hardware that is incompatible with your setup. This isn't unusual — which is why you never want to rely on AI alone when something plugs into the real world.

## A healthy way to use AI with Arduino

Over time, I've formed a simple opinion about using AI with Arduino. AI is a brilliant tool for *exploring* an idea, learning concepts, or finding a library you didn't know existed. But once your project gets more complex, it becomes essential that you understand the pseudocode — the underlying logic — behind what you're building.

If an AI writes a big block of code for you, make sure you can retrace the steps. That way, if something isn't working the way you expected, you can debug it without feeling lost.

I'd recommend:

- » Asking AI to explain a concept before asking for full code
- » Requesting pseudocode first, then turning it into real code
- » Building your sketch gradually, testing each piece as you go
- » Using AI to explore options, not decide them for you
- » Asking AI to search community forums or GitHub issues for real-world experiences
- » Keeping the final judgment (and wiring!) in your own hands

Think of AI as an assistant who is very fast, generally helpful, but occasionally wildly confident about things that aren't true. It accelerates you, but it doesn't replace the understanding you build by experimenting and testing.

## **Working smart with AI**

AI can make learning Arduino more accessible, more enjoyable, and far less intimidating. Tools like the Arduino Cloud AI Assistant can help you understand your own sketches and hardware, while general AI tools are great for exploring ideas, finding libraries, and seeing how other people approach similar problems. Used well, AI lets you prototype faster and learn more along the way.

The key is to treat AI as a learning companion rather than a shortcut. By building your projects step by step, asking for explanations as well as code, and testing each part as you go, you stay in control of how your project works. When you understand the logic behind what you've built, AI becomes a powerful support for your curiosity rather than a replacement for it.

- » Identifying your board
- » Setting the software
- » Uploading Blink
- » Completing your first Arduino sketch

## Chapter 3

# Blinking an LED

**B**race yourself. You are about to take your first real step into the world of Arduino! You've bought a board, maybe an Arduino starter kit (possibly from one of the suppliers I recommended), and you're ready to go.

It's always a good idea to have a clear work surface or desk to use when you're tinkering. It's not uncommon to drop or misplace some of the many tiny components you work with, so make sure your workspace is clear, well lit, and accompanied by a comfortable chair.

By its nature, Arduino is a device intended for performing practical tasks. The best way to learn about Arduino, then, is in practice — by working with the device and *doing* something. That is exactly the way I write about it throughout this book. In this chapter, I take you through some simple steps to get you on your way to making something.

I also walk you through uploading your first Arduino sketch. After that, you examine how it works and see how to change it to do your bidding.

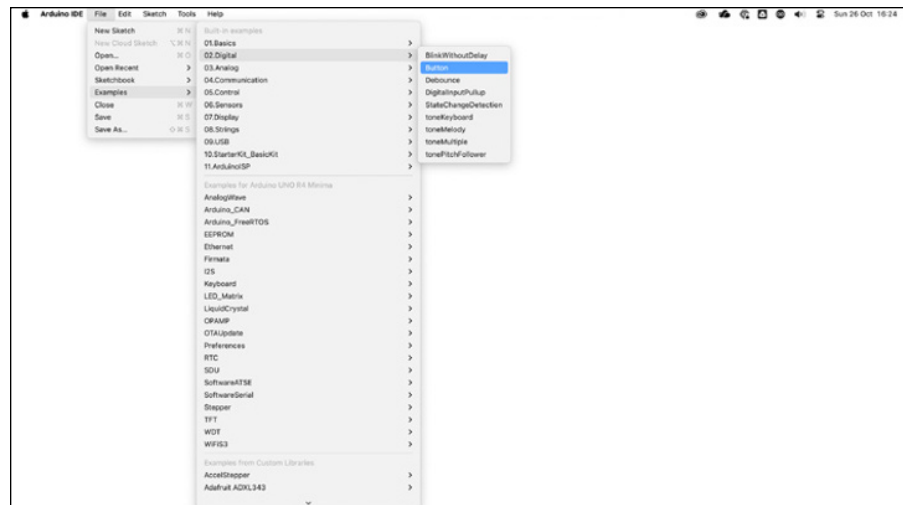
# Working with Your First Arduino Sketch

In front of you now should be an Arduino Uno, a USB cable, and a computer running your choice of operating system (Windows, macOS, Linux, or Web Editor). The next section shows what you can do with this little device.

## Finding the Blink sketch

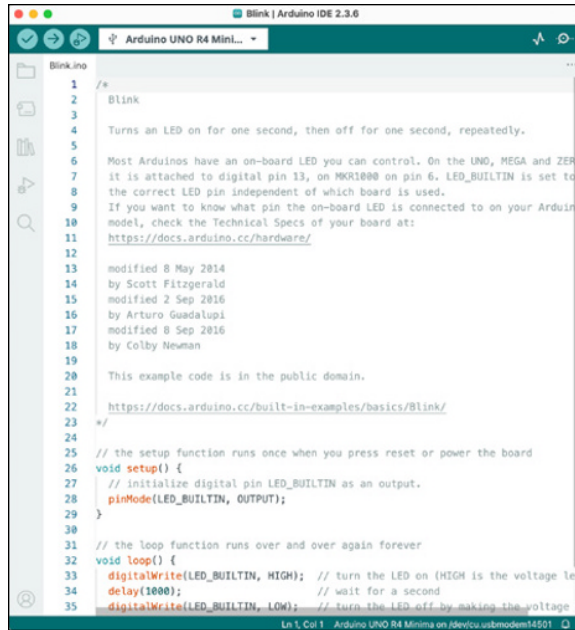
To make sure that the Arduino software is talking to the hardware, you upload a sketch. What is a sketch, you ask? Arduino was created as a device that allows people to quickly prototype and test ideas using little bits of code that demonstrate the idea — kind of like how you might sketch out an idea on paper. For this reason, programs written for Arduino are referred to as *sketches*. Although a device for quick prototyping was its starting point, Arduino devices are being used for increasingly complex operations. So don't infer from the name sketch that an Arduino program is trivial.

The specific sketch you want to use here is called Blink. It's about the most basic sketch you can write, a sort of “Hello, world!” for Arduino. Click in the Arduino window. From the menu bar, choose File ⇨ Examples ⇨ 01.Basics ⇨ Blink (see Figure 3-1).



**FIGURE 3-1:**  
Find your way to  
the Blink sketch.

A new window opens in front of your blank sketch and looks similar to Figure 3-2.



```
1  /*
2  3  Blink
4  5  Turns an LED on for one second, then off for one second, repeatedly.
6  7  Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
8  9  it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
10 11  the correct LED pin independent of which board is used.
12 13  If you want to know what pin the on-board LED is connected to on your Arduino
14 15  model, check the Technical Specs of your board at:
16 17  https://docs.arduino.cc/hardware/
18 19  modified 8 May 2014
20 21  by Scott Fitzgerald
22 23  modified 2 Sep 2016
24 25  by Arturo Guadalupi
26 27  modified 8 Sep 2016
28 29  by Colby Newman
30 31  This example code is in the public domain.
32 33  https://docs.arduino.cc/built-in-examples/basics/Blink/
34 35  */
36 37  // the setup function runs once when you press reset or power the board
38 39  void setup() {
40 41  // initialize digital pin LED_BUILTIN as an output.
42 43  pinMode(LED_BUILTIN, OUTPUT);
44 45  }
46 47  // the loop function runs over and over again forever
48 49  void loop() {
50 51  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
52 53  delay(1000); // wait for a second
54 55  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
56 57  }
```

**FIGURE 3-2:**  
The Arduino  
Blink sketch.

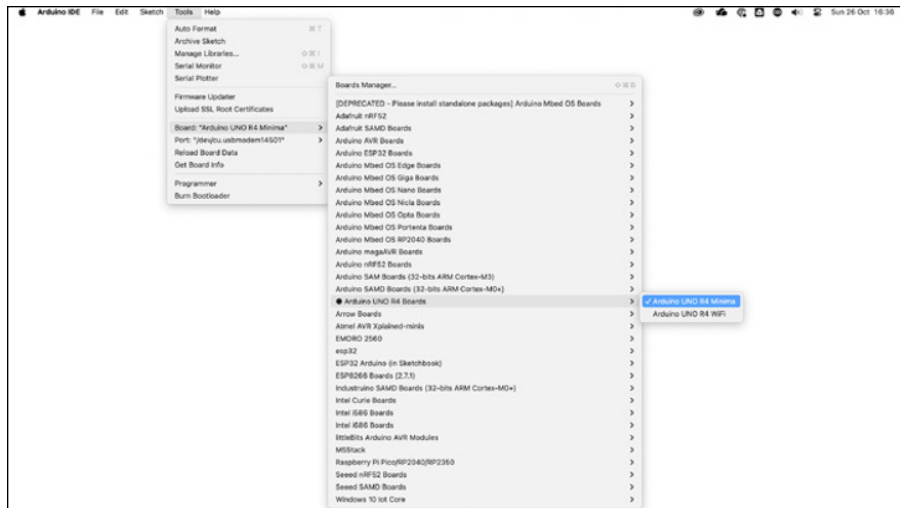
## Identifying your board

Before you can upload the sketch, you need to check a few things. First you should confirm which board you have. As I mention in Chapter 2, you can choose from a variety of Arduino devices and several variations on the USB board. The latest generation of USB boards is the Uno R4 Minima, so the following examples will feature that board. If you have an older board, such as the Uno R3, the core capabilities will be very similar but you will need to make sure that you adjust the examples to reference your board.

## Configuring the software

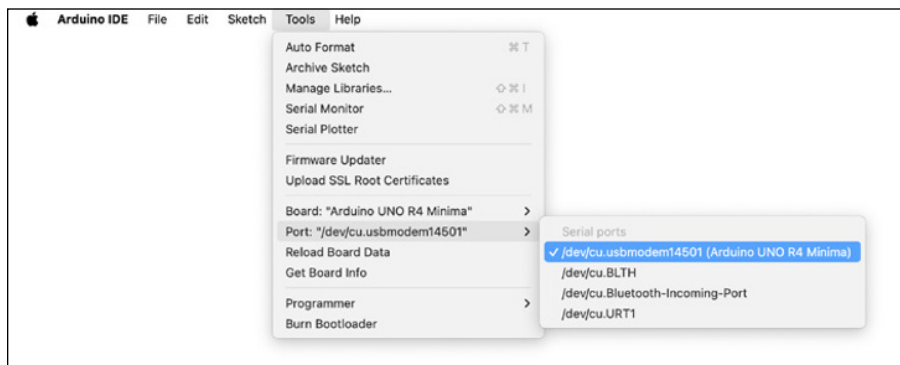
After you confirm the type of board you're using, you have to provide that information to the software. From the Arduino main menu bar (at the top of the Arduino window in Windows and at the top of the screen in macOS), choose Tools ⇨ Board. You should see a list of the different kinds of boards supported by the Arduino software. Select your board from the list, as shown in Figure 3-3.

Next, you need to select the serial port. The *serial port* is the connection that enables your computer and the Arduino device to communicate. *Serial* describes the way that data is sent, one bit of data (0 or 1) at a time. The *port* is the physical interface, in this case a USB socket. I talk more about serial communication in Chapter 6.



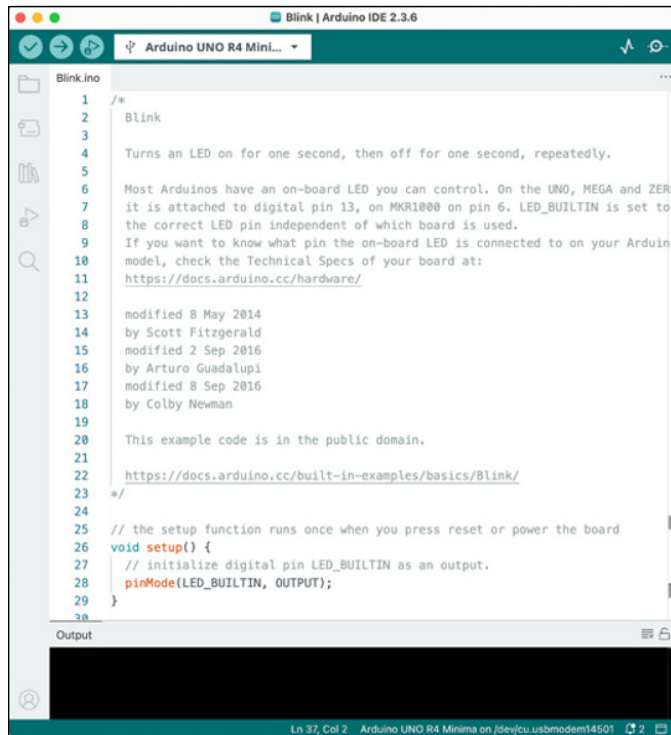
**FIGURE 3-3:**  
Select Arduino UNO R4 Minima from the Board menu.

To determine the serial port, choose Tools ⇨ Port. A list of devices connected to your computer is displayed (see Figure 3-4). This list contains any device that can talk in serial, but for the moment, you're interested only in finding the Arduino. If you've just installed Arduino and plugged it in, it should be at the top of the list. For macOS users, the Arduino is shown as `/dev/cu.usbmodemXXXXXX` or `/dev/cu.usbmodemXXXXXX` (where `XXXXXX` is a randomly assigned number). On Windows, the same is true, but the serial ports are named `COM1`, `COM2`, `COM3`, and so on. The highest number is usually the most recent device.



**FIGURE 3-4:**  
A list of serial connections available to the Arduino software.

After you find your serial port, select it. It should appear in the bottom-right of the Arduino graphic user interface (GUI), along with the board you selected (see Figure 3-5).



**FIGURE 3-5:**  
The Arduino GUI  
board and port.

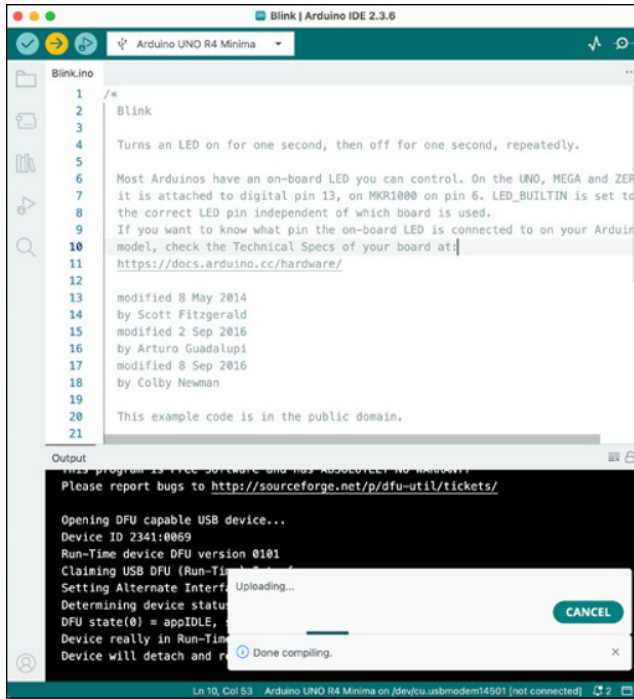
## Uploading the sketch

Now that you have told the Arduino software what kind of board you're communicating with and which serial port connection it's using, you can upload the Blink sketch you found earlier in this chapter.

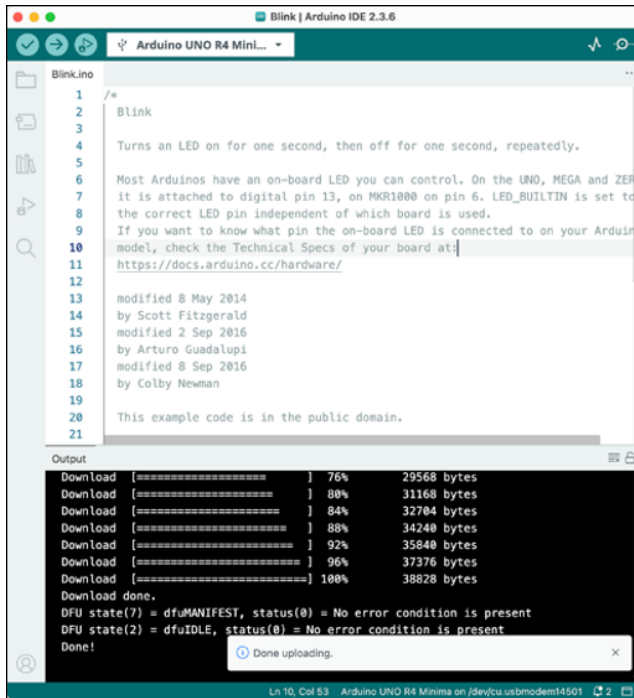
First click the Verify button (check mark). Verify checks the code to make sure it makes sense, that is, that the syntax is written in a way Arduino can understand (see Chapter 2). However, this process doesn't necessarily mean that your code will do what you're anticipating.

You should see a progress bar and the message *Compiling Sketch* for a few seconds, followed by the message *Done Compiling* (see Figure 3-6) after the process has finished.

If the sketch compiled successfully, you can now click the Upload button (right arrow) next to the Verify button. A progress bar appears, and you see a flurry of activity on your board from the two LEDs marked RX and TX (mentioned in Chapter 2). These show that the Arduino is sending and receiving data. After a few seconds, the RX and TX LEDs stop blinking, and a *Done Uploading* message appears at the bottom of the Arduino window (see Figure 3-7).



**FIGURE 3-6:** The progress bar shows that the sketch is compiling.



**FIGURE 3-7:** The Arduino software has successfully uploaded a sketch.

## Congratulate yourself!

You should see the LED marked L blinking away reassuringly: on for a second, off for a second. If that is the case, give yourself a pat on the back. You've just uploaded your first piece of Arduino code and entered the world of physical computing!

If you don't see the blinking L, go back through the previous sections. Make sure you have installed Arduino properly and then give it one more go. If you still don't see the blinking L, you may have a damaged or faulty board. You can contact the Arduino support team at <https://www.arduino.cc/en/contact-us/>.

## What just happened?

Without breaking a sweat, you've just uploaded your first sketch to an Arduino. Congratulations!

Just to recap, you have now:

- » Plugged your Arduino into your computer
- » Opened the Arduino software
- » Set the board and serial port
- » Opened the Blink sketch from the Examples folder and uploaded it to the board

In the following section, I walk you through the various parts of the sketch you just uploaded.

## Looking Closer at the Sketch

In this section, I show you the Blink sketch in a bit more detail so that you can see what's actually going on. When the Arduino software reads a sketch, it quickly works through it one line at a time, in order. So the best way to understand the code is to work through it the same way but slowly.

Arduino uses the programming language C, which is one of the most widely used languages of all time. It's an extremely powerful and versatile language, but it takes some getting used to.

If you followed the previous section, you should already have the Blink sketch on your screen. If not, you can find it by choosing File ⇨ Examples ⇨ 01.Basics ⇨ Blink (refer to Figure 3-1).

When the sketch is open, you should see something like this:

```
/*
  Blink

  Turns an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
  it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
  the correct LED pin independent of which board is used.
  If you want to know what pin the on-board LED is connected to on your Arduino
  model, check the Technical Specs of your board at:
  https://docs.arduino.cc/hardware/

  modified 8 May 2014
  by Scott Fitzgerald
  modified 2 Sep 2016
  by Arturo Guadalupi
  modified 8 Sep 2016
  by Colby Newman

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/basics/Blink/
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

The sketch is made up of lines of code. When looking at the code as a whole, you can identify three distinct sections:

- » Comments
- » void setup
- » void loop

Read on for more details about each of these sections.

## Comments

Here's what you see in the first section of the code:

```
/*  
  
  Blink  
  
  Turns an LED on for one second, then off for one second, repeatedly.  
  Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO  
  it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to  
  the correct LED pin independent of which board is used.  
  If you want to know what pin the on-board LED is connected to on your Arduino  
  model, check the Technical Specs of your board at:  
  https://docs.arduino.cc/hardware/  
  
  modified 8 May 2014  
  by Scott Fitzgerald  
  modified 2 Sep 2016  
  by Arturo Guadalupi  
  modified 8 Sep 2016  
  by Colby Newman  
  
  This example code is in the public domain.  
  
  https://docs.arduino.cc/built-in-examples/basics/Blink/  
*/
```

## Multiline comment

Note that the code lines are enclosed within the symbols `/*` and `*/`. These symbols mark the beginning and end of a *multiline* or *block comment*. Comments are written in plain English and, as the name suggests, provide an explanation or comment on

the code. The software ignores comments when the sketch is compiled and uploaded. Consequently, comments can contain useful information about the code without interfering with how the code runs.

In this example, the comment simply tells you the name of the sketch, what it does, and provides a note explaining that this example code is in the public domain. Comments often include other details such as the name of the author or editor, the date the code was written or edited, a short description of what the code does, a project URL, and sometimes even contact information for the author.

## Single-line comment

Further down the sketch, inside the `setup` and `loop` functions, text on your screen appears with the same shade of gray as the comments above. This text is also a comment. The symbols `//` signify a single-line comment as opposed to a multi-line comment. Any code written after these forward slashes will be ignored for that line. In this case, the comment is describing the piece of code that comes after it:

```
// the setup function runs once when you press reset or power the board
```

## Functions

The next two sections are functions and begin with the word `void`: `void setup` and `void loop`. A *function* is a bit of code that performs a specific task, and that task is often repetitive. Rather than writing the same code out again and again, you can use a function to tell the code to perform the task again.

Consider the general process you follow to assemble IKEA furniture. If you were to write these general instructions in code, using a function, they would look something like this:

```
void buildFlatpackFurniture() {  
    buy a flatpack;  
    open the box;  
    read the instructions;  
    put the pieces together;  
    admire your handiwork;  
    vow never to do it again;  
}
```

The next time you want to use these same instructions, rather than writing out the individual steps, you can simply call the function named `buildFlatpackFurniture()`.



TIP

Although not compulsory, the Camel Case naming convention exists for function and variable names that contain multiple words. (If the convention were a function name, it would be written as CamelCase or camelCase.) Because these names can't have spaces, you need a way to distinguish where all the words start and end; otherwise, it takes a lot longer to scan them. The convention is to capitalize the first letter of each word after the first. This convention greatly improves the readability of your code, so I highly recommend that you adhere to this rule in all your sketches for your benefit and the benefit of those reading your code!

The word `void` is used for a function that returns no value, and the word that follows is the name of that function. In some circumstances, you might either put one or more values into a function or expect one or more values back from it, the same way you might put numbers into a calculation and expect a total back.



REMEMBER

You must include `void setup` and `void loop` in every Arduino sketch; they're the minimum required to upload. But it's also possible to write your own custom functions for the task you need to do. For now, just remember that you have to include `void setup` and `void loop` in every Arduino sketch you create. Without these functions, the sketch will not compile.

## Setup

Setup is the first function an Arduino program reads, and it runs only once. Its purpose, as hinted in the name, is to set up the Arduino device, assigning values and properties to the board that do not change during its operation. In the Blink sketch, the `setup` function looks like this:

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}
```



TIP

Note on your screen that the word `void` is turquoise and the word `setup` is orange. These colors indicate that the Arduino software recognizes these words as *core* functions, as opposed to a function you have written. If you change the case of the words to `Void Setup`, they turn black, which illustrates that the Arduino code is *case-sensitive*. Having the correct case is an important point to remember, especially when it's late at night and the code doesn't seem to be working.

The contents of the `setup` function are contained within the curly brackets, `{` and `}`. Each function needs a matching set of curly brackets. If you have too many of either bracket, the code does not compile, and you are presented with an error message like the one in Figure 3-8.



**FIGURE 3-8:** The Arduino software is telling you that a bracket is missing.

## PinMode

The `pinMode` function configures a specified pin for either input (receive data) or output (send data). The function includes two parameters:

- » **pin:** The number of the pin whose mode you want to set
- » **mode:** Either INPUT or OUTPUT

In the Blink sketch, after the single-line comment, you see this line of code:

```
pinMode(LED_BUILTIN, OUTPUT);
```

The word `pinMode` is colored orange. As I mention previously in this chapter, highlighted text indicates that Arduino recognizes the word as a core function. `LED_BUILTIN` and `OUTPUT` are colored blue so that they can be identified as *pre-defined variables*, values that the Arduino integrated development environment (IDE) recognizes. In this case, `LED_BUILTIN` tells the function that you want to control the onboard LED on the Arduino board (marked L) and `OUTPUT` sets the mode of that pin. I go into more detail about variables in Chapter 6.

That's all you need for `setup`. Next up, the `loop` section.

# Loop

The next section you see in the Blink sketch is the `loop` function, with the word `void` in turquoise and the word `loop` in orange, which means the Arduino software recognizes it as a core function. `loop` is a function, but instead of running one time, it runs continuously until you press the reset button on the Arduino board or remove the power. Here is the `loop` code:

```
// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);                      // wait for a second
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);                      // wait for a second
}
```

## DigitalWrite

Within the `loop` function, you again see curly brackets and two orange functions: `digitalWrite` and `delay`.

First is `digitalWrite`:

```
digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
```

The comment says turn the LED on, but what exactly does that mean? The `digitalWrite` function sends a digital value to a pin. As mentioned in Chapter 2, digital pins have only two states: on or off. In electrical terms, these can be referred to as either a HIGH or LOW value, which is relative to the voltage of the board.

An Arduino Uno requires 5V to run, which is provided by either a USB or an external power supply (as mentioned in Chapter 2), which the Arduino board reduces to 5V. This voltage is interpreted by the Arduino. A HIGH value is equal to 5V and LOW is equal to 0V.

The `digitalWrite` function includes two parameters:

- » **pin:** The name or number of the pin whose mode you want to set
- » **value:** Either HIGH or LOW

So `digitalWrite(LED_BUILTIN, HIGH);` in plain English would be “send 5V to the onboard LED on the Arduino,” which is enough voltage to turn on an LED.

## Delay

In the middle of the `loop` code, you see this line:

```
delay(1000); // wait for a second
```

This function does just what it says: It stops the program, in this case for an amount of time in milliseconds. In this case, the value is 1000 milliseconds, which is equal to one second. During this time, nothing happens. Your Arduino is chilling out, waiting for the delay to finish.

The next line of the sketch provides another `digitalWrite` function to the same pin, but this time writing it low:

```
digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
```

This function tells Arduino to send 0V (ground) to the onboard LED, which turns off the LED. This line of code is followed by another delay that pauses the program for one second:

```
delay(1000); // wait for a second
```

At this point, the program returns to the start of the loop and repeats itself, *ad infinitum*.

So the loop is doing this:

- » Sending 5v to the LED
- » Waiting a second
- » Sending 0v to the LED
- » Waiting a second

As you can see, this gives you the blink!

## Blinking Brighter

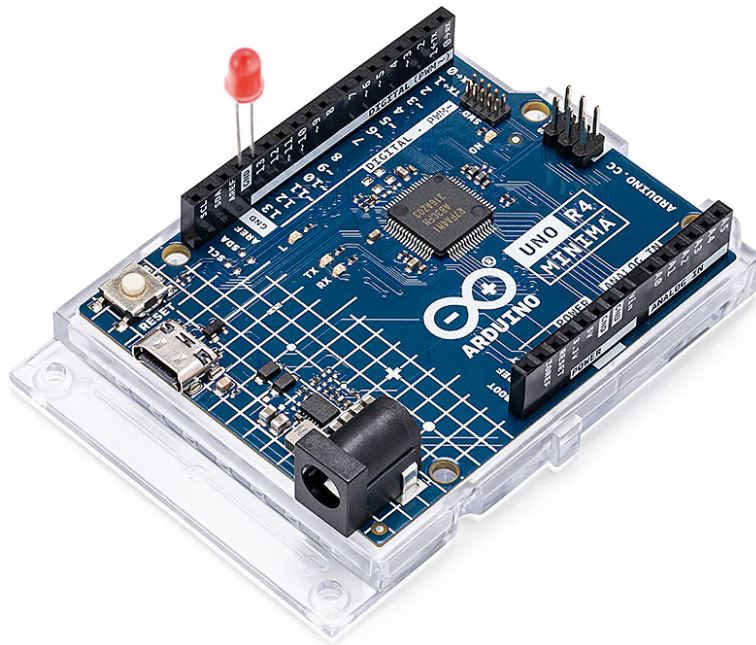
The LED marked L is actually connected just before it reaches pin 13. On earlier boards, it was necessary to provide your own LED. But because the LED proved so useful for debugging and signaling, one is now in permanent residence to help you out.

For this next bit, you need a loose LED from your kit. LEDs come in a variety of shapes, colors, and sizes but should look something like the one in Figure 3-9.



**FIGURE 3-9:**  
A lone LED, ready to be put to work.

Take a look at your LED and note that one leg is longer than the other. Place the long leg (anode or +) of the LED in pin 13 and the short leg (cathode or nd) in GND (ground). See Figure 3-10. You see the same blink, but it is (I hope) bigger and brighter depending on the LED you use.



**FIGURE 3-10:**  
Arduino  
LED pin 13.

As mentioned previously, the `LED_BUILTIN` variable is a predefined variable that refers to the onboard LED. Because the LED is controlled by pin 13 on your Arduino, it's possible to control a different component by connecting it to pin 13. The

voltage supplied by all other pins can be too high for most LEDs. Fortunately, another feature of pin 13 is its built-in, pull-down resistor. This resistor keeps your LED at a comfortable voltage and ensures that it has a long and happy life.

## Tweaking the Sketch

I've gone over this sketch in great detail, and I hope everything is making sense. The best way to understand what is going on, however, is to experiment! Try changing the delay times to see what results you get. Here are a couple of things you can try:

- » Make the LED blink the SOS signal.
- » See how fast you can make the LED blink before it appears to be on all the time.

While experimenting, it's wise to save your sketch under a different name. If the name is descriptive, such as `sosLED`, you can more easily find your project again. Also, each time you make a change to your code, remember to repeat the steps in this chapter to verify the code and upload it to your Arduino.

2

# Getting Physical with Arduino

## **IN THIS PART . . .**

Find out more about the prototyping tools you need to build your projects.

Dip into a bit of electronics theory.

Discover new and interesting things that your Arduino can do by building a few basic examples.

#### IN THIS CHAPTER

- » Introducing the breadboard, a blank canvas for circuits
- » Assembling a tool kit
- » Becoming an electronics super sleuth, with a multimeter

## Chapter 4

# Tools of the Trade

In Chapter 3, I cover one of the most basic Arduino applications: blinking an LED. This application requires only an Arduino and a few lines of code. Although blinking an LED is fun, you can use an Arduino for an almost unlimited number of other things — making interactive installations, controlling your home appliances, and talking with the Internet, to name a few.

In this chapter, you branch out by gaining an understanding of prototyping and how to use some basic prototyping tools to do more with your Arduino. Prototyping tools allow you to make temporary circuits to try new components, test circuits, and build simple prototypes. In this chapter, you find out about all the equipment and techniques you need to build your own circuits and prototype your ideas.

## Finding the Right Tools for the Job

*Prototyping* is all about exploring ideas, which also happens to be the core of what Arduino is all about. Although theory is important, you often learn better and faster from doing an experiment.

This section introduces you to some prototyping tools and components that you can use to start building circuits. You can then use these circuits to form the basis for your own projects.

A great number of tools are at your disposal to help you experiment. This section contains a short list of the recommended ones. Breadboards and jumper wires are included in most kits and are essential for building circuits for your Arduino project. Needle-nose pliers are not essential but are highly recommended.

## Breadboard

*Breadboards* are the most essential part of your prototyping kit. They are the base on which you can prototype your circuits. Breadboards allow you to temporarily use components rather than committing them to a circuit and soldering them in place. (Soldering is covered in Chapter 9.)

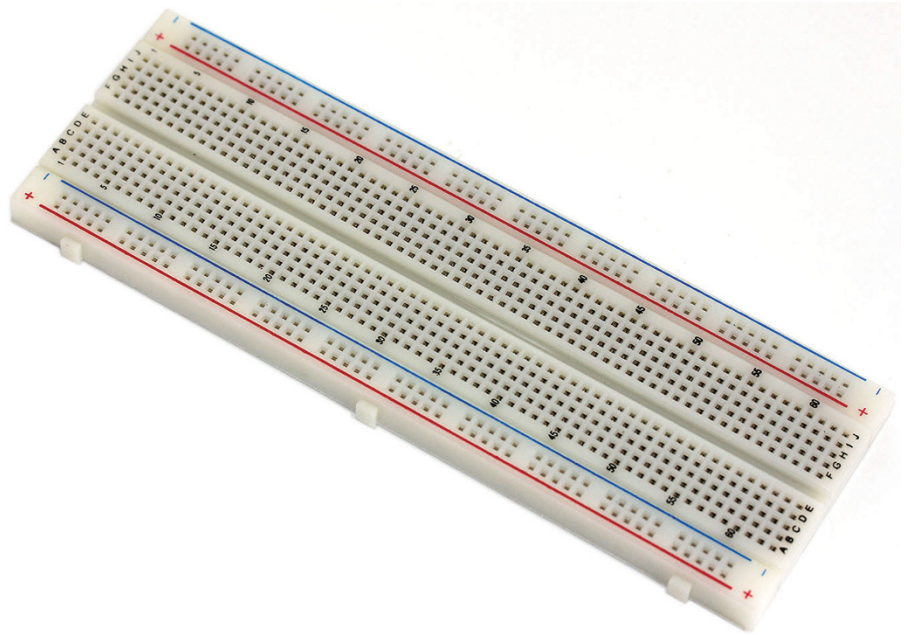
Breadboards get their name from a practice used in the early 1900s. At that time, components were a lot bigger, and people would prototype circuits by fixing them to an actual breadboard — that is, a board intended for cutting bread on. The components were joined by wrapping lengths of wire around nails that came in contact with the components. By unwrapping the wire from one nail and wrapping it around another, you could quickly change the circuit.

Modern breadboards are much more refined. The outside consists of a plastic case with rows and columns of holes, underneath which are tracks of copper. These tracks allow you to quickly and easily connect components electrically.

Breadboards come in different shapes and sizes, and many have connectors to allow you to lay them out in the arrangement you need.

Figure 4-1 shows a fairly standard breadboard. If you were to remove the cream-colored plastic coating, two parallel copper tracks would run down each of the long sides of the board. These copper lengths are generally used to provide a source of power (PWR) and ground (GND) and are sometimes marked with a positive (+) or negative (–) symbol or a red and black or a red and blue line. Red is always positive (+) and black or blue is negative (–).

A *power rail*, or *ground rail*, is basically a source of voltage or ground. Circuits often need power or ground for a variety of different functions, and one source often isn't enough. When you have multiple wires that all need to get to the same place, you use a rail. From this rail, jump wires can source whatever is needed.



**FIGURE 4-1:**  
A breadboard.



TIP

Although these tracks are marked, you can use them for anything. However, keeping to convention ensures that other people can easily understand your circuit, so I advise that you do so.

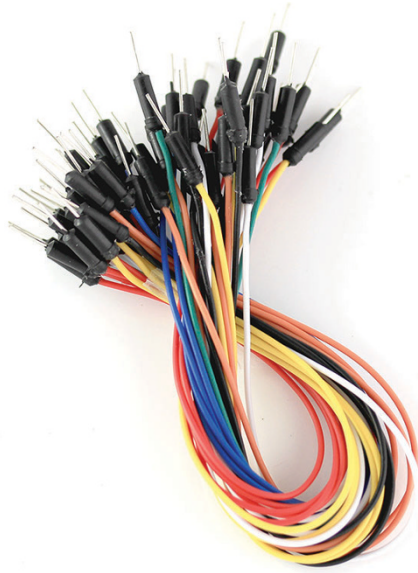
Down the middle of the breadboard are lots of short tracks running parallel to the short edge, separated with a trench down the middle. You use this middle trench to mount components and integrated circuits, including pushbuttons (discussed in Chapter 6) and optocouplers (see the bonus chapter “Hacking Other Hardware” at <https://www.dummies.com/go/arduino4d>). The trench makes it easier to lay out your circuit and also provides space for jump wires to connect your components to the places they need to get to.

When you place a jump wire or a component into one of the breadboard’s sockets, you should feel a bit of friction. A pincer-like device holds the wire or component in place. It provides enough grip to hold things in place for a project while working at a desk, but it’s loose enough for you to easily remove it with your fingers.

I have seen people taping breadboards into boxes for their projects, but this practice isn’t advisable. Smaller components or loose wires can easily come loose, and it can be a pain to figure out why something isn’t working. If you have a project working and want to get it out into the real world, jump ahead to soldering (explained in Chapter 9). Soldering makes your project last longer, and it’s a lot of fun, too!

## Jump wires

*Jump wires* (shown in Figure 4-2) are short lengths of insulated equipment wire. You use them to connect your components to the rows of your breadboard, other components, and your Arduino. A jump wire is no different from other wire in a material sense, but it's usually cut to a short length that is useful for breadboards.



**FIGURE 4-2:**  
A selection of  
jump wires.

You can find insulated wire everywhere. It can be the thick power cord used to plug in a household appliance or it can be much thinner, such as wire used for an earphone. Insulated wire is basically a conductive metal wire surrounded by an insulator that protects you from the electricity and protects the electrical signal from any outside interference.

The wire you use most often in this book (and most useful for your Arduino projects) could be subcategorized as *insulated equipment wire*. This type of wire is generally used on a small scale for low-voltage electrical applications.

The equipment wire you use is one of two varieties: single core and multicore. *Single-core wire* is a single piece of wire, just as a coat hanger is a single piece. Single-core wire is extremely good for holding its shape but shears if you bend it too much. Therefore, this type of wire is useful for laying out wires on your breadboard neatly as long as you won't need to move them much.

*Multicore wire* can have the same diameter as single core but consists of lots of little wires instead of just one wire. The little wires are twisted together, giving the multicore wire more strength and resistance to bending than a single-core wire. This twisted design is the same technique used on suspension bridges. Multicore wires are especially suited to connections that will change often, and they last a long time.

You can cut jump wires yourself, but they also come in handy packs of assorted colors and lengths. Cutting them yourself is significantly cheaper because you can buy a large reel of wire. On the other hand, if you want a variety of colors, you have to buy a reel for each color, which can amount to a fairly big investment for your first circuit. The packs save you this initial cost and give you the variety you need. You can buy the large reels when you know you'll need them.

Also, when considering cutting your own wire, bear in mind the difference in finishing for homemade jump wires. Single-core wires are much the same whether you or someone else cuts them yourself, but they deteriorate quickly as they bend. Multicore wires last longer, but if they are cut from a reel, you're left with a few small wires on the ends that can easily fray in the same way as the end of a piece of string or thread does. You have to fiddle with them often, twisting with your thumb and forefinger between uses to ensure that the small wires stay as straight and rigid as possible.

Premade multicore jump wires are usually either soldered together into a single point or have a connecting pin soldered to the end of the strands of wire. This design ensures that the connecting end is as reliable as single core while giving you the flexibility of multicore.

Ideally, you should have a pack of premade multicore jump wires to get you going. These packs are the most versatile and long-lasting choice for your prototyping kit. Eventually, you'll want to have a variety of jump wires so that you can be prepared for any situation when building a circuit.

## Needle-nose pliers

*Needle-nosed pliers*, shown in Figure 4-3, are the same as regular pliers but with a very fine point, ideal for picking up tiny components. Electronics can be an extremely fiddly business, and it's extremely easy to mangle the delicate legs of components when pushing them into a breadboard. These specialist pliers are not essential but add a little bit of finesse to building circuits.

**FIGURE 4-3:** Needle-nose pliers are essential for detailed work.



## Multimeter

A *multimeter* is a meter that measures volts, amps, and resistance. It can tell you the values of different components and what's going on with different parts of your circuit. Because you can't see what's going on in a circuit or component, a multimeter is essential for understanding its inner workings. Without it, you have to rely on guesswork, which is always a bad idea with electronics.

Figure 4-4 shows a good mid-range digital multimeter. As does this one, most multimeters include:

- » **A digital display:** This display looks like the one on your digital alarm clock and shows the values you're reading. Because of the limited number of digits, the decimal place moves to accommodate a larger or smaller number. A multimeter with an auto-ranging function finds the number automatically. If your multimeter doesn't have this feature, you must change the mode manually to the range you require.
- » **A mode-selection dial:** This dial allows you to choose among the different functions on the multimeter. These functions can be for volts, amperes, and ohms as well as for the range within each of these, such as hundreds, thousands, tens of thousands, hundreds of thousands, and millions of ohms.



TIP

The best multimeters also include a continuity tester, which tells you whether your connections are actually connected by sounding a tone. Having a tone has saved me hours of work retracing my steps on projects, so I definitely recommend investing in a good mid-range multimeter with this feature.

- » **A set of probes:** Probes (also called *test leads*) are the implements you use to test parts of your circuit. Multimeters come with two skewer-like probes that are designed to fit into tight places and are most useful for making contact with small components on circuit boards. You can also find probes with crocodile clips on the end, or you can simply buy crocodile clips that you can attach yourself. These clips are especially useful for grabbing onto wires.

» **A set of sockets:** The probes can be repositioned into different sockets depending on the use. In this case, the sockets are labeled A, mA, COM, and V $\Omega$ Hz. The socket marked A is for measuring large currents in amps (A), up to 20A. Note that the probe can read that current for only 10 seconds. These limits are indicated on a line between this socket and the COM socket, which also indicates that the two probes should be placed in A (red probe) and COM (black probe). The mA socket (red probe) is for smaller currents that are less than 500mA. The COM socket (black probe) is short for Common and is a point of reference for your measurements. In most cases, COM is the ground of your circuit and uses the black probe. The socket marked V $\Omega$ Hz (red probe) is used for measuring the voltage (V or volts), resistance ( $\Omega$  or ohms), and frequency (Hz or hertz) between this socket and the COM port (black probe).

**FIGURE 4-4:**  
A good digital  
multimeter can  
often save  
your project.



## Using a Multimeter to Measure Voltage, Current, and Resistance

All Arduinists should know a few basic techniques for checking their circuits. Volts, amps, and current can all be calculated in advance (as you learn in Chapter 5), but in the real world, many other factors can arise that you can't account for. If you have a broken component or a faulty connection, you can lose hours guessing what could be wrong, so a multimeter becomes essential for solving problems in your circuit. In this section, you learn about measuring voltage, current, and resistance and checking the continuity of your connections.

## Measuring voltage in a circuit

Measuring the voltage is essential, whether you're checking the voltage of a battery or the voltage passing through a component. If things aren't lighting up or whirring, a connection might be loose or you might have sent too much power and burnt out what you were trying to power. When something isn't working, it's time to check the voltage in your circuit and make sure it is correct.

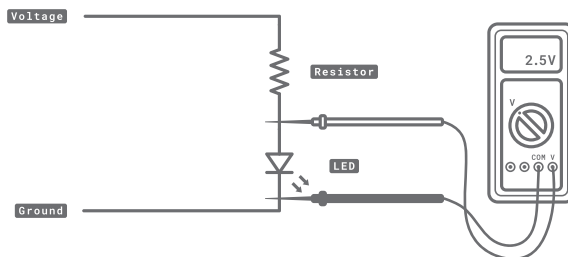
First you need to check that the probes of your multimeter are in the correct sockets. Insert the red probe in the socket marked V, for volts, and insert the black probe in the socket marked COM, for ground. Next, set your multimeter to volts by using the dial in the center of the multimeter.

Some multimeters, such as the one in Figure 4-4, have a button to toggle between DC and AC. On other multimeters, a DC voltage is signified by a V followed by a square-shaped digital wave, as opposed to an AC voltage, which is indicated by a smooth, analog wave.



TECHNICAL  
STUFF

Voltage is measured in parallel. To measure in parallel you must bridge the part of the circuit that you want to measure without interfering in it. Figure 4-5 shows how you do this. The positive probe should always be on the positive side of the component and the negative on the other side. Getting the probes the wrong way won't cause any damage to your circuit but will give you a negative reading rather than a positive one.



**FIGURE 4-5:**  
A multimeter is used in parallel to find the voltage.

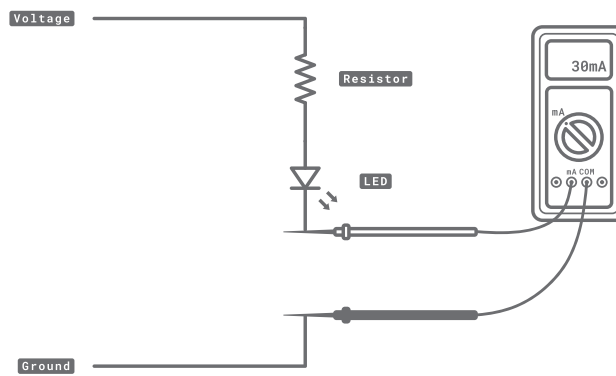
A good way to test your multimeter is to measure the voltage between the 5V pin and GND on your Arduino. Make sure that your Arduino is plugged in, and connect a jump wire to each pin to make them easier to access with your probes. Place the red voltage probe on the 5V wire and the black common probe on the GND wire. Doing so should return a value of 5V on the multimeter screen and prove that your Arduino is supplying 5 volts as expected.

## Measuring current in a circuit

You may have the right voltage, but there isn't enough current to power the light or motor you're driving. The best way to find out is to check the circuit to see how much current is being drawn and compare that to the power supply you're using.

Check that your probes are connected to the correct sockets of the multimeter. Some meters have two possible sockets, one for very high currents measured in amps (or A) and another for low currents measured in milliamps (mA). Most basic Arduino circuits require only a reading in milliamps, but circuits with large lights, motors, or other devices, require a reading in amps. Then turn the dial on your meter to select the correct level of amps, A or mA or even  $\mu\text{A}$  (microamps).

Current is measured in series, which means that the multimeter must be placed in line with the other components in the circuit so that the current flows through the multimeter as if it were another component. Figure 4-6 shows this series measurement in action.



**FIGURE 4-6:**  
A multimeter is used in series to find the current.

By using the 5V and GND pins on your Arduino, you can use jump wires to build the circuit in Figure 4-6. By using your multimeter to connect the LED and GND, you will be able to read the current in the circuit. You'll learn how to build circuits similar to Figure 4-6 in Chapter 6.

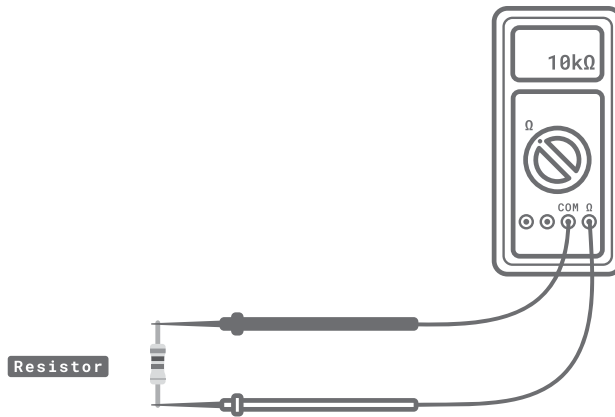


If you're blinking or fading the output, your current changes, so you may want to set it to always be on to be sure of the maximum continuous current.

## Measuring resistance of a resistor

Sometimes it can be difficult to read the value of a resistor, and it is necessary, or just easier, to confirm it with a multimeter. Simply set the multimeter to ohms or  $\Omega$  and place one probe on each leg of the resistor, as shown in Figure 4-7.

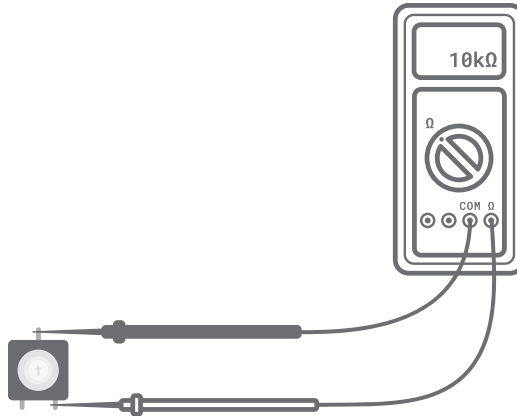
**FIGURE 4-7:**  
Finding the  
resistance of  
a resistor.



## Measuring resistance of a variable resistor

With variable resistors, it can be good to know that you're getting the full range of resistances promised on the label. Variable resistors are similar to passive resistors but have three legs. If you connect the probes across the legs on either side, you are reading the maximum resistance of the variable resistor, and the reading does not change when you move the dial. If you place the probes between the center and one side of the resistor, you are reading the value of variable resistance, which changes as you turn the dial, as shown in Figure 4-8. If you switch to the center and the opposite side, you change the direction of the dial.

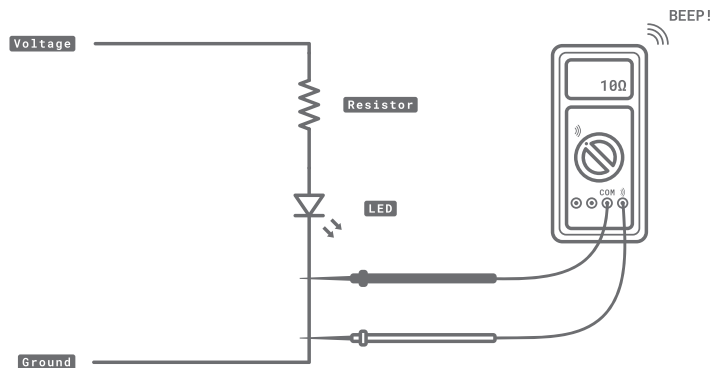
**FIGURE 4-8:**  
Finding the  
resistance of a  
variable resistor.



## Checking the continuity (in beeps) of your circuit

If you have a high-quality multimeter, it should have a continuity tester, represented by a speaker or sound symbol on the dial. You use the continuity tester to verify that parts of your circuit are connected; the multimeter communicates the connection by beeping, ideally producing a continuous tone when the connection is good.

First test the continuity feature itself by turning your dial to the continuity test symbol and touching the ends together. If you hear an unbroken tone, the feature is working. Next, test the connection in your circuit by placing the probes along any length of wire or connection, as shown in Figure 4-9.



**FIGURE 4-9:** Checking the continuity of your circuit.



#### IN THIS CHAPTER

- » Coming to grips with electricity
- » Understanding a few useful equations
- » Finding your way around circuit diagrams
- » Realizing the importance of color coding

## Chapter 5

# A Primer on Electricity and Circuitry

In this chapter, you look at the fundamentals of electricity. In later chapters, you delve deeper into electronics, so it's important that you have a basic understanding of how electricity behaves in your circuit.

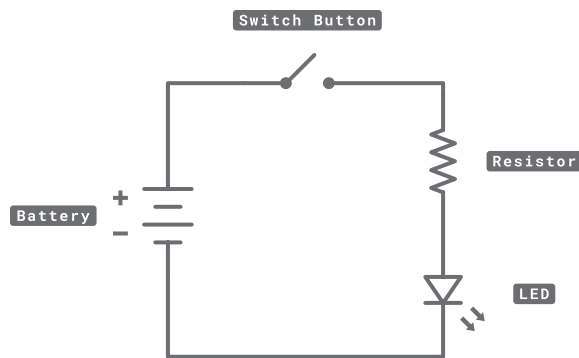
The great thing about Arduino is that you don't need to study electronics for years to use it. That said, it's a good idea to know a bit of the theory to back up the practical side of your project. In this chapter, you look at a few equations to help you build a balanced and efficient circuit, you look at circuit diagrams, which provide you with a roadmap of your circuit, and you learn a bit about color coding, which can make your life easier when building circuits.

## Understanding Electricity

Most people take electricity for granted but find it difficult to define. Simply put, *electricity* is a form of energy resulting from the existence of charged particles (such as electrons or protons), either statically as an accumulation of charge or dynamically as a current.

This definition of electricity is describing electricity on an atomic level, which is more complex than you need to know when dealing with the circuitry in your Arduino projects. Your main concern is simply to understand that electricity is energy and has a current. For those of you who want to understand electricity at this level, you can check out *Electronics For Dummies*, 3rd Edition, by Cathleen Shamieh.

To illustrate the idea of the flow of a current, take a look at a simple light switch circuit (see Figure 5-1). The circuit is similar to those you may have made in physics or electronics classes at school with no Arduino involved, using only a battery, a switch, a resistor, and an LED.



**FIGURE 5-1:**  
A basic light  
switch circuit.

In this circuit, you have a source of electrical power in the form of a battery. Power is supplied in watts and is made up of voltage (in volts) and current (in amps). Voltage and current are supplied to the circuit through the positive (+) end of the battery.

You use a switch to control the power to the circuit. The switch can either be open, which breaks the circuit and stops the flow of current, or closed, which completes the circuit and allows it to function.

The power can be used for various applications. In this case, the circuit is powering an LED. The battery is supplying the LED with 4.5V, which is more than the LED needs to light. If the LED were supplied with this much voltage, you would risk damaging it, so you need a resistor before it to resist the voltage. Also, if the voltage is too low, the LED will not reach full brightness.

To complete the circuit, the power must return to ground at the negative (-) end of the battery. The LED draws as much current as necessary to light to full brightness.

By drawing current, the LED is also resisting the flow, ensuring that only the required current is drawn. Unless the current is used or resisted by components, the circuit draws all the available current as quickly as possible. This is known as a *short circuit*. An example of a short circuit is connecting the positive directly to the negative, with no components in between.

The basic principles that you need to understand follow:

- » An electrical circuit is, as the name suggests, a circular system.
- » The circuit needs to use the power inside it before it returns to the source.
- » If the circuit does not use the power, that power has nowhere to go and can damage the circuit.
- » The easiest way to interact with a circuit is to break it. By controlling when and where the power is, you have instant control over the outputs.

## Using Equations to Build Your Circuits

You are now aware of a few characteristics of electricity:

- » Power (P) is in watts, such as 60W
- » Voltage (V) is in volts, such as 12V
- » Current (I) is in amps, or amperes, such as 3A
- » Resistance (R) is in ohms, such as 150Ω

These characteristics can be quantified and put into equations, which allow you to carefully balance your circuit to ensure that everything works in harmony. A variety of equations exist for determining all manner of attributes, but in this section I cover two basic ones that will be the most useful to you when working with Arduino: Ohm's Law and Joule's Law.

### Ohm's Law

Perhaps the most important relationship to understand is that among voltage, current, and resistance. In 1827, Georg Simon Ohm discovered that voltage and current were directly proportional if applied to a simple equation (recall from the preceding list that *I* stands for current, measured in *amps*, or *amperes*):

$$V = I \times R$$

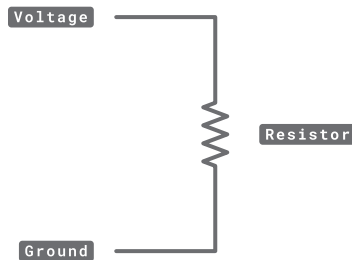
This equation became known as *Ohm's Law*. Using algebra, the equation can be rearranged to give you any one value from the remaining two:

$$I = \frac{V}{R}$$

or

$$R = \frac{V}{I}$$

You can take a look at this equation at work in an actual circuit. Figure 5-2 shows a simple partial circuit with a power source and a resistor.



**FIGURE 5-2:**  
A power source  
and resistor.

In many situations, you will know the voltage of your power supply and the value of the resistor, so you can first calculate the current of the circuit as follows:

$$I = \frac{V}{R} = \frac{4.5V}{150\Omega} = 0.03A$$

This equation works in any order you want to put the values:

$$R = \frac{V}{I} = \frac{4.5V}{0.03A} = 150\Omega$$
$$V = I \times R = 0.03A \times 150\Omega = 4.5V$$

The easiest way to remember Ohm's Law is as a pyramid (see Figure 5-3). By eliminating any one element from the pyramid, you are left with the equation.



**FIGURE 5-3:**  
Ohm's Law  
pyramid.

“But how is this calculation useful to me in the context of Arduino?” I hear you cry. Here’s a practical example that you might run into in a basic Arduino circuit.

The digital pins on an Arduino can supply up to 5V, so this is the most common supply of power that you use. An LED is one of the most basic outputs you want to control, and a fairly standard LED requires a voltage of 2V and about 30 milliamps or 30mA (0.03A) of current.

If you were to plug in the LED directly to the power supply, you would promptly witness a bright light followed by a plume of smoke and a burning smell. You’re not likely to want that! To make sure that you can use the LED again and again safely, you should add a resistor.

Ohm’s Law tells you that

$$R = \frac{V}{I}$$

But you also have to include two different voltage values, the voltage of the power supply (supply voltage) and the voltage required to power the LED (forward voltage). *Forward voltage*, a term that is often found in datasheets, especially when referring to diodes, indicates the recommended amount of voltage that the component can take in the direction that the current is intended to flow. For LEDs, this direction is from anode to cathode, with the anode connected to positive and the cathode to negative.

When referring to non-light-emitting diodes (covered in Chapter 7), you’re using them to resist the flow of current in the opposite direction, from cathode to anode. In this case, the term is *reverse voltage*, which indicates the value in volts that the circuit must exceed for current to flow through the diode.

In this case, the voltages are labelled  $V_{\text{SUPPLY}}$  and  $V_{\text{FORWARD}}$ , respectively. The Ohm’s Law equation requires the voltage across the resistor (voltage that passes through the resistor), which is equal to the supply voltage minus the LED forward voltage, or

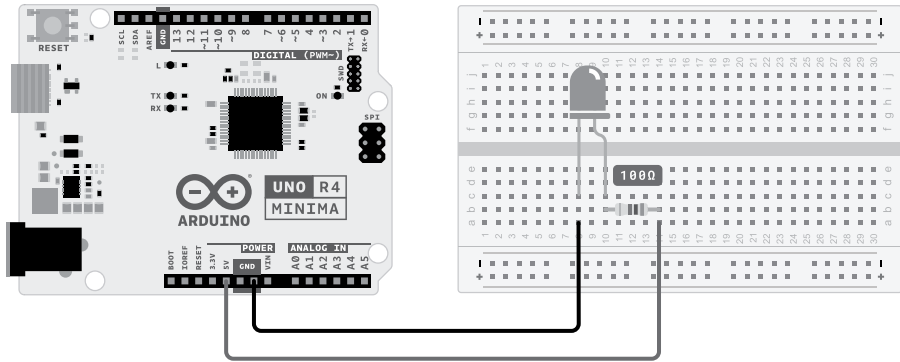
$$V_{\text{SUPPLY}} - V_{\text{FORWARD}}$$

The new equation looks like this:

$$R = \frac{V_{\text{SUPPLY}} - V_{\text{FORWARD}}}{I} = \frac{5\text{V} - 2\text{V}}{0.03\text{A}} = 100\Omega$$

This tells you that you need a 100-ohm resistor to power an LED safely; the example circuit is shown in Figure 5-4.

**FIGURE 5-4:**  
Applying Ohm's  
Law to an  
Arduino circuit.



## Calculating power

To calculate the power consumption of your circuit in watts, you multiply the voltage and current of the circuit. The equation is

$$P = V \times I$$

If you apply this equation to the same circuit shown earlier in the chapter (see Figure 5-4), you can calculate its power:

$$P = (V_{\text{SUPPLY}} - V_{\text{FORWARD}}) \times I = (5V - 2V) \times 0.03A = 0.09W$$

This value represents the power dissipated by the 100  $\Omega$  resistor, which drops the remaining 3V between the Arduino's 5V supply and the LED's forward voltage. The LED also dissipates power, but that depends on its forward voltage rather than its resistance.

This algebra works in the same way as Ohm's Law and can be reconfigured to find the missing value:

$$V = \frac{P}{I}$$

$$I = \frac{P}{V}$$

This calculation is useful because some types of hardware, such as a light bulb, show only the power and voltage rating, leaving you to figure out the current draw. The calculation is especially useful if you are trying (or failing) to run power-hungry devices, such as lighting or motors, off your Arduino pins.

A USB-powered Arduino is still limited by how much current the computer's USB port can supply (typically around 500 mA), but the way that current is available on the board has changed with newer models. On the Arduino Uno R4 Minima, each digital or analog I/O pin can safely source or sink about 8 mA of current, which is



You can also do the same calculation for situations where you know only the voltage and resistance:

$$\begin{array}{l} \text{If, } I = \frac{V}{R(\text{Ohm's Law})} \text{ and } P = \frac{I}{V(\text{Power Calculation})} \\ \text{Then, } P = (V/R) \times V \end{array}$$

Which can also be written as

$$P = \frac{V^2}{R}$$

Try the same circuit again to check the results:

$$\frac{(5V - 2V) \times (5V - 2V)}{100\Omega} = 0.09W$$

This equation can also be rearranged into any combination, depending on which values you know:

$$\begin{array}{l} V = \frac{P}{V \times R} \\ R = \frac{V^2}{P} \end{array}$$

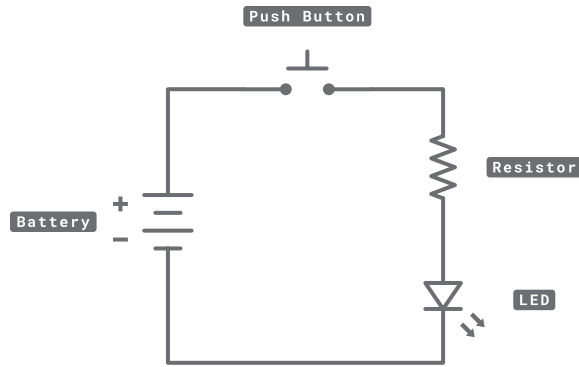
Many Arduinists, myself included, are more practical than theoretical, attempting to build the circuit based on examples and documentation before doing the sums. This approach is perfectly all right and in the spirit of Arduino! In most cases, your circuit will have the desired outcome. However, with these few equations, it's possible to fill in the blanks in most circuits and ensure that everything is in order.

## Working with Circuit Diagrams

Recreating circuits from photos or illustrations can be difficult, and for that reason, standardized symbols are used to represent the variety of components and connections in a circuit. These *circuit diagrams* are like maps of the subway system: They show you every connection clearly but have little resemblance to the way things look or connect in the physical world. The following sections delve a bit more into circuit diagrams.

### A simple circuit diagram

This section looks at a basic light switch circuit made up of four components: a battery, a pushbutton, a resistor, and an LED, as shown in Figure 5-5.



**FIGURE 5-5:**  
A simple  
light switch  
circuit diagram.

Table 5-1 shows the individual symbols for each component.

**TABLE 5-1**

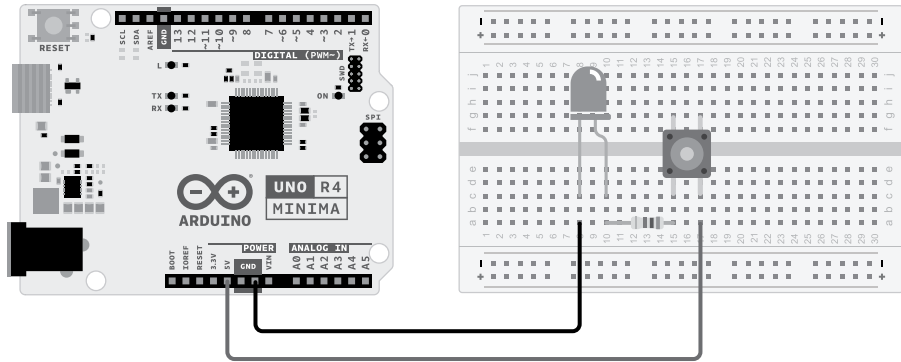
### Basic Symbols

Name	Symbol
Battery	
Pushbutton	
Resistor	
LED	

Figure 5-6 shows the same circuit laid out on a breadboard. The first thing you may notice is that this example has no battery. Because your Arduino has a 5V pin and a GND pin, these take the place of the positive (+) and negative (-) of the battery and allow you to make the same circuit. The second thing you may notice is that the circuit uses a pushbutton and, therefore, is not technically a light *switch*. Many pushbuttons are made to fit a breadboard, which is more convenient.

I find that the best way to compare a circuit diagram to the actual circuit is to follow the connections from positive to negative.

**FIGURE 5-6:**  
A simple light switch circuit laid out on a breadboard.



If you start at the positive (+) 5V pin on the Arduino, it leads to the pushbutton. The physical pushbutton has four legs, whereas the symbol has only two. The legs of the physical pushbutton are mirrored so that two are connected on one side and two on the other. For this reason, it's important to get the orientation of the pushbutton right. The physical switch has four legs to make it more versatile, but as far as the circuit diagram is concerned, there is only one switch with one line in and one line out.

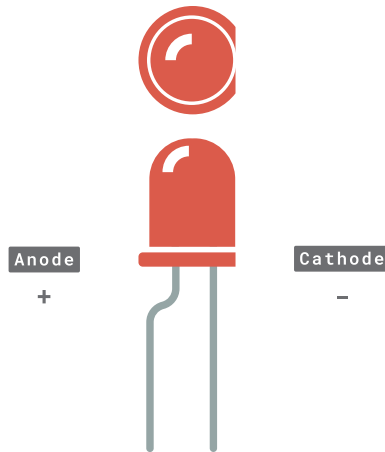
The other side of the pushbutton is connected to a resistor. Other than the fact that the resistor symbol in the diagram is not as bulbous as the physical resistor, the diagram and physical resistor match up well; one wire goes into the resistor and another goes out. The value of the resistor is written alongside the component, as opposed to the color-coded stripes on the physical one. Resistors do not have polarity (no positive or negative), so there is nothing else to show.

An LED, by contrast, does have a polarity. If you connect it the wrong way around, it won't illuminate. In the circuit diagram, the polarity of the LED is marked by the triangle in the symbol, pointing in the direction of the current flow from + (anode) to - (cathode), with a horizontal line as a barrier in the other direction. On the physical LED, a long leg marks the anode (+) and the flat section on the side of the lens marks the cathode (-), in case the legs are chopped off. See Figure 5-7.

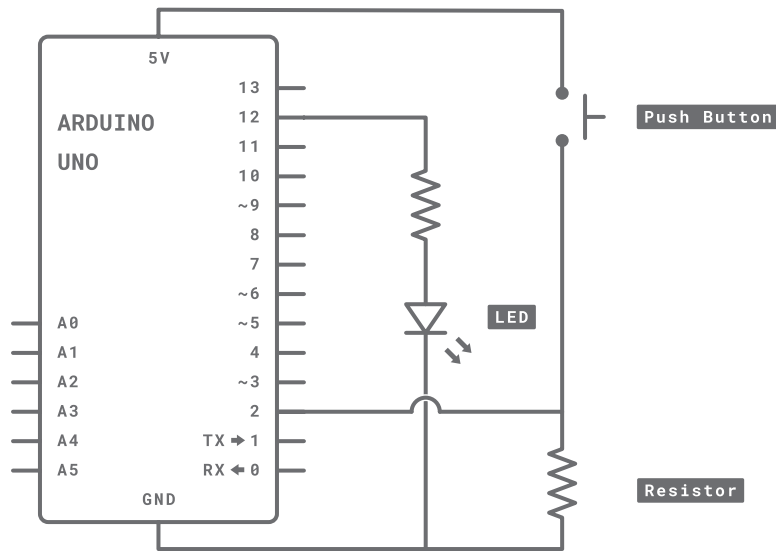
The - (cathode) of the LED is connected to the negative (-) GND pin on the Arduino and then to the negative terminal of the battery to complete the circuit.

## Using a circuit diagram with an Arduino

Although it's useful to understand the simple circuit in Figure 5-6, you will most likely be using an Arduino in your circuit somewhere, so take a look again at the same circuit powered from an Arduino (see Figure 5-8).



**FIGURE 5-7:** A long leg or a flat side marks an LED's polarity.



**FIGURE 5-8:** A light switch circuit running through an Arduino.

This circuit has more components than the circuit described in the previous section.

The Arduino is on the left in the diagram. This symbol is standard for an integrated circuit and is similar to its physical representation — a rectangle with lots of legs poking out. All the legs or pins are labeled so that you can tell them apart.

Also, rather than have one circuit, the diagram shows two, each running back to the Arduino, to illustrate how the Arduino fits in with conventional circuits. Instead of switching the power on and off, you're sending a signal to the Arduino, which interprets it and outputs it to the LED.

Here is a great practice to adopt when you have a complicated circuit: Rather than tackle it as a whole, break it up into its components. This circuit has one input circuit and one output. I describe this circuit in more depth in Chapter 6.

## Color-Coding

An important technique in electronics is color-coding, and it becomes even more important as you progress to more complex circuits. Wiring circuits correctly can be hard enough, but staring at a sea of same-colored wires makes the task infinitely harder.

You're probably aware of color-coding even if you're new to electronics. Traffic lights, for example, are color coded to give drivers a clear message of what to do:

- » Green means proceed.
- » Amber means prepare to stop.
- » Red means stop.

Color-coding is a quick and easy way to visually get a message across without lots of words.

All sorts of electrical applications, such as the 120v or 240v plugs and sockets in your home, are color-coded. Because plugs and sockets are widely used and potentially dangerous, the colors need to be consistent from plug to plug and match national standards. This color-coding makes it easy for any electrician or DIY enthusiast to make the correct connections.

Because you will be working with low-voltage DC electronics, you have less potential for causing yourself serious harm, but you still have great potential for destroying the delicate components in your circuit. No definitive rules exist for organizing your circuit, but here are a few conventions that can help you and others know what's going on:

- » Red is positive (+).
- » Black is negative (-).
- » Different colors are used for different signal pins.

These conventions are true on most breadboard circuits. Power and ground colors can change; for example, they can be white (+) and black (-) or brown (+) and blue (-), sometimes depending on the wire that is available to the person. As long as you use a color-coding system of some sort (and use it consistently), reading, copying, and fixing the circuit will be easier.

I've fixed many circuits that were broken because of the simple error of connecting the wires to the wrong places.

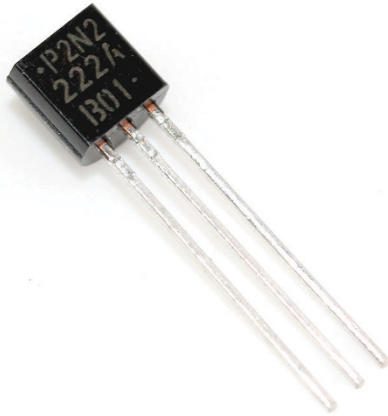
If the color-coding of the wire is ever questionable, check the connection (using the continuity checker on your multimeter) or the voltage running through the wire (using the voltage meter on your multimeter) to make sure that everything is as expected.

## Datasheets

Picture the scene. Your friend has heard that you know a bit of electronics and has asked you to look at a circuit, copied from the Internet, that isn't working. But the board has lots of nondescript integrated circuits, so what do you do? The answer: Google!

The world contains millions, if not billions, of different components. The information you need to make sense of them is normally presented in the form of a *datasheet*. Every component should have a datasheet, provided by the manufacturer, that lists every detail of the component (often more detail than you need).

The easiest way to find a datasheet is to Google it. To find the right one, you need to know as much about the component as you can find out. The most important information for search purposes is the model number of the component. Figure 5-9 shows the model number of a transistor: P2N2 222A B01. If you Google that number plus the word *datasheet*, you should locate numerous PDF files that provide details about the component. If you can't find a model or part number, try to find it out from the place where you purchased the component.



**FIGURE 5-9:**  
The small print  
on this transistor  
tells you exactly  
what it is.

## Resistor Color Charts

Resistors are extremely important to Arduino projects, and you can find a great variety of them to allow you to finely tune your circuit. Resistors can also be extremely small, making it impossible to write the resistance value on the resistor. For this reason, a color chart system exists to tell you what you need to know about these tiny components. If you take a close look at a resistor, such as the one in Figure 5-10, you can see a number of colored bands, which indicate the value in ohms of the resistor.



**FIGURE 5-10:**  
The color bands  
on a resistor.

A standard resistor has four bands that represent the following:

- » First digit
- » Second digit
- » Multiplier
- » Tolerance



TIP

Resistors are small and some colors can be difficult to distinguish unless you have extremely good vision. You can use a magnifying glass or smartphone camera to see the colors of the bands more clearly.

Table 5-2 lists the value in ohms, the multiplier, and the tolerance value that each color represents. First, you need to know the order in which to read the bands. Normally, you will see an equal-sized gap between the first three bands and a larger gap separating the fourth tolerance band.

**TABLE 5-2** Resistor Color Chart

Color	Value	Multiplier	Tolerance
Black	0	$\times 10^0$	-
Brown	1	$\times 10^1$	$\pm 1\%$
Red	2	$\times 10^2$	$\pm 2\%$
Orange	3	$\times 10^3$	-
Yellow	4	$\times 10^4$	$\pm 5\%$
Green	5	$\times 10^5$	$\pm 0.5\%$
Blue	6	$\times 10^6$	$\pm 0.25\%$
Violet	7	$\times 10^7$	$\pm 0.1\%$
Gray	8	$\times 10^8$	$\pm 0.05\%$
White	9	$\times 10^9$	-
Gold	-	$\times 10^{-1}$	$\pm 5\%$
Silver	-	$\times 10^{-2}$	$\pm 10\%$
None	-	-	$\pm 20\%$

For example, here a few values you might find in your kit:

Orange (3), Orange (3), Brown ( $\times 10^1$ ), Gold ( $\pm 5\%$ ) =  $33 \times 10 = 330$  ohms with  $\pm 5\%$  tolerance

Red (2), Red (2), Red ( $\times 10^3$ ), Gold ( $\pm 5\%$ ) =  $22 \times 10 \times 10 = 2.2K$  ohms  $\pm 5\%$  tolerance

Brown (1), Black (0), Orange ( $\times 10^3$ ), Gold ( $\pm 5\%$ ) =  $10 \times 10 \times 10 \times 10 = 10K$  ohms  $\pm 5\%$  tolerance

Seeing the colors and sometimes even telling which end to start reading from can be difficult. So, in most situations, it's best to use a multimeter to check the value of your resistor in ohms. (Remember to set the dial on your multimeter to ohm, or  $\Omega$ .)



TIP

Resistors of the same value are often supplied on a reel of paper tape that holds the resistors together in a kind of ladder. This arrangement enables machines to feed in a reel of resistors in an orderly fashion before placing them on a PCB. If you write the value of that reel of resistors, you won't have to spend time reading or measuring the resistors each time you use them.

- » Fading like a pro
- » Understanding inputs
- » Varying resistances with potentiometers
- » Showing off your stats with the serial monitor

## Chapter 6

# Basic Sketches: Inputs, Outputs, and Communication

In this chapter, I discuss some of the basic sketches needed to get you on your Arduino feet. This chapter covers a broad range of inputs and outputs using the sensors in your kit. If you don't yet have a kit, I suggest reading through Chapter 2 to find one of the recommended ones.

The Blink sketch (described in Chapter 3) gives you the basis of an Arduino sketch. This chapter expands it by adding circuits to your Arduino. I walk you through building circuits using a breadboard, as mentioned in Chapter 4, and introduce a few new components from your kit.

I detail uploading the appropriate code to your Arduino, walk you through each sketch line by line, and suggest tweaking the code yourself to gain a better understanding of it.

# Uploading a Sketch

Throughout this chapter and much of the book, you learn about a variety of circuits, each with their respective sketches. The content of the circuits and sketches can vary greatly and are detailed in each of the examples in this book. Before you get started, you need to know one simple process for uploading a sketch to an Arduino board.

Follow these steps to upload your sketch:

- 1. Connect your Arduino using the USB-C cable.**
- 2. Select your board type and serial port.**

If you're using the Web IDE, it automatically detects the board type and port. You can find the full list of supported boards by clicking the board's drop-down menu and clicking Select Other Board and Port.

Using the downloadable IDE requires a few steps. Select your board type by choosing Tools ⇨ Board ⇨ Arduino UNO R4 Boards ⇨ Arduino UNO R4 Minima from the menu bar. You can find a list of all available serial ports by choosing Tools ⇨ Port ⇨ comX or /dev/cu.usbmodemXXXXX, where X is a sequentially or randomly assigned number.

In Windows, if you have just connected your Arduino, the COM port will normally be the highest number, such as com 3 or com 15. Many devices can be listed on the COM port list, and if you plug in multiple Arduinos, each one will be assigned a new number.

On macOS, the /dev/cu.usbmodem *number* will be randomly assigned and can vary in length, such as /dev/cu.usbmodem1421 or /dev/cu.usbmodem262471. Unless you have another Arduino connected, it should be the only one visible.

- 3. Click the Verify button.**

The Verify button is a check mark. Your code is checked to make sure that it can be understood by the IDE. Common mistakes such as typos will be highlighted.

- 4. Click the Upload button (arrow).**

The code is compiled and sent to the Arduino.

Now that you know how to upload a sketch, you should be suitably hungry for some more Arduino sketches. To help you understand the first sketch in this chapter, I first tell you about a technique called *pulse-width modulation (PWM)*. The next section briefly describes PWM and prepares you for fading an LED.

# Using Pulse-Width Modulation (PWM)

In Chapter 2, I mention that sending an analog value uses something called pulse-width modulation (PWM). This technique allows your Arduino, which is a digital device, to act like an analog device. In the following example, PWM allows you to fade an LED rather than just turn it on or off.

Here's how it works: A digital output is either on or off. But it can be turned on and off extremely quickly thanks in part to the miracle of silicon chips. If the output is on half the time and off half the time, it is described as having a 50 percent duty cycle. The *duty cycle* is the period of time during which the output is active, so that could be any percentage — 20 percent, 30 percent, 40 percent, and so on.

When you're using LEDs as an output, the duty cycle has a special effect. Because the LED is blinking faster than the human eye can perceive, an LED with a 50 percent duty cycle looks as though it is at half brightness. This same effect allows you to perceive still images shown at 24 frames per second (or above) as a moving image.

With a DC motor as an output, a 50 percent duty cycle has the effect of moving the motor at half speed. In this case, PWM allows you to control the speed of a motor by pulsing it at an extremely fast rate.

So despite PWM's being a digital function, it is referred to as `analogWrite` because of the perceived effect it has on components.

## The LED Fade Sketch

In this sketch, you make an LED fade on and off. In contrast to the sketch that resulted in a blinking LED in Chapter 3, you need some extra hardware to make the LED fade on and off.

For this project you need:

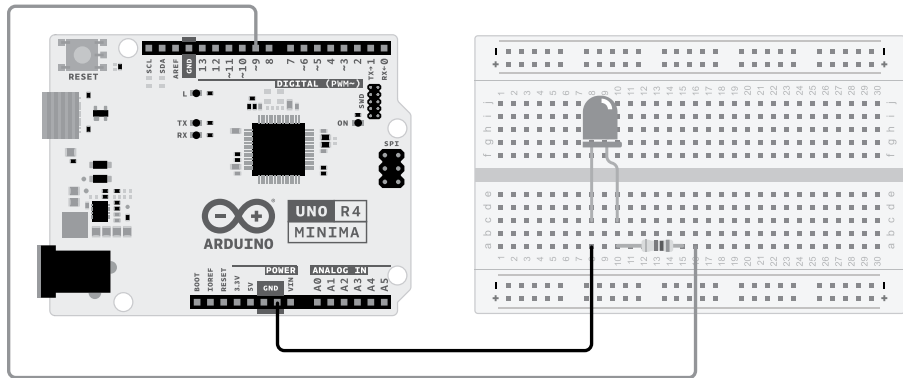
- » An Arduino Uno
- » A breadboard
- » An LED
- » A resistor (greater than 120 ohm)
- » Jump wires



WARNING

It's always important to make sure that your circuit is not powered while you're making changes to it. You can easily make incorrect connections, potentially damaging the components. So before you begin, make sure that the Arduino is unplugged from your computer or any external power supply.

Lay out the circuit as shown in Figure 6-1, which is a simple circuit like the one used for the Blink sketch in Chapter 3 but uses pin 9 instead of pin 13. Pin 9, unlike pin 13, is capable of pulse-width modulation, which is necessary to fade the LED. However, note that pin 9 requires a resistor to limit the amount of current supplied to the LED. (On pin 13, this resistor is included on the Arduino board itself.)



**FIGURE 6-1:** Pin 9 is connected to a resistor and an LED and then goes back to ground.

## PUTTING UP RESISTANCE

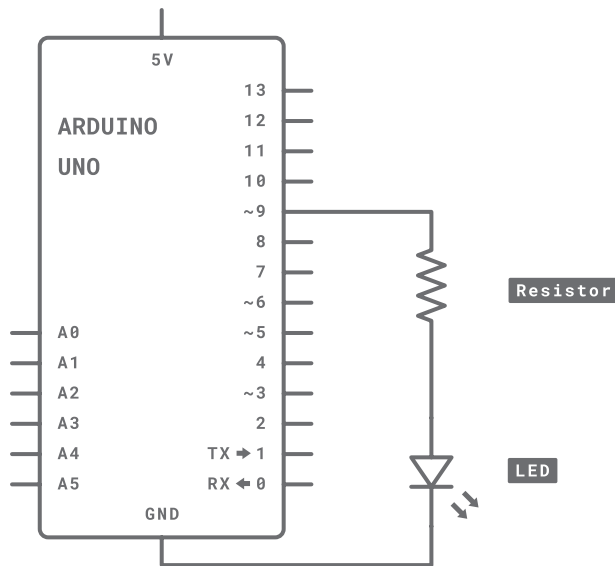
As you learn in Chapter 5, calculating the correct resistance is important for a safe and long-lasting circuit. In the circuit you're building in this chapter, you are potentially supplying your LED with a source of 5V (volts), the maximum that a digital pin can supply. A typical LED such as those in your kit has an approximate maximum forward voltage of 2.1V (volts), so a resistor is needed to protect it. The LED draws a maximum current of approximately 25mA (milliamps). Using these figures, you can calculate the resistance (ohms):

$$R = \frac{V_S - V_L}{I}$$
$$R = \frac{5 - 2.1}{0.025} = 116 \text{ ohms}$$

The nearest fixed resistor above this calculation that you can buy is 120 ohms (brown, red, brown), so you're in luck if you have one of those. If not, you can apply the rule of using the nearest resistor above this value. This resistor resists more voltage than the optimum, but your LED is safe and you can always switch out the resistor later when you want to make your project more permanent. Suitable values from various kits include 220Ω, 330Ω, and 560Ω.

You can always refer to Chapter 5 to find your resistor value on the color chart or use a multimeter to measure the value of your resistors. There are even smartphone apps that have resistor color charts (although this may be a source of great embarrassment and ridicule among friends).

The schematic in Figure 6-2 shows you the simple circuit connection. The digital pin, pin 9, is connected to the long leg of the LED; the short leg connects to the resistor, which goes on to ground, GND. In this circuit, the resistor can be either before or after the LED, as long as it is in the circuit.



**FIGURE 6-2:**  
A schematic of  
the circuit to  
fade an LED.



**TIP**

It's always a good idea to *color code* your circuits — that is, use various colors to distinguish one type of circuit from another. Doing so greatly helps keep things clear and can make problem solving much easier.

You should adhere to a few good standards. The most important areas to color code are power and ground. These are nearly always colored red and black, respectively, but you might occasionally see them as white and black as well, as mentioned in Chapter 5. The other type of connection is usually referred to as a *signal wire*, which is a wire that sends or receives an electrical signal between the Arduino and a component. Signal wires can be any color that is not the same as the power or ground color.

After you assemble your circuit, you need the appropriate software to use it. From the Arduino menu, choose File ⇨ Examples ⇨ 01.Basics ⇨ Fade to call up the Fade sketch. The complete code for the Fade sketch follows:

```
/*
  Fade

  This example shows how to fade an LED on pin 9 using the analogWrite()
  function.

  The analogWrite() function uses PWM, so if you want to change the pin
  you're using, be sure to use another PWM capable pin. On most Arduino,
  the PWM pins are identified with a "~" sign, like ~3, ~5, ~6, ~9, ~10 and ~11.

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/basics/Fade/

  */

int led = 9; // the PWM pin the LED is attached to
int brightness = 0; // how bright the LED is
int fadeAmount = 5; // how many points to fade the LED by

// the setup routine runs once when you press reset:
void setup() {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  // set the brightness of pin 9:
  analogWrite(led, brightness);
```

```

// change the brightness for next time through the loop:
brightness = brightness + fadeAmount;

// reverse the direction of the fading at the ends of the fade:
if (brightness <= 0 || brightness >= 255) {
  fadeAmount = -fadeAmount;
}
// wait for 30 milliseconds to see the dimming effect
delay(30);
}

```

Upload this sketch to your board following the instructions at the start of the chapter. If everything has uploaded successfully, the LED fades from off to full brightness and then back off again.

If you don't see any fading, double-check your wiring:

- » Make sure that you're using the correct pin number.
- » Check that your LED is correctly situated, with the long leg connected by a wire to pin 9 and the short leg connected via the resistor and a wire to GND (ground).
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the Fade sketch

By the light of your fading LED, take a look at how this sketch works.

The comments at the top of the sketch reveal exactly what's happening in this sketch: Using pin 9, a new function called `analogWrite()` causes the LED to fade off and on. After the comments, three declarations appear:

```

int led = 9; // the PWM pin the LED is attached to
int brightness = 0; // how bright the LED is
int fadeAmount = 5; // how many points to fade the LED by

```

“But what is a declaration?” I hear you ask. Read on to find out.

# Declarations

*Declarations* (which aren't something you put up at Christmas, ho ho ho) are values stored for later use by the program. In this case, three variables are declared, but you could declare many other variables or even include libraries of code in your sketch. For now, just remember that variables are declared before the `setup` function.

# Variables

*Variables* are values that can change depending on what the program does with them. In C, you can declare the type, name, and value of the variable before the main body of code, much as ingredients are listed at the start of a recipe:

```
int led = 9;
```

The first part sets the type of the variable, creating an integer (`int`). An integer is a whole number, positive or negative, so no decimal places are required. Note that Arduino has lower and upper limits for the `int` variable:  $-32,768$  to  $32,767$ . Beyond those limits, a different type of variable must be used, known as a `long`. (You learn more about `long` variables in Chapter 10.) For now, an `int` will do just fine. The name of the variable is `led` and is purely for reference; it can be any single word that's useful for figuring out what the variable applies to. Finally, the value of the variable is set to `9`, which is the number of the pin that is being used.

Variables are especially useful when you refer to a value repeatedly. In this case, the variable is called `led` because it refers to the pin to which the physical LED is attached. Now, every time you want to refer to pin `9`, you can write `led` instead. Although this approach may seem like extra work initially, if you decided to change the pin to, say, pin `11`, you would need to change only the variable at the start; every subsequent mention of `led` would automatically pick up the new value. That's a big timesaver over having to trawl through the code to update every occurrence of `9`.

The Fade sketch has three variables: `led`, `brightness`, and `fadeAmount`. These are integer variables and are capable of the same range of values, but each one is used for a different part of the process of fading an LED.



TIP

With declarations made, the code enters the `setup` function. The comments are reminders that `setup` runs only once and that just one pin is set as an output. Here you can see the first variable at work. Instead of writing `pinMode(9, OUTPUT)`, you have `pinMode(led, OUTPUT)`. Both work the same, but the latter uses the `led` variable:

```
// the setup routine runs once when you press reset:
void setup() {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}
```

Then the loop starts to get a bit more complicated:

```
// the loop routine runs over and over again forever:
void loop() {
  // set the brightness of pin 9:
  analogWrite(led, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness &lt;= 0 || brightness &gt;= 255) {
    fadeAmount = -fadeAmount;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

Instead of just on and off values, a fade needs a range of values. `analogWrite` allows you to send a value of 0 to 255 to a PWM pin on the Arduino. 0 is equal to 0V and 255 is equal to 5V, and any value in between gives a proportional voltage, thus fading the LED.

The loop begins by writing the brightness value to pin 9. A brightness value of 0 means that the LED is currently off:

```
// set the brightness of pin 9:
analogWrite(led, brightness);
```

Next you add the fade amount to the brightness variable, making it equal to 5. This value won't be written to pin 9 until the next loop:

```
// change the brightness for next time through the loop:
brightness = brightness + fadeAmount;
```

The brightness must stay within the range that the LED can understand. This is accomplished by using an `if` statement that tests the value of the brightness variables to determine what to do next.

The word `if` starts the statement. The conditions are in the parentheses that follow, so in this case you have two. Is `brightness` less than or equal to 0 (indicated by the `<=` symbol) is the first condition. Is `brightness` greater than or equal to 255 (indicated by the `>=` symbol) is the second condition. In between the two conditional statements is the `||` symbol, which is the symbol for OR:

```
// reverse the direction of the fading at the ends of the fade:
if (brightness <= 0 || brightness >= 255) {
  fadeAmount = -fadeAmount;
}
```

So the complete statement is, “If the variable named `brightness` is less than or equal to 0, or greater than or equal to 255, do whatever is inside the curly brackets.” When this eventually becomes true, the line of code inside the curly brackets is read. This basic mathematical statement inverts the `fadeAmount` variable. During the fade up to full brightness, 5 is added to the `brightness` with every loop. When 255 is reached, the `if` statement becomes true and `fadeAmount` changes from 5 to -5. Then every loop updates to “add minus 5” to the `brightness` until 0 is reached, when the `if` statement becomes true again. This inverts the `fadeAmount` of -5 to 5 to bring everything back to where it started:

```
fadeAmount = -fadeAmount;
```

These conditions give us a number continually counting up and then down that the Arduino can use to continually fade your LED on and then off again.

## Tweaking the Fade sketch

You can accomplish a task on your Arduino board in many ways. In this section, I will show you one different way to fade an LED using the circuit that you created in the preceding section. The following code is the Fading example. From the Arduino menu, choose File ⇨ Examples ⇨ 03.Analog ⇨ Fading to open the Fading sketch. Upload it and you will see that no visible difference exists between this and the preceding example.



REMEMBER

Some areas of the code appear colored on your screen, most often turquoise, green, or orange. This coloring marks a function or a statement recognized by the Arduino environment (and can be extremely handy for spotting typos). Color is impossible to recreate in a black-and-white book, so any colored code appears here in **bold**.

```

/*
  Fading

  This example shows how to fade an LED using the analogWrite() function.

  The circuit:
  - LED attached from digital pin 9 to ground.

  created 1 Nov 2008
  by David A. Mellis
  modified 30 Aug 2011
  by Tom Igoe

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/analog/Fading/
*/

int ledPin = 9; // LED connected to digital pin 9

void setup() {
  // nothing happens in setup
}

void loop() {
  // fade in from min to max in increments of 5 points:
  for(int fadeValue = 0; fadeValue <= 255; fadeValue += 5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }

  // fade out from max to min in increments of 5 points:
  for(int fadeValue = 255 ; fadeValue >= 0; fadeValue -= 5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
}

```

The Fade example is efficient and does a simple fade very well, but it relies on the `loop` function to update the LED value. This version uses `for` loops, which operate within the main Arduino `loop` function.

## USING FOR LOOPS

When a sketch enters a `for` loop, it sets up the criteria for exiting the loop and can't move out of it until the criteria are met. `for` loops are often used for repetitive operations; in this case, `for` loops are used to increase or decrease a number at a set rate to create the repeating fade.

The first line of the `for` loop defines the initialization, the test, and the amount of increment or decrement:

```
for(int fadeValue = 0; fadeValue <= 255; fadeValue += 5)
```

In plain English, this reads: "Make a variable called `fadeValue` (that is local to this `for` loop) equal to a value of 0; check to see whether it is less than or equal to 255; if it is, set `fadeValue` to be equal to `fadeValue` plus 5." `fadeValue` is equal to 0 only when it is created; after that, it is increased by 5 every time the `for` loop cycles.

Within the loop, the code updates the `analogWrite` value of the LED and waits 30 milliseconds (ms) before attempting the loop one more time:

```
for (int fadeValue = 0 ; fadeValue <= 255; fadeValue += 5) {  
  // sets the value (range from 0 to 255):  
  analogWrite(ledPin, fadeValue);  
  // wait for 30 milliseconds to see the dimming effect  
  delay(30);  
}
```

This `for` loop behaves in the same way as the main loop in the default Fade example, but because the `fadeValue` is contained in its own loop, and broken into fade up and fade down loops, it is a lot easier to experiment with fading patterns in a more controlled way. For example, change `+= 5` and `-= 5` to different values (that divide into 255 neatly) for some interesting asymmetrical fading.

You could also copy and paste the same `for` loops to create further fading animations. However, bear in mind that your Arduino can do nothing else while it is in a `for` loop.

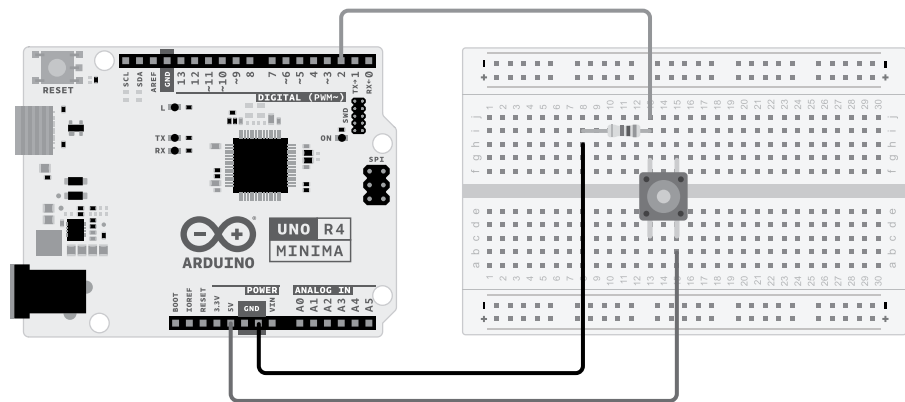
# The Button Sketch

The first and perhaps most basic of inputs that you can and should learn for your Arduino projects is the modest pushbutton.

For the Button project, you need the following:

- » An Arduino Uno
- » A breadboard
- » A 10k ohm resistor
- » A pushbutton
- » An LED
- » Jump wires

Figure 6-3 shows the breadboard layout for the Button circuit. It's important to note which legs of the pushbutton are connected. In most cases, these small pushbuttons are made to bridge the gap over the center of your breadboard exactly. If they do bridge the gap, the legs are usually split at 90 degrees to the gap (left to right on this diagram).

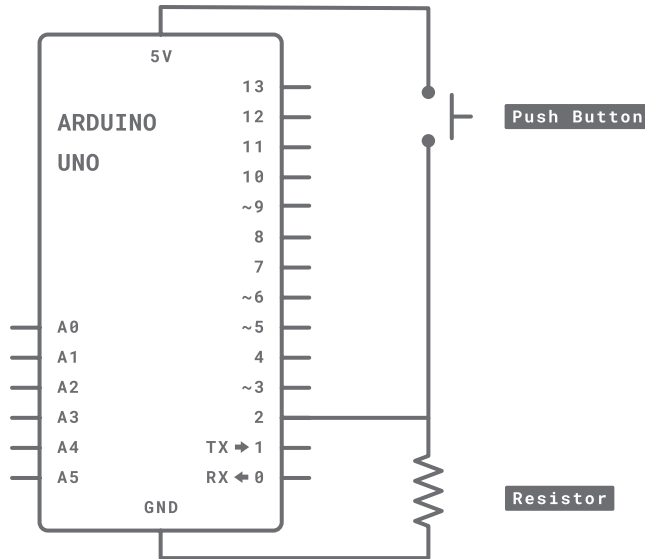


**FIGURE 6-3:**  
Pin 2 is reading  
the pushbutton.

You can test the legs of a pushbutton with a continuity tester if your multimeter has that function (as detailed in Chapter 4).

From the schematic in Figure 6-4, you can see that the resistor leading to ground should be connected to the same side of the pushbutton as pin 2, and that when

the pushbutton is pressed, it connects the 5V pin to both D2 and GND. This setup is used to compare ground (0V) to a voltage (5V) so that you can tell whether the switch is open or closed.



**FIGURE 6-4:**  
A schematic of  
the pushbutton  
circuit.

Build the circuit and upload the code from File ⇨ Examples ⇨ 02.Digital ⇨ Button.

```
/*  
Button  
  
Turns on and off a light emitting diode(LED) connected to digital pin 13,  
when pressing a pushbutton attached to pin 2.  
  
The circuit:  
- LED attached from pin 13 to ground  
- pushbutton attached to pin 2 from +5V  
- 10K resistor attached to pin 2 from ground  
- Note: on most Arduinos there is already an LED on the board  
  attached to pin 13.  
  
created 2005  
by DojoDave <http://www.0j0.org>;  
modified 30 Aug 2011  
by Tom Igoe
```

This example code is in the public domain.

```
https://docs.arduino.cc/built-in-examples/digital/Button/

*/

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2; // the number of the pushbutton pin
const int ledPin = 13; // the number of the LED pin

// variables will change:
int buttonState = 0; // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

After you upload the sketch, give your button a press and you should see the pin 13 LED light. (To make the light easier to see, add a bigger LED to your Arduino board between pin 13 and GND.)

If you don't see anything lighting up, double-check your wiring:

- » Make sure that your button is connected to the correct pin number.
- » If you're using an additional LED, check that it is correctly situated, with the long leg in pin 13 and the short leg in GND. You can also remove it and monitor the LED mounted on the board (marked L) instead.
- » Check the connections on the breadboard. Make sure that the jump wires and components are connected using the correct rows in the breadboard.

## Understanding the Button sketch

This sketch is your first interactive Arduino project. The previous sketches were all about outputs, but now you can affect those outputs by providing real-world human input!

While pressed, your button turns on a light. When the button is released, the light turns off. Let's take a look at the sketch from the top to see how this happens.

The first step in the Button sketch is to declare constants and variables. A *constant*, declared using the keyword `const`, is a value or an identifier whose value doesn't change for the duration of the program. In the next example the values of `buttonPin` and `ledPin` are both defined as constants. This approach is best used for values that aren't supposed to change; this way, you are making doubly sure that they won't. Pin numbers are being assigned because you won't change the pin number physically.

The variable `buttonState` is set to 0. This variable monitors changes to the button:

```
const int buttonPin = 2; // the number of the pushbutton pin
const int ledPin = 13; // the number of the LED pin

// variables will change:
int buttonState = 0; // variable for reading the pushbutton status
```

`setup` establishes `pinMode`, with `ledPin` (pin 13) as the output and `buttonPin`, (pin 2) as the input:

```
void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
```

```
// initialize the pushbutton pin as an input:
pinMode(buttonPin, INPUT);
}
```

In the main loop, you can see the order of things clearly. First, the `digitalRead` function is used on pin 2. Just as `digitalWrite` can write a HIGH or LOW (1 or 0) value to a pin, `digitalRead` can read the current value, either HIGH (1) or LOW (0), from a pin. That value is then stored in the `buttonState` variable:

```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);
}
```

With the button state established, a test uses an `if` statement to determine what happens next. The statement reads: “If there is a HIGH value (voltage connected to the circuit), then send a HIGH value to `ledPin` (pin 13) to turn the LED on; if there is a LOW value (the pin is grounded), then send a LOW value to `ledPin` to turn the LED off; repeat.”

```
// check if the pushbutton is pressed.
// if it is, the buttonState is HIGH:
if (buttonState == HIGH) {
  // turn LED on:
  digitalWrite(ledPin, HIGH);
}
else {
  // turn LED off:
  digitalWrite(ledPin, LOW);
}
}
```

## Tweaking the Button sketch

It’s often necessary to invert the output of a switch or sensor, and you can do this in two ways. The easiest is to change one word in the code.

By changing the line of code in the preceding sketch from

```
if (buttonState == HIGH)
```

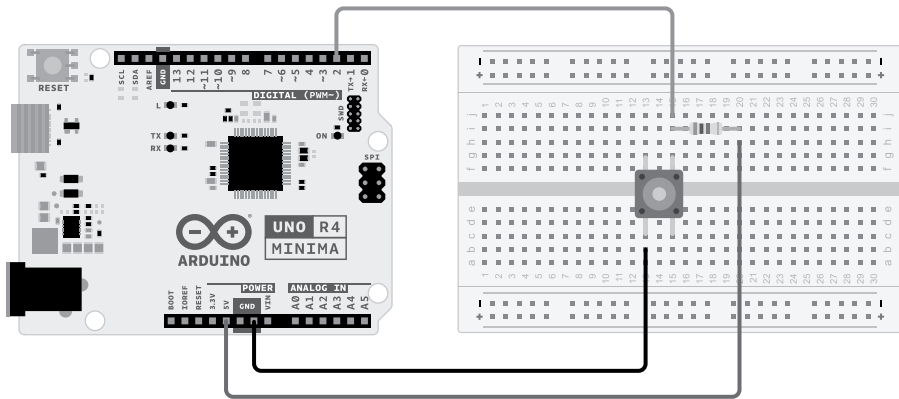
to

```
if (buttonState == LOW)
```

the output is reversed. This means that the LED is on until the button is pressed. If you have a computer, this option is the easiest. Simply upload the code.

However, sometimes (such as when your laptop battery is dead) you don't have the means to upload the edited code. Often, the easiest way to flip the logic is to flip the polarity of the circuit.

Instead of connecting pin 2 to a resistor and then GND, connect that resistor to 5V and move the GND wire to the other side of the button, as shown in Figure 6-5.



**FIGURE 6-5:**  
A button with the  
polarity flipped.

## The AnalogInput Sketch

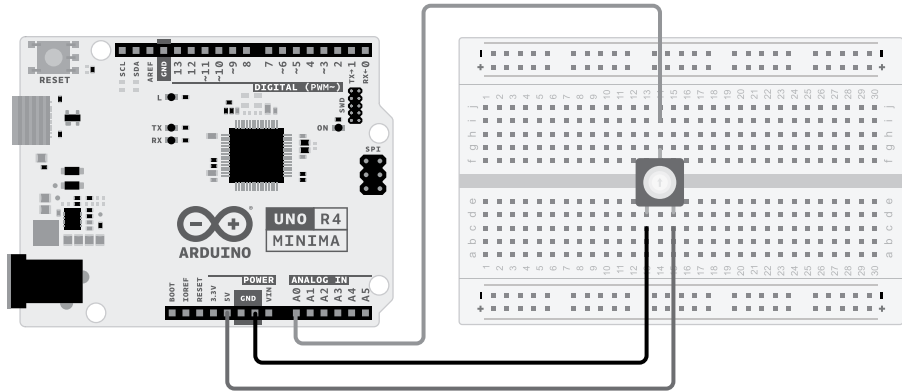
The preceding sketch showed you how to use `digitalRead` to read either on or off, but what if you want to handle an analog value such as a dimmer switch or a volume control knob?

For the AnalogInput project, you need

- » An Arduino Uno
- » A breadboard
- » A 10k ohm variable resistor
- » An LED
- » Jump wires

In Figure 6-6 you see the layout for this circuit. You need an LED and a resistor for your output, and a variable resistor for your input. If you're not quite sure what a variable resistor is, check out the “Variable Resistors” sidebar.

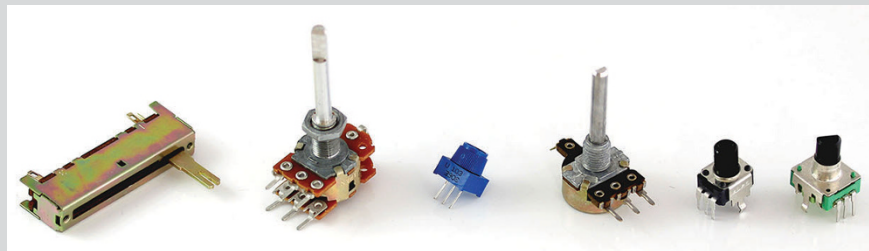
**FIGURE 6-6:**  
The potentiometer (variable resistor) is connected to Analog Pin A0.



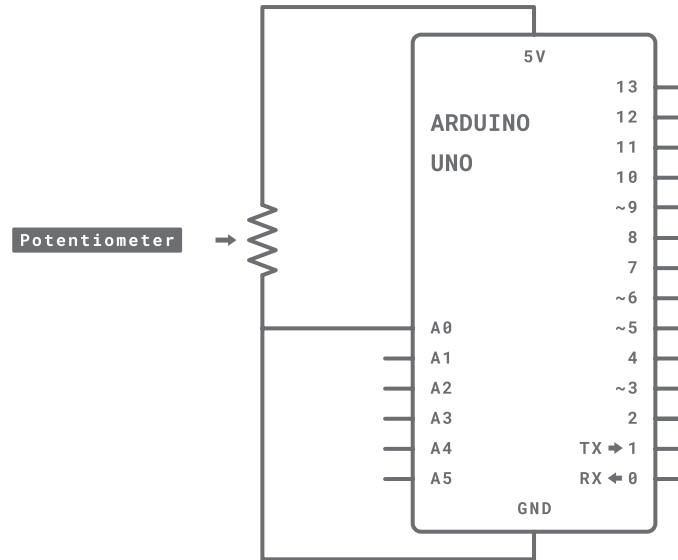
## VARIABLE RESISTORS

*Variable resistors* (also known as *potentiometers* or *pots*), like standard passive resistors, resist the flow of current in a circuit. The difference is that rather than having a fixed value, they have a range. Normally, the upper limit of this range is printed on the resistor. For example, a variable resistor with a value of 10K  $\Omega$  gives you a range of 0 ohms to 10,000 ohms. This change is something that can be monitored electrically to give a variable analog input.

Variable resistors come in a variety of shapes and sizes, as shown in the following figure. Anything with this analog movement, such as a thermostat, the dial on your washing machine, or the dial on your toaster to set the time, most likely contains a potentiometer.



In Figures 6-6 and 6-7, the variable resistor has power and ground connected across opposite pins, with the central pin providing the reading. To read the analog input, you need to use the special set of analog input pins on the Arduino board.



**FIGURE 6-7:**  
A schematic  
of the  
AnalogInput  
circuit.



TIP

Note that if you were to swap the resistor's polarity (the positive and negative wires), you would invert the direction of the potentiometer. This can be a quick fix if you find that your resistor's value is going in the wrong direction.

Build the circuit in Figure 6-6, and then upload the code from [File](#) ⇨ [Examples](#) ⇨ [03.Analog](#) ⇨ [AnalogInput](#) to start controlling your variable resistor.

```
/*  
Analog Input  
  
Demonstrates analog input by reading an analog sensor on analog pin 0 and  
turning on and off a light emitting diode(LED) connected to digital pin 13.  
The amount of time the LED will be on and off depends on the value obtained  
by analogRead().  
  
The circuit:  
- potentiometer  
  center pin of the potentiometer to the analog input 0
```

```
one side pin (either one) to ground
the other side pin to +5V
- LED
  anode (long leg) attached to digital output 13
  cathode (short leg) attached to ground

- Note: because most Arduinos have a built-in LED attached to pin 13 on the
  board, the LED is optional.
```

```
created by David Cuartielles
modified 30 Aug 2011
```

By Tom Igoe

This example code is in the public domain.

```
  https://docs.arduino.cc/built-in-examples/analog/AnalogInput/
*/

int sensorPin = A0; // select the input pin for the potentiometer
int ledPin = 13;   // select the pin for the LED
int sensorValue = 0; // variable to store the value coming from the sensor

void setup() {
  // declare the ledPin as an OUTPUT:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // read the value from the sensor:
  sensorValue = analogRead(sensorPin);
  // turn the ledPin on
  digitalWrite(ledPin, HIGH);
  // stop the program for <sensorValue> milliseconds:
  delay(sensorValue);
  // turn the ledPin off:
  digitalWrite(ledPin, LOW);
  // stop the program for <sensorValue> milliseconds:
  delay(sensorValue);
}
```

After the sketch is uploaded, turn the potentiometer. The result is an LED that blinks slower or faster depending on the value of the potentiometer. You can add another LED between pin 13 and GND to improve the effect of this spectacle.

If you don't see anything lighting up, double-check your wiring:

- » Make sure that you're using the correct pin number for your variable resistor.
- » Check that your LED is the correct way around, with the long leg in pin 13 and the short leg in GND.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the AnalogInput sketch

Analog sensors come in a variety of forms, but the principle is generally the same for each of them. In this section you examine the AnalogInput sketch to get a better understanding of how Arduino interprets these sensors.

The declarations state the pins that this sketch uses. The pin for the analog reading is written as `A0`, which is short for analog input pin 0 to mark it as the first analog input pin in the row of 6 (numbered 0 to 5). Both `ledPin` and `sensorValue` are declared as standard variables. It's worth noting that `ledPin` and `sensorPin` could both be declared as constant integers (`const`) because they don't change. Because the `sensorValue` value will change, it is stored as a variable:

```
int sensorPin = A0; // select the input pin for the potentiometer
int ledPin = 13; // select the pin for the LED
int sensorValue = 0; // variable to store the value coming from the sensor
```

During setup, you need only declare the `pinMode` of the digital `ledPin`. The analog input pins, as their name implies, are for input only.



TIP

You can use the analog input pins also as more basic digital input or output pins. Instead of referring to them as analog pins `A0–A5`, you could number them as digital pins `14–19`, as an extension of the existing digital pins. Each must then be declared as either an input or an output using the `pinMode` function, as with any digital pin:

```
void setup() {
  // declare the ledPin as an OUTPUT:
  pinMode(ledPin, OUTPUT);
}
```

Similarly to the Button sketch, the AnalogInput sketch reads the sensor first. When using the `analogRead` function, it interprets the voltage value of an analog pin. As the resistance changes, so too does the voltage. The accuracy of the value depends on the quality of the variable resistor. The Arduino uses the analog-to-digital converter on the ATmega328 chip to read this analog voltage. Instead of 0V, 0.1V, 0.2V, and so on, the Arduino returns a value as an integer in the range of 0–1023. For example, a voltage of 2.5V is interpreted as 511.

It's generally a good idea to read the sensor data first to prevent any delays when reading values, even though the looping occurs extremely quickly. Otherwise, this can give the effect of a lag in the response of the sensor.

After `sensorValue` is read, the sketch is essentially the same as the Blink sketch, but with a variable delay. The `ledPin` is written HIGH, waits, is written LOW, waits for the same amount of time, updates the sensor value, and repeats.

Using the raw sensor value (0–1023) makes the delay between 0 seconds and 1.023 seconds:

```
void loop() {
  // read the value from the sensor:
  sensorValue = analogRead(sensorPin);
  // turn the ledPin on
  digitalWrite(ledPin, HIGH);
  // stop the program for <sensorValue> milliseconds:
  delay(sensorValue);
  // turn the ledPin off:
  digitalWrite(ledPin, LOW);
  // stop the program for <sensorValue> milliseconds:
  delay(sensorValue);
}
```

This sketch blinks your LED at various rates. However, as the blinks become slower, the delays in the loop become longer and, therefore, the readings from the sensor become less frequent. This can make your sensor less responsive when it's at the higher values, giving you less consistent readings. For another look at sensors, as well as how to smooth and calibrate them, head over to Chapter 10.

## Tweaking the AnalogInput sketch

The `analogRead` function has supplied an integer value, and you can apply all sorts of conditions or calculations to that number in your sketch. In this example,

I show you how to measure whether a sensor value is above a certain number or threshold.

By putting an `if` statement around the `digitalWrite` part of the loop, you are able to set a threshold. In this example, the LED blinks only if it is over the sensor's halfway value of 511:

```
void loop() {
  // read the value from the sensor:
  sensorValue = analogRead(sensorPin);
  if (sensorValue > 511){
    // turn the ledPin on
    digitalWrite(ledPin, HIGH);
    // stop the program for <sensorValue> milliseconds:
    delay(sensorValue);
    // turn the ledPin off:
    digitalWrite(ledPin, LOW);
    // stop the program for <sensorValue> milliseconds:
    delay(sensorValue);
  }
}
```

Try adding some conditions, but be aware that the sensor won't update as frequently if there are too many delays. For other sketches that remedy this problem, check out the `BlinkWithoutDelay` sketch in Chapter 10.

## Talking Serial

It's good to see the effects of your circuit through an LED, but unless you can see the values, you might find it difficult to know whether a circuit is behaving as expected. The project in this section and the one following are designed to display the value of inputs by using the *serial monitor*.

*Serial* is a method of communication between a peripheral and a computer. In this case, it is serial communication over Universal Serial Bus (USB). Data is sent one byte at a time in the order that it is written. When reading sensors with an Arduino, the values are sent over this connection and can be monitored or interpreted on your computer.

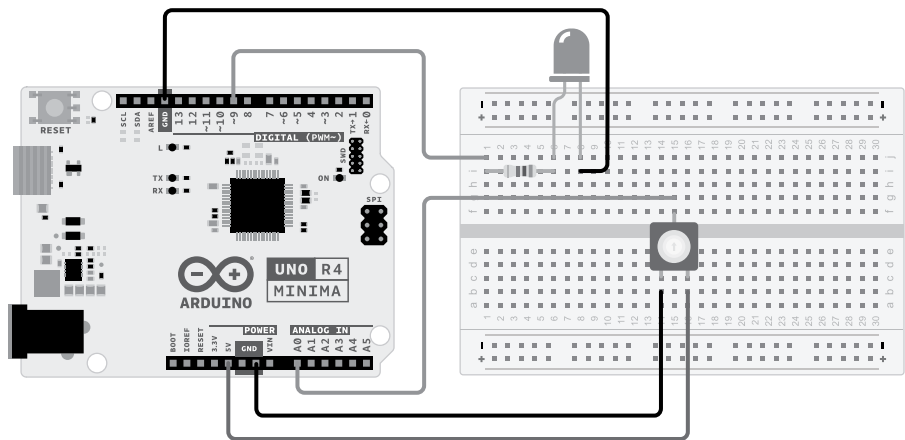
# The AnalogInOutSerial sketch

In the AnalogInOutSerial project, you monitor an analog value sent by a variable resistor over the serial monitor. These variable resistors are the same as the volume control knobs on your stereo, but people often have no idea how they work. In this example, you monitor the value as detected by your Arduino and display it on your screen in the serial monitor, giving you a greater understanding of the range of values and the performance of this analog sensor.

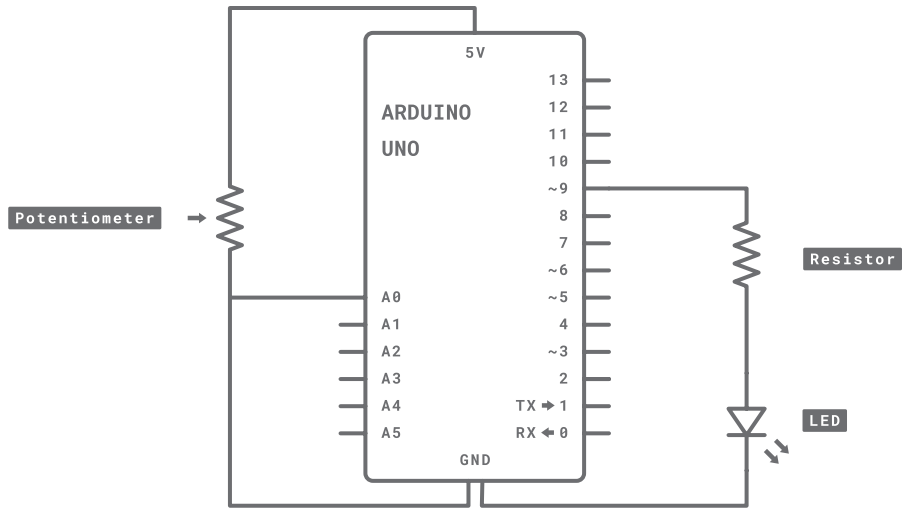
You need the following:

- » An Arduino Uno
- » A breadboard
- » A 10k ohm variable resistor
- » A resistor (greater than 120 ohm)
- » An LED
- » Jump wires

The circuit, as shown in Figures 6-8 and 6-9, is similar to the example for the AnalogInput circuit, but with the addition of an LED connected to pin 9 as in the Fade circuit. The code fades the LED on and off according to the turn of the potentiometer. Because the input and the output have a different range of values, the sketch includes a conversion to use the potentiometer to fade the LED. This circuit is a great example of using the serial monitor for debugging and displays both the input and output values for maximum clarity.



**FIGURE 6-8:**  
Dimmer circuit  
to be read  
over serial.



**FIGURE 6-9:**  
A schematic of  
the dimmer  
circuit.

Complete the circuit and upload the code from File ⇨ Examples ⇨ 03.Analog ⇨ AnalogInOutSerial.

```

/*
  Analog input, analog output, serial output

  Reads an analog input pin, maps the result to a range from 0 to 255 and uses
  the result to set the pulse width modulation (PWM) of an output pin.
  Also prints the results to the Serial Monitor.

  The circuit:
  - potentiometer connected to analog pin 0.
    Center pin of the potentiometer goes to the analog pin.
    side pins of the potentiometer go to +5V and ground
  - LED connected from digital pin 9 to ground

  created 29 Dec. 2008
  modified 9 Apr 2012
  by Tom Igoe

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/analog/AnalogInOutSerial/

  */

```

```

// These constants won't change. They're used to give names
// to the pins used:
const int analogInPin = A0; // Analog input pin that the potentiometer is
    attached to
const int analogOutPin = 9; // Analog output pin that the LED is attached to
int sensorValue = 0; // value read from the pot
int outputValue = 0; // value output to the PWM (analog out)

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

void loop() {
  // read the analog in value:
  sensorValue = analogRead(analogInPin);
  // map it to the range of the analog out:
  outputValue = map(sensorValue, 0, 1023, 0, 255);
  // change the analog out value:
  analogWrite(analogOutPin, outputValue);

  // print the results to the serial monitor:
  Serial.print("sensor = ");
  Serial.print(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);

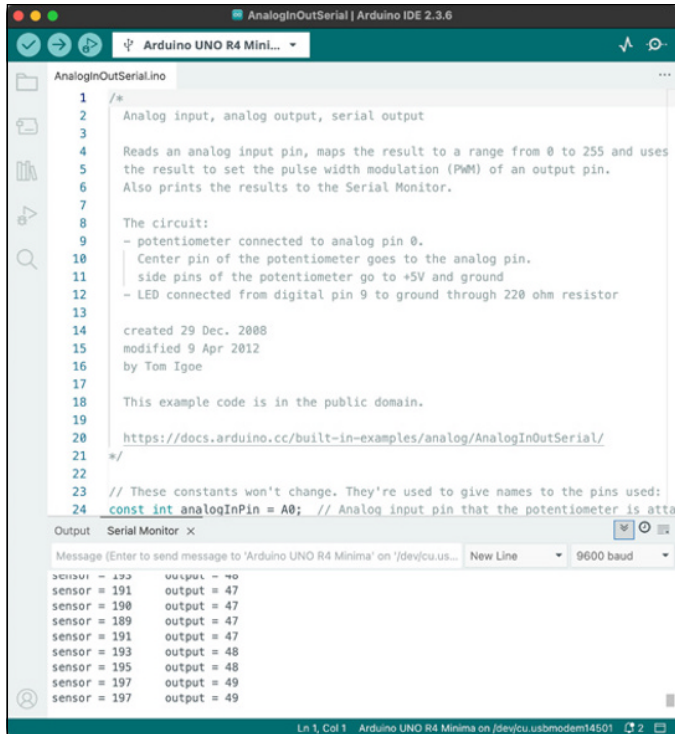
  // wait 2 milliseconds before the next loop
  // for the analog-to-digital converter to settle
  // after the last reading:
  delay(2);
}

```

After you upload the sketch, turn the potentiometer with your fingers. The result should be an LED that fades on and off depending on the value of the potentiometer. Now click the serial monitor button on the top right of the Arduino window to monitor the numerical values you are receiving and sending to the LED. (See Figure 6-10.)

If you don't see anything happening, double-check your wiring:

- » Make sure that you're using the correct pin number for your variable resistor.
- » Check that your LED is the correct way round, with the long leg connected to Pin 9 and the short leg connected to GND, via a resistor.



**FIGURE 6-10:** Sensor data is displayed in the Serial Monitor.

- » Check the connections on the breadboard. The jump wires and components will not work if you are not using the correct rows in the breadboard.
- » If you receive strange characters instead of words and numbers, check the baud rate in the serial monitor. If it is not set to 9600, use the drop-down menu to select that rate.

## Understanding the AnalogInOutSerial sketch

The start of the sketch is straightforward. It declares constants for both of the pins in use for the analog input and the PWM output. There are also two variables for the raw data from the sensor (`sensorValue`) and the value that is sent to the LED (`outputValue`):

```
const int analogInPin = A0; // Analog input pin that the potentiometer is
    attached to
const int analogOutPin = 9; // Analog output pin that the LED is attached to
```

```
int sensorValue = 0; // value read from the pot
int outputValue = 0; // value output to the PWM (analog out)
```

In the `setup` function, you have little to do beyond opening the serial communication line:

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

The real action is in the loop. As in the `Fade` sketch, the best place to start is with reading the input. The `sensorValue` variable stores the reading from `analogInPin`, which will be in the range of 0–1024:

```
void loop() {
  // read the analog in value:
  sensorValue = analogRead(analogInPin);
```

Because fading an LED using PWM requires a range of 0–255, you need to scale the `sensorValue` down to make it fit the lower range of `outputValue`. To do so, you use the `map` function. The `map` function scales a variable. By setting the variable’s minimum and maximum to a new minimum and maximum, all the scaling is handled for you. The `map` function creates an `outputValue` directly proportional to the `sensorValue` but on a smaller scale:

```
// map it to the range of the analog out:
outputValue = map(sensorValue, 0, 1023, 0, 255);
```



TIP

Functions like this are useful but can sometimes be overkill. In this example, you can achieve the same result by simply dividing the sensor value by 4:

```
outputValue = sensorValue/4;
```

Because `outputValue` is an integer, it is rounded to the nearest whole number. The `outputValue` is then written to the LED using the `analogWrite` function:

```
// change the analog out value:
analogWrite(analogOutPin, outputValue);
```

This code is enough for the circuit to function, but if you want to know what’s going on, you need to write some values to the serial port. The code has three lines of `Serial.print` before `Serial.println`. Each occurrence of `Serial.print` writes text on the same line of the serial monitor. Using `Serial.println`

finishes that text with a carriage return, starting a new line. Therefore, a new line of text is written every time the program completes a loop.

Text inside the quotation marks is for labeling or adding characters. You can also use special characters, such as `\t`, which adds a tab for spacing:

```
// print the results to the serial monitor:  
Serial.print("sensor = ");  
Serial.print(sensorValue);  
Serial.print("\t output = ");  
Serial.println(outputValue);
```

An example of this line in the serial monitor follows:

```
sensor = 1023 output = 511
```

The loop is finished with a short delay to stabilize the results and then the loop is repeated, updating the input, output, and readings on the serial monitor.

```
// wait 2 milliseconds before the next loop  
// for the analog-to-digital converter to settle  
// after the last reading:  
delay(2);  
}
```



TIP

This delay time is largely arbitrary. The 2 ms could just as well be 1 ms, as in the previous example. You may have to experiment with these small delays. If a sensor is jumpy, you may want to go up to 10 ms, or you may find that the reading is perfectly smooth and can be removed completely. There is no magic value.

#### IN THIS CHAPTER

- » Moving DC motors
- » Switching bigger loads with transistors
- » Speeding up your motor
- » Turning with precision using a stepper motor
- » Making electronic music with a buzzer

## Chapter 7

# More Basic Sketches: Motion and Sound

Chapter 6 shows you how to use some simple LEDs as outputs for various circuits. In Arduino land, nothing is more beautiful than a blinking LED, but you have a variety of other outputs as options. In this chapter, I explore two other areas: motion provided by motors and sound from a buzzer.

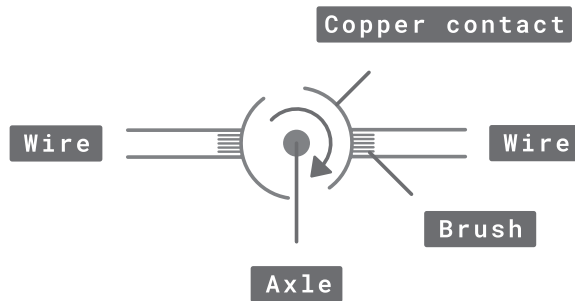
## Working with Electric Motors

Electric motors allow you to move things with electricity using the power of *electromagnetism*. When an electrical current is passed through a coil of wire, it creates an electromagnetic field. This process works similarly to a normal permanent bar magnet but gives you control over the presence of the field, meaning that you can turn it on and off at will and even change the direction of the magnetism. As you may remember from school, magnets have two possible states: attraction or repulsion. In an electromagnetic field, you can switch between these by changing the polarity, which in practical terms means switching the positive and negative wires.

Electromagnets have a variety of uses, such as in electrically operated locks, automated plumbing valves, and read-write heads on hard disks. They're used also for lifting scrap metal. Even the CERN Large Hadron Collider uses an electromagnet. In this chapter, I focus on another important use: electric motors.

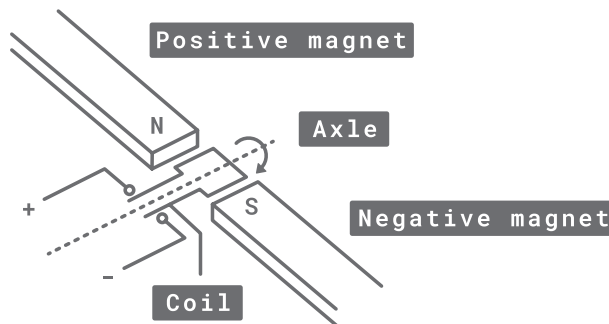
An electric motor is made up of a coil of wire (electromagnet) between two regular, permanent magnets. By alternating the polarity of the coil, it is possible to rotate it because it is pulled by one magnet and then pushed toward the next. If this is done fast enough, the coil gathers momentum to spin.

The first part to understand is how the coil can spin if it is attached to wires. This spin is achieved by mounting two copper brushes on the axle. The brushes stay in contact with two semicircles of copper, as shown in Figure 7-1, so a connection can be maintained without any fixed wires. The semicircles also mean that the two points are never in contact, which would cause a short circuit.



**FIGURE 7-1:** How a motor's axle can be connected but still be free to spin.

With a freely spinning coil in place on an axle, you can affect the coil by placing two permanent bar magnets near it. As shown in Figure 7-2, the magnets are placed on either side of the coil, with different poles on each side. If you put electrical current across the coil, you give it a polarity — either north or south, as with conventional bar magnets. If the coil is north, it is repelled by the north bar magnet and attracted by the south bar magnet.



**FIGURE 7-2:** A diagram of an electric motor.

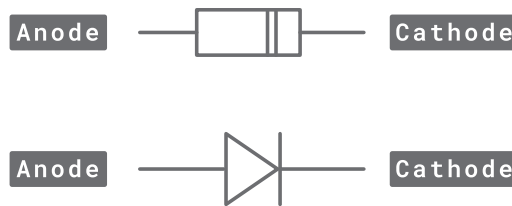
If you look at the brush again, you realize that something else happens when the coil does a half rotation: The polarity flips. When this happens, the cycle starts again and the north coil becomes south and is pushed away by the south magnet back to north. Because of the momentum produced when the coil is repelled, this movement continues in the same direction while sufficient power exists.

This type of electric motor is the most basic; modern ones are highly refined, with more coils and magnets to produce a smoother movement. Other motors are also based on this principle but have more advanced controls to move, for example, by a precise number of degrees or to a specific location. In your kit, you should have two varieties of electric motor: a DC motor and a servomotor.

## Discovering Diodes

An essential component for motor control circuits is the diode. As explained earlier in this chapter, you can spin an electric motor by putting voltage through it. But if a motor is spinning or is turned without having a voltage put through it, it generates a voltage in the opposite direction; this is how electric generators and dynamos produce electricity from movement.

If this reversal of voltage happens in your circuit, the effects can be disastrous, including damaged or destroyed components. So to control this reverse current, you use a diode. *Diodes* block current in one direction and allow it in the other. Current can flow from the anode to the cathode. Figure 7-3 shows how the anode and cathode are marked for both the physical diode and the circuit diagram, with a band on the physical diode and a solid line on the schematic, both indicating the cathode.



**FIGURE 7-3:**  
A physical diode  
and its sche-  
matic symbol.

## Spinning a DC Motor

The DC motor in your kit (also known as a hobby motor or a brushed DC motor) is the most basic of electric motors and is used in all types of hobby electronics such as model planes and trains. When current is passed through a DC motor, it spins

continuously in one direction until the current stops. Unless specifically marked with a + or -, DC motors have no polarity, which means that you can swap the two wires to reverse the direction of the motor. Many other, bigger motors exist, but in this example I stick to the small hobby motors.

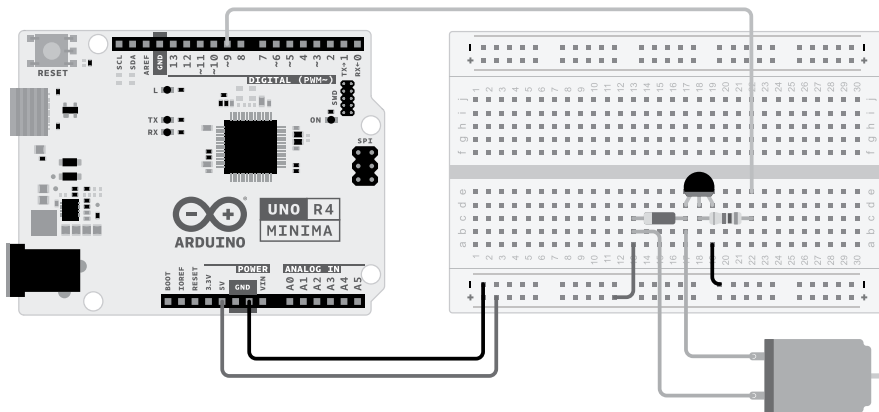
## The Motor sketch

In this section, I show you how to set up a simple control circuit to turn your motor on and off.

You need the following:

- » An Arduino Uno
- » A breadboard
- » A transistor
- » A DC motor
- » A diode
- » A 2.2k ohm resistor
- » Jump wires

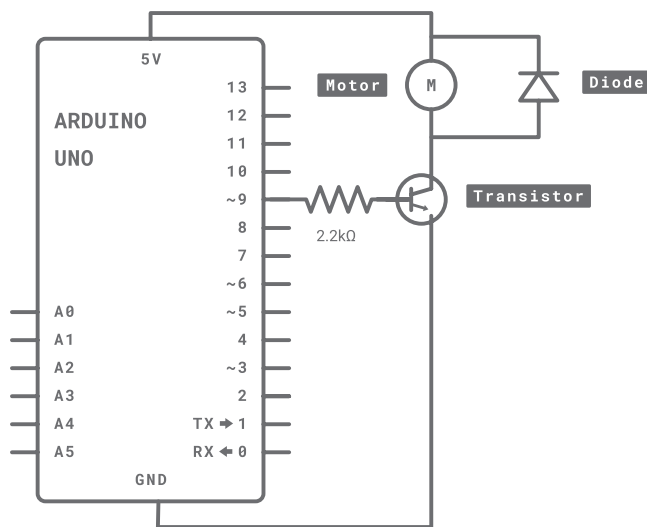
Figure 7-4 shows the layout for this circuit.



**FIGURE 7-4:**  
A transistor  
circuit to drive  
your electric  
motor.

The circuit diagram in Figure 7-5 should clarify exactly what is going on. To power the motor, you need to send 5V through it and then on to ground. This voltage spins the motor, but you have control of it. To give your Arduino control of the

motor's power, and therefore its rotation, you place a transistor just after the motor. The transistor, as described in the sidebar “Understanding Transistors,” is an electrically operated switch that can be activated by your Arduino’s digital pins. In this example, it is controlled by pin 9 on your Arduino, in the same way as an LED except that the transistor allows you the turn the motor circuit on and off.



**FIGURE 7-5:**  
A circuit diagram  
of a transistor  
circuit.

## UNDERSTANDING TRANSISTORS

Sometime it's not possible or advisable to power an output directly from your Arduino pins. By using a transistor, you can control a bigger circuit from your modestly powerful Arduino.

Motors and other high-power outputs (such as large LED arrays) need more voltage or current than an Arduino pin can safely supply, so they must be powered from a separate circuit. A transistor acts like an electronic switch, allowing the Arduino to turn that higher-power circuit on and off using a very small control signal, similar to how you blink an LED.

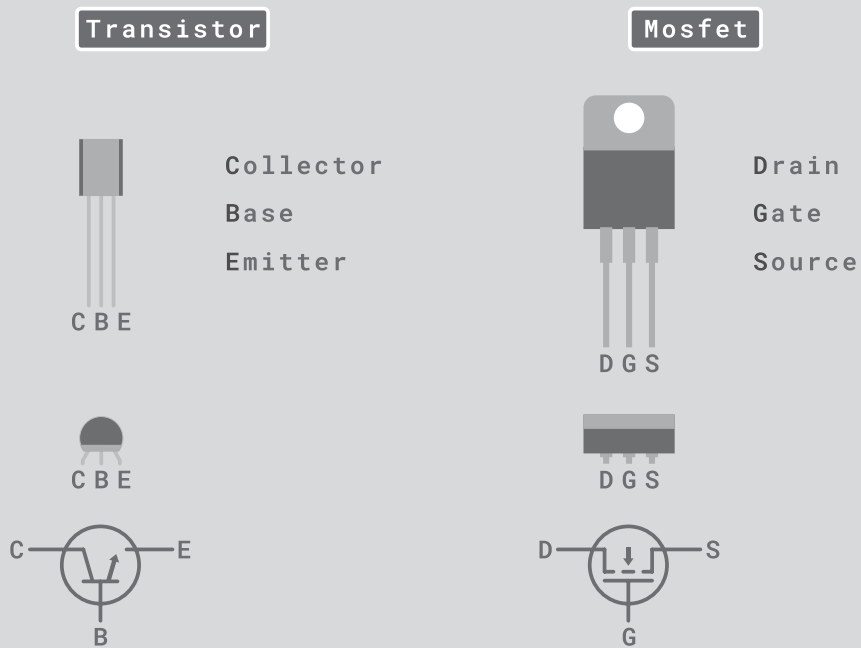
There are many transistors in the world, and each has its own product number that you can Google for details. The one I use in this section's example is a P2N2222A, which is an NPN-type transistor.

*(continued)*

(continued)

There are two main kinds of transistor: NPN and PNP. The difference is simply how they switch. An NPN transistor turns on when a positive voltage is applied to its base, which makes it ideal for use with Arduino's digital output pins. A PNP transistor works the opposite way and turns on when the base is pulled low (toward ground).

A transistor has three legs: base, collector, and emitter. The *base* (or *gate*) is where the Arduino digital signal is sent; the *collector* (or *drain*) is the power source; and the *emitter* (or *source*) is the ground. The legs are numbered and, you hope, named in the datasheet to tell you which leg is which. In a circuit diagram, a transistor is drawn as in the following figure, with the collector at the top, the base to the left, and the emitter at the bottom.



This circuit works, but it still allows the chance of creating a reverse current because of the momentum of the motor as it slows down or because the motor could be turned. If reverse current is generated, it travels from the negative side of the motor and tries to find the easiest route to ground. This route may be through the transistor or through the Arduino. You can't know for sure what will happen, so you need to provide a way to control this excess current.

To be safe, you place a diode across the motor. The diode faces toward the source of the voltage, so the voltage is forced through the motor, which is what you want.

If current is generated in the opposite direction, it will now be blocked from flowing into the Arduino.



WARNING

If you place the diode the wrong way, the current bypasses the motor and you create a short circuit. The short circuit tries to ground all the available current and could break your USB port or, at the very least, display a warning message, informing you that your USB port is drawing too much power.

Build the circuit as shown, and open a new Arduino sketch. Press the Save button and save the sketch with a memorable name, such as myMotor, and then type the following code:

```
int motorPin = 9;

void setup() {

  pinMode(motorPin, OUTPUT);
}

void loop() {

  digitalWrite(motorPin, HIGH);
  delay(1000);

  digitalWrite(motorPin, LOW);
  delay(1000);

}
```

After you've typed the sketch, save it and click the Compile button to check your code. The Arduino IDE (introduced in Chapter 2) checks your code for any syntax errors (code grammar) and highlights them in the message area. The most common mistakes include typos, missing semicolons, and case sensitivity.

If the sketch compiles correctly, click Upload to upload the sketch to your board. You should see your motor spinning for one second and stopping for one second repeatedly.

If that's not what happens, you should double-check your wiring:

- » Make sure that you're using pin number 9.
- » Check that your diode is facing the correct way, with the band facing the 5V connection.

» Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the Motor sketch

The Motor sketch is a basic sketch, and you may have noticed that it's a variation on the Blink sketch. This example changes the hardware but uses the same code to control an LED.

First, the pin is declared using digital pin 9:

```
int motorPin = 9;
```

In the setup function, pin 9 is defined as an output:

```
void setup() {  
  
    pinMode(motorPin, OUTPUT);  
  
}
```

The loop tells the output signal to go to HIGH, wait for 1000mS (1 second), go to LOW, wait for another 1000mS, and then repeat. This scenario gives you the most basic of motor control, telling the motor when to go on and off:

```
void loop() {  
  
    digitalWrite(motorPin, HIGH);  
    delay(1000);  
  
    digitalWrite(motorPin, LOW);  
    delay(1000);  
  
}
```

## Controlling the Speed of Your Motor

On and off is all well and good, but sometimes you want greater control over the speed of your motor. In this section, you find out how to add an input into your circuit to give you full control of the motor on the fly.

# The MotorControl sketch

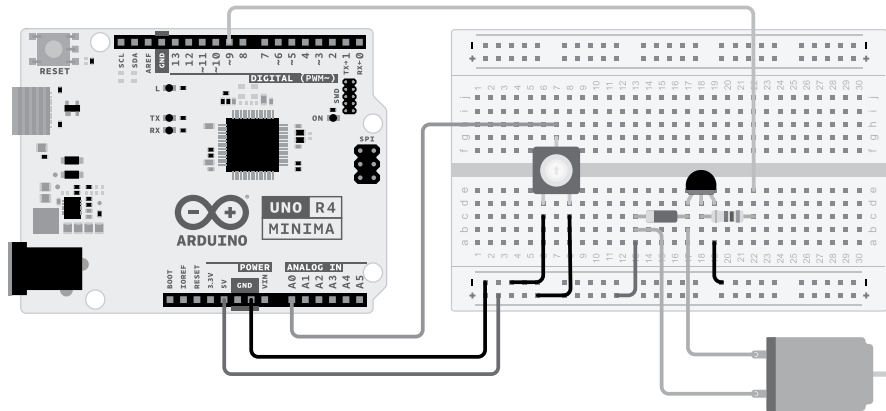
To gain control of the speed of your motor whenever you need it, add a potentiometer to your circuit.

You need the following:

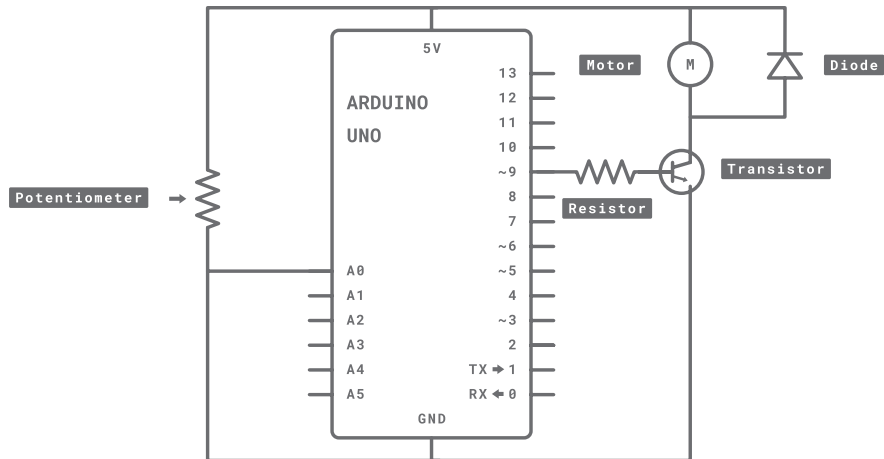
- » An Arduino Uno
- » A breadboard
- » A transistor
- » A DC motor
- » A diode
- » A 10k ohm variable resistor
- » A 2.2k ohm resistor
- » Jump wires

Follow the diagram in Figure 7-6 and the circuit diagram in Figure 7-7 to add a potentiometer alongside your motor control circuit.

**FIGURE 7-6:**  
A transistor circuit to drive your electric motor.



Find a space on your breadboard to place your potentiometer. The central pin of the potentiometer is connected back to pin 9 using a jump wire, and the remaining two pins are connected to 5V on one side and GND on the other. The 5V and GND can be on either side, but switching them will invert the value that the potentiometer sends to the Arduino. Although the potentiometer uses the same power and ground as the motor, note that they are separate circuits that both communicate through the Arduino.



**FIGURE 7-7:**  
A circuit diagram  
of a transistor  
circuit.

After you've built the circuit, open a new Arduino sketch and save it with another memorable name, such as `myMotorControl`. Then type the following code:

```
int potPin = A0;
int motorPin = 9;

int potValue = 0;
int motorValue = 0;

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

void loop() {
  potValue = analogRead(potPin);
  motorValue = map(potValue, 0, 1023, 0, 255);

  analogWrite(motorPin, motorValue);
  Serial.print("potentiometer = ");
  Serial.print(potValue);
  Serial.print("\t motor = ");
  Serial.println(motorValue);

  delay(2);
}
```

After you've typed the sketch, save it and click the Compile button to highlight any syntax errors.

If the sketch compiles correctly, click Upload to upload the sketch to your board. When it has finished uploading, you should be able to control your motor using the potentiometer. Turning the potentiometer in one direction speeds up the motor; turning it the other way slows down the motor. The next section explains how the code allows the potentiometer to change the speed.

## Understanding the MotorControl sketch

The MotorControl sketch is a variation on the AnalogInOutSerial sketch and works the same way, with a few name changes to better indicate what you are controlling and monitoring on the circuit.

As always, you declare the different variables used in the sketch. You use `potPin` to assign the potentiometer pin and `motorPin` to send a signal to the motor. The `potValue` variable is used to store the raw value of the potentiometer, and the `motorValue` variable stores the converted value that you want to output to the transistor to switch the motor:

```
int potPin = A0;
int motorPin = 9;

int potValue = 0;
int motorValue = 0;
```

For more details on the workings of this sketch, see the AnalogInOutSerial example in Chapter 6.

## Tweaking the MotorControl sketch

You may find that the motor just hums below a minimum speed. It does so because it doesn't have enough power to spin. By monitoring the values sent to the motor using the MotorControl sketch, you can find the motor's minimum value to turn, and then optimize `motorValue` to turn the motor within its true range.

To find the range of `motorValue`, follow these steps:

- 1. With the MotorControl sketch uploaded, click the serial monitor button at the top right of your Arduino window.**

The serial monitor window displays the potentiometer value followed by the output value being sent to the motor, in this fashion:

```
potentiometer = 1023  motor = 255
```

These values are displayed in a long list and are updated as you turn the potentiometer. If you don't see the list scrolling down, make sure that the Autoscroll option is selected.

2. **Starting with your potentiometer reading a value of 0, turn your potentiometer very slowly until the humming stops and the motor starts spinning.**
3. **Make a note of the value displayed at this point.**
4. **Use an `if` statement to tell the motor to change speed only if the value is greater than the minimum speed needed to spin the motor, as follows:**
  - a. *Find the part of your code that writes the `motorValue` to the motor:*

```
analogWrite(motorPin, motorValue);
```

- b. *Replace it with the following piece of code:*

```
if(motorValue > yourValue) {  
  analogWrite(motorPin, motorValue);  
} else {  
  digitalWrite(motorPin, LOW);  
}
```

5. **Replace `yourValue` with the number that you noted.**

If the value is greater than `motorValue`, the motor speeds up. If it is lower, the pin is written `LOW` so that it is fully off. You could also type `analogWrite(motorPin, 0)` to accomplish the same thing. Tiny optimizations like this can help your project function smoothly, with no wasted movement or values.

## Getting to Know Servomotors

A servomotor (or servo) is made up of a motor and a device called an encoder that can track the rotation of the motor. *Servomotors* are used for precision movements, moving by a number of degrees to an exact location. Using your Arduino, you can tell the servomotor what degree you want it to move to, and it will go there from its current position. Most servomotors can move only 180 degrees, but you can use gearing to extend this range.

The servo in your kit will most likely be a hobby servo, similar to those shown in Figure 7-8. A hobby servomotor has plastic gears and can manage only relatively light loads. After you experiment with small servos, you have plenty of larger ones to choose from for heavy lifting. Servos are widely used in the robotics community for walkers that need precise movement in each of their feet.

**FIGURE 7-8:**  
Two  
servomotors.



The examples in the following section walk you through the basic operations of sending signals to a servo and controlling one directly with a potentiometer.

## Creating Sweeping Movements

This first servomotor example requires only a servomotor and will allow you to turn the motor through its full range of movement. The servo sweeps from  $0^\circ$  to  $179^\circ$  and then back again, in a similar way to the movement of an old rotary clock.

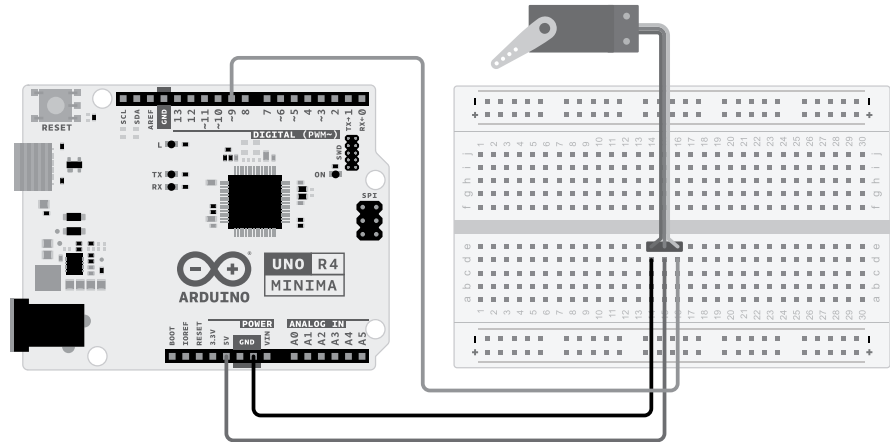
### The Sweep sketch

You need the following for the Sweep sketch:

- » An Arduino Uno
- » A servo
- » Jump wires

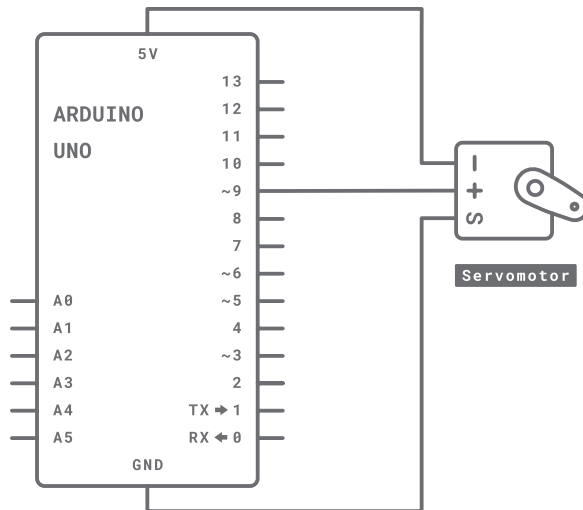
The wiring for a servo is simple because it comes with a neat three-pin socket. To connect it to your Arduino, simply use jump wires between the Arduino pins and the servo sockets directly or use a set of header pins to connect the socket to your breadboard.

As shown in Figures 7-9 and 7-10, the servo has a set of three sockets with wires connected to them, usually red, black, and white. All the calculations and readings to move the motor are done on the circuitry inside the servo itself, so all that is needed is power and a signal from the Arduino.



**FIGURE 7-9:**  
A servomotor  
wired to your  
Arduino.

Red is connected to 5V on the Arduino to power the motor and the circuitry inside it; black is connected to GND to ground the servo; and white is connected to pin 9 to control the servo's movement. The colors of these wires can vary, so always check the datasheet or any available documentation for your specific motor. Other common colors are red (5V), brown (GND), and yellow (signal).



**FIGURE 7-10:**  
A circuit diagram  
of the servo  
circuit.

Complete the circuit as described and open the Sweep sketch by choosing File ⇨ Examples ⇨ Servo ⇨ Sweep.



TIP

The Servo library is included with the Arduino software, but depending on your version of the Arduino IDE, it may not appear in the Examples menu straight away. If you don't see Servo ⇨ Sweep, open the Library Manager and install the Servo library, as described in Chapter 12.

The Sweep sketch is as follows:

```
/* Sweep
by BARRAGAN <http://barraganstudio.com>
This example code is in the public domain.

modified 8 Nov 2013
by Scott Fitzgerald
https://www.arduino.cc/en/Tutorial/LibraryExamples/Sweep
*/

#include <Servo.h>

Servo myservo;          // create servo object to control a servo
                        // twelve servo objects can be created on most boards

int pos = 0;           // variable to store the servo position

void setup() {
  myservo.attach(9);    // attaches the servo on pin 9 to the servo object
}

void loop() {
  for(pos = 0; pos < 180; pos += 1)    // goes from 0 degrees to 180 degrees
  {                                     // in steps of 1 degree
    myservo.write(pos);                // tell servo to go to position in
                                        // variable 'pos'
    delay(15);                          // waits 15ms for the servo to reach
                                        // the position
  }
  for(pos = 180; pos >= 1; pos -= 1)    // goes from 180 degrees to 0 degrees
  {                                     // in steps of 1 degree
    myservo.write(pos);                // tell servo to go to position in
                                        // variable 'pos'
    delay(15);                          // waits 15ms for the servo to reach
                                        // the position
  }
}
```

After you find the sketch, click the Compile button to check the code. The compiler should, as always, highlight any grammatical errors in red in the message area.

If the sketch compiles correctly, click Upload to upload the sketch to your board. When the sketch has finished uploading, your motor should start turning backward and forward through 180 degrees, doing a dance on the table.

If nothing happens, you should double-check your wiring:

- » Make sure that you're using pin 9 for the data (white/yellow) line.
- » Check that you have the other servo wires connected to the correct pins.

## Understanding the Sweep sketch

A servo library is included at the start of the Sweep sketch. This library will help you to get a lot out of your servo with little complex code:

```
#include <Servo.h>
```

The next line makes a servo object. The library knows how to use servos but needs you to give each one a name so that it can talk to each one. In this case, the new servo object is called `myservo`. Using a name is similar to naming your variables; that is, they can be any name as long as they're consistent throughout your code and you don't use any Arduino reserved names, such as `int` or `delay`:

```
Servo myservo; // create servo object to control a servo
               // twelve Servo objects can be created on most boards
```

The final line in the declarations is a variable to store the position of the servo:

```
int pos = 0; // variable to store the servo position
```

In the `setup` function, the only item to set is the pin number of the Arduino pin that is communicating with the servo. In this case, you are using pin 9, but it could be any PWM pin.

```
void setup()
{
  myservo.attach(9); // attaches the servo on pin 9 to the servo object
}
```

The `loop` function performs two simple actions, and both are `for` loops. The first `for` loop gradually increases the `pos` variable from 0 to 180. Because of the library, you can write values in degrees rather than the normal 0 to 255 used for PWM control. With every loop, the value is increased by 1 and sent to the servo using a function specific to the servo library, `<servoName>.write(<value>)`. After the loop updates the value, a short delay of 15 milliseconds occurs while the servo reaches its new location.



REMEMBER

In contrast to other outputs, after a servo is updated, it starts moving to its new position instead of needing to be told to do so.

```
void loop()
{
  for(pos = 0; pos < 180; pos += 1) // goes from 0 degrees to 180 degrees
  {
    myservo.write(pos);           // tell servo to go to position in
                                  // variable 'pos'
    delay(15);                    // waits 15ms for the servo to reach
                                  // the position
  }
}
```

The second `for` loop does the same in the opposite direction, returning the servo to its start position:

```
for(pos = 180; pos >= 1; pos -= 1) // goes from 180 degrees to 0 degrees
{
  myservo.write(pos);           // tell servo to go to position in
  // variable 'pos'
  delay(15);                    // waits 15ms for the servo to reach
  // the position
}
}
```

It's a good idea to use this simple servo example to test whether your servo is working correctly, before coding more complex examples.

## Controlling Your Servo

Now that you have mastered control of the servo, you can try something with a bit more interaction. By using a potentiometer (or any analog sensor), you can directly control your servo in the same way that you'd control a mechanical claw at the arcades.

# The Knob sketch

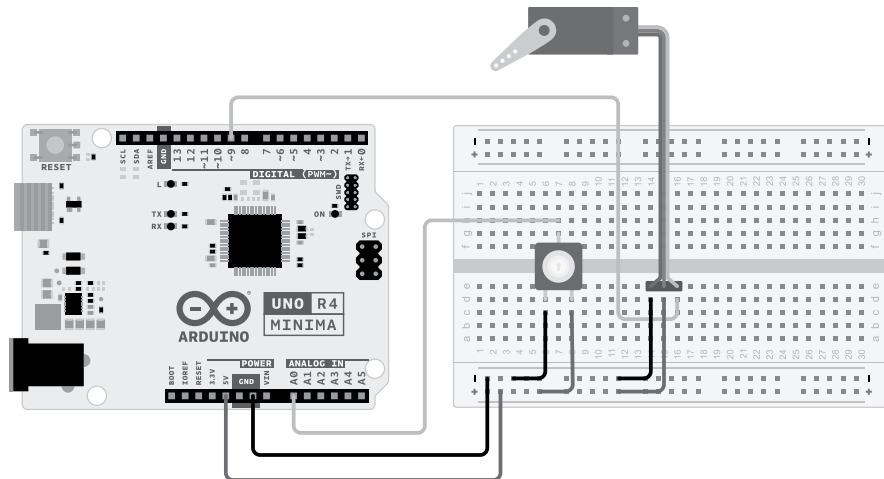
The Knob example shows you how you can easily use a potentiometer to move your servo to a specific degree.

You need the following:

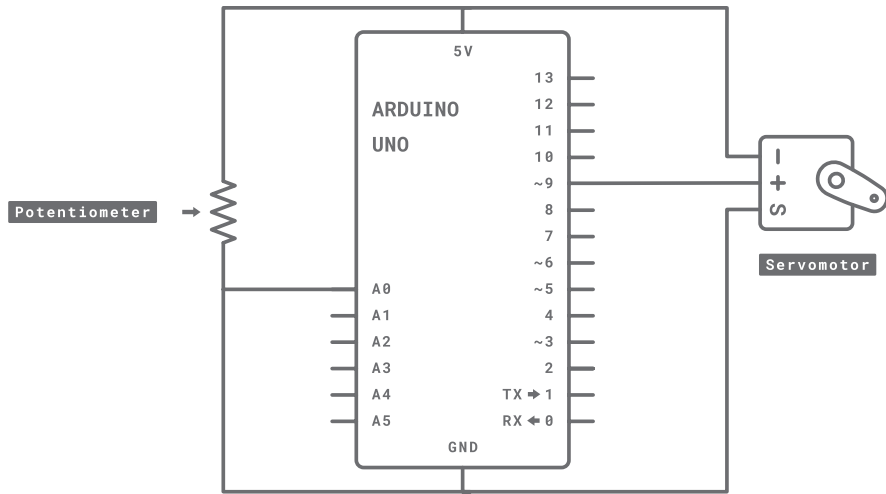
- » An Arduino Uno
- » A breadboard
- » A servo
- » A 10k ohm variable resistor
- » Jump wires

The servo is wired exactly as in the Sweep example, but this time you need extra connections to 5V and GND for the potentiometer, so you must use a breadboard to provide the extra pins. Connect the 5V and GND pins on the Arduino to the positive (+) and negative (-) rows on the breadboard. Connect the servo to the breadboard by using either a row of three header pins or three jump wires.

Connect the red socket to the 5V row, the black/brown socket to the GND row, and the white/yellow socket to pin 9 on the Arduino. Find a space on the breadboard for the potentiometer. Connect the center pin to pin A0 on the Arduino and the remaining pins to 5V on one side and GND on the other. Refer to the circuit diagram in Figure 7-11 and the schematic in Figure 7-12.



**FIGURE 7-11:**  
A servomotor  
with a control  
knob.



**FIGURE 7-12:**  
A circuit diagram for a servomotor and a potentiometer.

After you have built the circuit, open the sketch by choosing File ⇨ Examples ⇨ Servo ⇨ Knob. The code for the sketch is as follows:



TIP

The Servo library is included with the Arduino software, but depending on your version of the Arduino IDE, it may not appear in the Examples menu straight away. If you don't see **Servo > Knob**, open the Library Manager and install the Servo library, as described in Chapter 12.

```

/*
Controlling a servo position using a potentiometer (variable resistor)
by Michal Rinott <http://people.interaction-ivrea.it/m.rinott>

modified on 8 Nov 2013
by Scott Fitzgerald
http://www.arduino.cc/en/Tutorial/Knob
*/

#include <Servo.h>

Servo myservo;           // create servo object to control a servo

int potpin = 0;          // analog pin used to connect the potentiometer
int val;                 // variable to read the value from the analog pin

void setup() {
  myservo.attach(9);     // attaches the servo on pin 9 to the servo object
}

```

```

void loop() {
  val = analogRead(potpin);           // reads the value of the potentiometer
                                       // (value between 0 and 1023)
  val = map(val, 0, 1023, 0, 179);    // scale it to use it with the
                                       // servo (value between 0 and 180)
  myservo.write(val);                 // sets the servo position according
                                       // to the scaled value
  delay(15);                          // waits for the servo to get there
}

```



TIP

With all Arduino examples, it's best to assume that they are works-in-progress and may not always be accurate. You may have noticed a few discrepancies between the comments and the code. When referring to the range of degrees to move the servo, the sketch mentions both 0 to 179 and 0 to 180. The correct range is 0 to 179, which gives you 180 values. Counting from zero is referred to as *zero indexing* and is a common occurrence in Arduino, as you may have noticed by this point.

After you've found the sketch, click the Compile button to check the code. If the compiler encounters any syntax errors, they are highlighted in the message area, which lights up red.

If the sketch compiles correctly, click Upload to upload the sketch to your board. When it has finished uploading, your servo should turn as you turn your potentiometer.

If that isn't what happens, you should double-check your wiring:

- » Make sure that you're using pin 9 to connect the data (white/yellow) line to the servo.
- » Check your connections to the potentiometer and make sure that the center pin is connected to analog pin 0.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the Knob sketch

In the declarations, the servo library, `Servo.h`, and a new servo object are named. The analog input pin is declared with a value of 0, showing that you are using analog 0.



TIP

You may have noticed that the pin is numbered 0, not A0 as in other examples. Either is fine, because A0 is just an alias of 0, as A1 is of 1, and so on. Using A0 is good for clarity but optional.

One last variable stores the value of the reading, which will become the output:

```
#include <Servo.h>

Servo myservo;          // create servo object to control a servo

int potpin = 0;         // analog pin used to connect the potentiometer
int val;                // variable to read the value from the analog pin
```

In the setup function, the only item to define is `myservo`, which is using pin 9:

```
void setup()
{
  myservo.attach(9);    // attaches the servo on pin 9 to the servo object
}
```

Rather than use two separate variables for input and output, the Knob sketch simply uses one. First, `val` is used to store the raw sensor data, a value from 0 to 1023. The `map` function processes this value by scaling its range to that of the servo: 0 to 179. `myservo.write` then writes this value to the servo. There is a 15-millisecond delay to allow the servomotor to reach its destination. Then the loop repeats and updates the position of the servo as necessary:

```
void loop()
{
  val = analogRead(potpin);          // reads the value of the potentiometer
                                     // (value between 0 and 1023)
  val = map(val, 0, 1023, 0, 179);   // scale it to use it with the servo
                                     // (value between 0 and 180)
  myservo.write(val);                // sets the servo position according
                                     // to the scaled value
  delay(15);                          // waits for the servo to get there
}
```

With this simple addition to the circuit, it's possible to control a servo with any sort of input. In this example, the code uses an analog input, but with a few changes it could just as easily use a digital input.

# Making Noises

If you've just finished the motor sketches, you have mastered movement and must be ready for a new challenge. In this section, you look at a project that's a bit more tuneful than the previous ones: making music (or noise at least) with your Arduino. Yes, you can make electronic music — albeit simple — using a piezo buzzer.

## Piezo buzzer

A piezo or piezoelectric buzzer is found in hundreds of thousands of devices. If you hear a tick, buzz, or beep, it's likely caused by a piezo. The *piezo* is composed of two layers, a ceramic and a metal plate joined together. When electricity passes from one layer to the other, the piezo bends on a microscopic level and makes a sound, as shown in Figure 7-13.

**FIGURE 7-13:**  
An exaggeration  
of the miniature  
movements  
of a piezo.



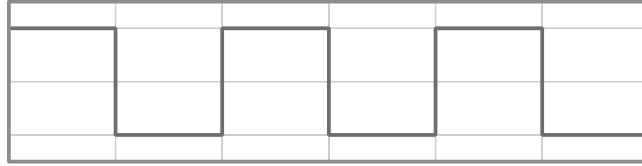
If you switch between a voltage and ground, the piezo bends and generates a tick sound; if this happens fast enough, these ticks turn into a tone. This tone can be quite harsh, similar to the old mobile phone ringtone or computer game sounds from the 1980s, and is known as a square wave. Every time the piezo changes polarity fully, it produces a square wave with abrupt, hard edges, like a square. Other types of waves include triangle waves and sine waves, which are progressively less harsh. Figure 7-14 is an illustration of these waves so you can see the differences between them.

Piezos generate square waves, resulting in a buzzing sound. The buzzer isn't restricted to just one pitch. By changing the frequency that the buzzer is switched at (the width between the square waves), you can generate different frequencies and therefore different notes.

## The toneMelody sketch

With the `toneMelody` sketch, you see how to change the frequency of your piezo and play a predefined melody. This circuit allows you to program your own sounds. With a bit of time and consideration you can turn these sounds into melodies.

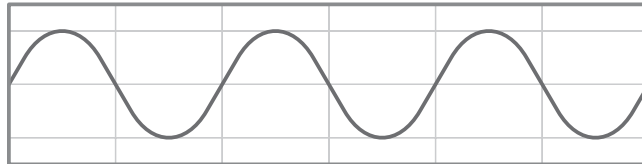
Square



Triangle



Sine



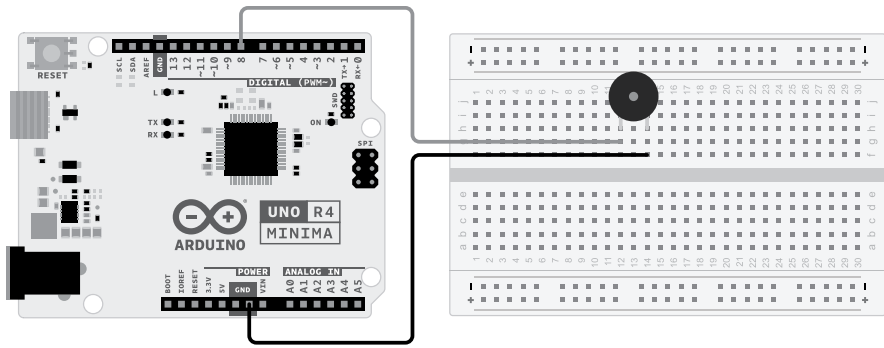
**FIGURE 7-14:** Square, triangle, and sine waves have different shapes, which produce different sounds.

Piezo buzzers are supplied in most Arduino kits but can take many different forms. They can be supplied without an enclosure, exposing the brass and ceramic disks, whereas others are enclosed in plastic housing ranging from small cylinders to flat coin-like shapes. They may also have different connections, either a set of two pins protruding from the underside of the piezo or two wires protruding from its side. For the `toneMelody` sketch, you need the following:

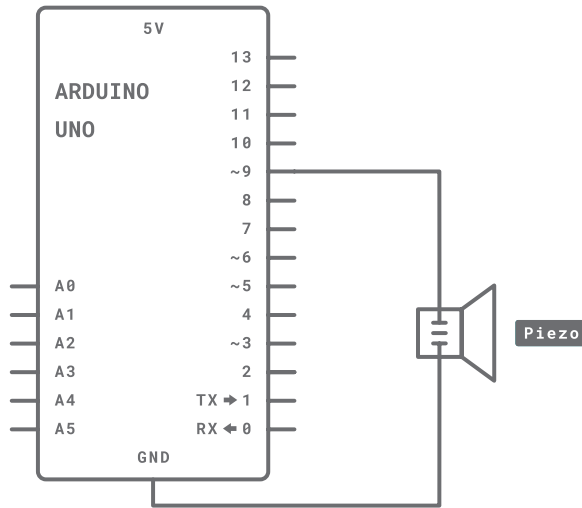
- » An Arduino Uno
- » A breadboard
- » A piezo buzzer
- » Jump wires

Connect the piezo buzzer to the breadboard and use a set of jump wires to connect it to digital pin 8 on one side and ground on the other. Some piezos have a polarity, so make sure that you connect the positive (+) to pin 8 and the negative (-) to GND. Other piezos don't have a polarity, so if you don't see any symbols, don't worry. The piezo circuit is shown in Figure 7-15, and the circuit diagram appears in 7-16.

**FIGURE 7-15:**  
A piezo  
buzzer circuit.



**FIGURE 7-16:**  
A circuit diagram  
of a piezo  
buzzer circuit.



Complete the circuit and open the sketch by choosing **File** ⇨ **Examples** ⇨ **02.Digital** ⇨ **toneMelody**. You see the following code:

```

/*
Melody

Plays a melody

circuit:
* 8-ohm speaker on digital pin 8

created 21 Jan 2010
modified 30 Aug 2011
by Tom Igoe

```

This example code is in the public domain.

```
https://docs.arduino.cc/built-in-examples/digital/toneMelody/

*/
#include "pitches.h"

// notes in the melody:
int melody[] = {
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4};

// note durations: 4 = quarter note, 8 = eighth note, etc.:
int noteDurations[] = {
  4, 8, 8, 4,4,4,4,4 };

void setup() {
  // iterate over the notes of the melody:
  for (int thisNote = 0; thisNote < 8; thisNote++) {

    // to calculate the note duration, take one second
    // divided by the note type.
    //e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
    int noteDuration = 1000/noteDurations[thisNote];
    tone(8, melody[thisNote],noteDuration);

    // to distinguish the notes, set a minimum time between them.
    // the note's duration + 30% seems to work well:
    int pauseBetweenNotes = noteDuration * 1.30;
    delay(pauseBetweenNotes);
    // stop the tone playing:
    noTone(8);
  }
}

void loop() {
  // no need to repeat the melody.
}
```

In the `toneMelody` sketch, you have another tab called `pitches.h`, which contains all the data needed to make the correct tones with your buzzer. In your Arduino sketch folder, this tab (and other, additional tabs) appears as its own individual file and must be included in the main sketch using the `#include` function followed by the name of the file to be included. In this case, the code reads `#include`

"`pitches.h`". After you've found the sketch, click the Compile button to check the code. The message area highlights grammatical errors in red if any are discovered.

If the sketch compiles correctly, press Upload to upload the sketch to your board. When it has finished uploading, you should hear a buzzer that sings a tune to you and then stops. To hear the tune again, press the reset button on your Arduino.

If you don't hear a buzzer, you should double-check your wiring:

- » Make sure you're using pin 8 as your output.
- » Check that your piezo is correctly positioned. Symbols may be hidden on the underside if they are not visible on the top. If you don't see any markings, try the piezo in the other orientation.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the sketch

The `toneMelody` sketch is the first one in this book that uses multiple tabs. You sometimes use *tabs* as a convenient way of separating sketches. In this case, the `pitches.h` tab is a reference or lookup table for all the possible notes in the piezo's range. Because this code won't change, it doesn't need to be in the main body of code.

At the top of the `toneMelody` sketch is a note to include `pitches.h`, which is treated in the same way as a library. It is an external file that can be brought into sketches if needed. In this case, we need it to determine which frequencies are used to create the notes:

```
#include "pitches.h"
```

Now that the sketch knows the different notes, the melody is defined in an array so that the notes can be stepped through in order. To find out more about arrays, see the "Introducing Arrays" sidebar. The names, such as `NOTE_C4`, refer to the names of notes in the `pitches.h` tab. If you look at `pitches.h`, you will see that it uses a C function called `define` for each of these note references and follows them with a number, such as `#define NOTE_C4 262`. So whenever `NOTE_C4` is mentioned, it is really just a variable name for the value 262:

```
// notes in the melody:  
int melody[] = {  
  NOTE_C4, NOTE_G3, NOTE_G3, NOTE_A3, NOTE_G3, 0, NOTE_B3, NOTE_C4};
```

## INTRODUCING ARRAYS

In its simplest form, an *array* is a list of data. Think of it as being like a shopping list, as shown in the following table. Each row has a number, referred to as the *index*, and the data contained in that part of the list. This kind of array is a one-dimensional array, containing only one item of data for each, in this case the name of a piece of fruit.

In computing, it's common to use *zero indexing*, which means starting your list from zero, as shown in the following table.

Index	Value
0	apples
1	bananas
2	oranges

How is an array relevant to Arduino? Arrays can store integers, floats, characters, or any other type of data, but I use integers here to keep things simple. Here is an array of six integer values:

```
int simpleArray[] =
    {1, 255, -51, 0, 102, 27};
```

First, `int` defines the type of data being stored as integers (whole numbers). The data type could also be `float` for floating-point numbers or `char` for characters. The name of the array is `simpleArray` but can be any relevant name that best describes your array. The square brackets (`[]`) store the length of the array (the number of values that can be stored in the array); in this case, the space is blank, which means that this array has no fixed length. The numbers inside the curly braces `{ }` are values defined in the array. These are optional, so if they are not defined, the array will be left empty.

There are other correct ways to declare arrays, including the following:

```
int simpleArray[10];
float simpleArray[5] = {2.7, 42.1, -9.1, 300.6};
char simpleArray[14] = "hello, world!";
```

Note that the character array entry has a number one greater than the number of characters. Remember this requirement if you're getting errors.

Now that your array is defined, you need to know how to use it. To use values in an array, you refer to them by their index. If you wanted to send a value to the serial monitor, you would write the following:

```
Serial.println(simpleArray[2]);
```

(continued)

(continued)

This line would display a value of `-51` because that is the value stored in index 2 of the array.

You can also update values in the array. An effective way to update is with a `for` loop, to count through each index in the array (see Chapter 10 for more details) as in the following example:

```
for (int i = 0; i < 6; i++) {  
  simpleArray[i] = analogRead(sensorPin);  
}
```

The `for` loop in this case will loop six times, increasing the `i` variable by 1 each loop. The `i` variable is used also to represent the index of the array, so with each loop, a new analog reading from `sensorPin` is stored in the current index and the index of the array is incremented for the next loop.

This loop is a clever and efficient way to work through arrays, either using or updating the data stored in them. Arrays can get even more complicated, storing many strings of text, or they can be multidimensional, like a spreadsheet, with many values associated with each index. For more information, head over to the official Arduino reference page on arrays at [docs.arduino.cc/language-reference/en/variables/data-types/array/](https://docs.arduino.cc/language-reference/en/variables/data-types/array/).

Without the beat, your melody wouldn't sound right, so another array stores the duration for each note:

```
// note durations: 4 = quarter note, 8 = eighth note, etc.:  
int noteDurations[] = {  
  4, 8, 8, 4,4,4,4,4 };
```

In `setup`, a `for` loop is used to cycle through each of the eight notes, from 0 to 7. The `thisNote` value is used as an index to point to the correct items in each array:

```
void setup() {  
  // iterate over the notes of the melody:  
  for (int thisNote = 0; thisNote < 8; thisNote++) {
```

The duration is calculated by dividing 1,000 (or 1 second) by the required duration, 4 for a quarter note or crotchet, 8 for an eighth note or quaver, and so on. This value is then written to the function `tone`, which sends the current note to pin 8 for the assigned duration:

```
// to calculate the note duration, take one second
// divided by the note type.
//e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
int noteDuration = 1000/noteDurations[thisNote];
tone(8, melody[thisNote],noteDuration);
```

A small pause between notes is used so that they are better defined. In this case, the pause is set to 130 percent of the note duration, giving each note time to finish and leaving a small gap before the next one so the melody sounds clearer:

```
// to distinguish the notes, set a minimum time between them.
// the note's duration + 30% seems to work well:
int pauseBetweenNotes = noteDuration * 1.30;
delay(pauseBetweenNotes);
```

Next, the `noTone` function is used to turn off pin 8, stopping the note after it has played for its duration:

```
// stop the tone playing:
noTone(8);
}
}
```

In the loop, nothing happens. As it stands, the melody plays once at the start and then ends. The melody could be moved to the loop to play forever, but this decision may cause mild headaches:

```
void loop() {
  // no need to repeat the melody.
}
```



TIP

The `toneMelody` sketch is a great example of using a melody as an audio signal at the start of a sketch. Audio feedback can be a great alternative or addition to visual feedback in your project.

## Making an Instrument

In the preceding section, you find out how to make your project play a sound rather than blink a light, as in previous sketches. In the example in this section, you see how to go beyond playing a sound — you create your own instrument, similar to the Theremin. The *Theremin*, named after its inventor Léon Theremin,

was one of the first electronic instruments, developed in the 1920s. It worked by detecting the electromagnetic field of the player's hands to change signals: one hand for volume and the other for pitch.

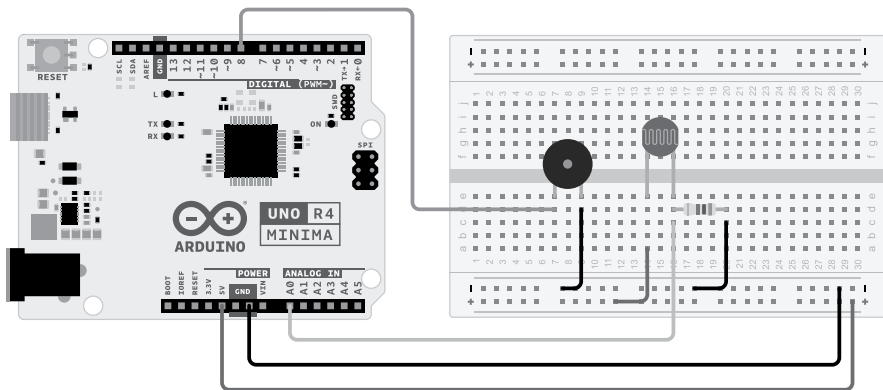
## The PitchFollower sketch

In the PitchFollower sketch, you find out how to make a budget Theremin by using a piezo as a light sensor to control the pitch.

You need the following:

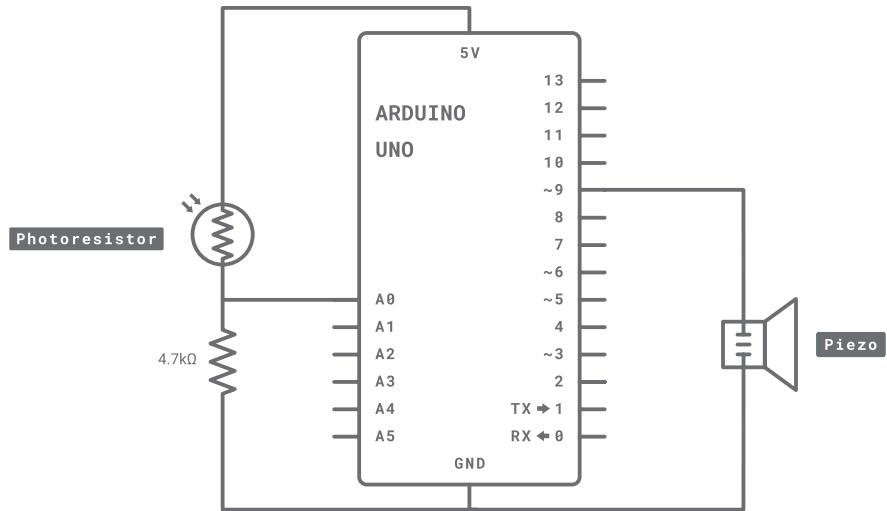
- » An Arduino Uno
- » A breadboard
- » A piezo
- » A light sensor
- » A 4.7k ohm resistor
- » Jump wires

This circuit has two separate halves: the piezo and the light sensor circuit. The piezo is wired as in the toneMelody sketch, with one wire to digital pin 9 and the other to GND. The light sensor is connected to analog 0 on one side and 5V on the other; the 4.7K resistor is connected between analog 0 and ground (as shown in Figures 7-17 and 7-18). If you do not have a 4.7K resistor, use the nearest you have to that value.



**FIGURE 7-17:**  
A light-sensor-  
controlled  
Theremin circuit.

**FIGURE 7-18:**  
A circuit diagram of the light-sensor-controlled Theremin.



Complete the circuit, and open the sketch by choosing File ⇨ Examples ⇨ 02.Digital ⇨ tonePitchFollower:

```

/*
Pitch follower

Plays a pitch that changes based on a changing analog input

circuit:
* 8-ohm speaker on digital pin 9
* photoresistor on analog 0 to 5V
* 4.7K resistor on analog 0 to ground

created 21 Jan 2010
modified 9 Apr 2012
by Tom Igoe

This example code is in the public domain.

https://docs.arduino.cc/built-in-examples/digital/tonePitchFollower/
*/

void setup() {
// initialize serial communications (for debugging only):
Serial.begin(9600);
}

```

```

void loop() {
  // read the sensor:
  int sensorReading = analogRead(A0);
  // print the sensor reading so you know its range
  Serial.println(sensorReading);
  // map the analog input range (in this case, 400 - 1000 from the
  // photoresistor) to the output pitch range (120 - 1500Hz)
  // change the minimum and maximum input numbers below depending on the
  // range your sensor's giving:
  int thisPitch = map(sensorReading, 400, 1000, 120, 1500);

  // play the pitch:
  tone(9, thisPitch, 10);
  delay(1); // delay in between reads for stability
}

```

After you've found the sketch, click the Compile button to check the code. Any syntax errors turn the message area red when they are discovered, and you see an error message stating what is wrong.

If the sketch compiles correctly, press Upload to upload the sketch to your board. When it has finished uploading, you should have a light sensor that will change the pitch of your buzzer. If you don't hear a change, make sure that you are in a well-lit area or turn a desk lamp on over your breadboard to increase the difference when you cover the light sensor with your hand.

If nothing happens, you should double-check your wiring:

- » Make sure that you're using the correct pin number for the inputs and outputs.
- » Check that your piezo is turned the correct way. Symbols may be hidden on the underside if they are not visible.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the sketch

This sketch is a lot shorter than the `toneMelody` sketch, presented earlier in the chapter, because it converts the readings from the light sensor to a frequency directly rather than requiring a lookup table. Because you are converting readings, you can slide between notes as well as choose them individually.

In the `setup` function, the serial port is opened to allow you to monitor the sensor readings as they come in:

```
void setup() {  
  // initialize serial communications (for debugging only):  
  Serial.begin(9600);  
}
```

In the main loop, the light sensor is read from analog pin 0. This reading is also forwarded to the serial monitor:

```
void loop() {  
  // read the sensor:  
  int sensorReading = analogRead(A0);  
  // print the sensor reading so you know its range  
  Serial.println(sensorReading);  
}
```

To convert the sensor's range to the range of frequencies that the buzzer can cover, you use the `map` function:

```
// map the analog input range (in this case, 400 - 1000 from the  
// photoresistor) to the output pitch range (120 - 1500Hz)  
// change the minimum and maximum input numbers below depending on the  
// range your sensor's giving:  
int thisPitch = map(sensorReading, 400, 1000, 100, 1000);
```

The `tone` function then outputs the note with the mapped sensor value and a very short duration of 10 milliseconds. This duration serves to make the sound audible, but the real duration is determined by how long you hold your hand over the sensor, as described previously:

```
// play the pitch:  
tone(9, thisPitch, 10);
```

Finally, a tiny delay occurs at the end of the loop to improve the stability for the readings:

```
delay(1); // delay in between reads for stability  
}
```

With this setup, you can quickly make an easy controller and maybe even form a traveling Theremin band with your friends.





# **Building on the Basics**

## **IN THIS PART . . .**

See some of the varied uses of Arduino through a few projects already out in the world.

Make your basic prototypes into something more solid by learning about soldering.

Use code to improve the reliability of your project.

Learn how to choose the right sensors for your projects.

Discover new possibilities by using shields and libraries.

#### IN THIS CHAPTER

- » Exploring iconic Arduino projects from art, design, and industry
- » Understanding how each project worked and why it mattered
- » Finding inspiration for your own creations using examples from this book

## Chapter 8

# Learning by Example

One of the best ways to learn about Arduino is by seeing what others have done with it. Since its release, Arduino has powered an enormous variety of projects — from interactive artworks and scientific tools to connected consumer products. Some of these projects have gone on to shape how designers and engineers think about hardware prototyping.

In this chapter, you look at a handful of influential projects that show what's possible with simple boards and creative thinking. Each one demonstrates a different aspect of Arduino — sensors, connectivity, motors, and lighting — and shows how small ideas can scale into polished, professional outcomes. You also see how to build smaller, related examples using the circuits and sketches covered throughout this book.

## Chorus

Chorus was a kinetic light and sound installation by United Visual Artists (UVA), a London-based art and design collective. The piece consisted of tall pendulums swinging in perfect synchrony, each fitted with lights and speakers that filled the

space with rhythm and movement. (See Figure 8-1.) Visitors could walk beneath the pendulums as the pattern shifted from chaos to calm, creating a mesmerizing, immersive experience.



**FIGURE 8-1:**  
Chorus in  
full swing.

*Courtesy of United Visual Artists*

## How it works

Each pendulum contained an Arduino that controlled motors, lights, and sound. The swing motion came from a motor and reduction gearbox, which the Arduino triggered through a relay circuit. Every pendulum also had custom LED boards and a speaker in its base. The Arduinos communicated with custom software that synchronized all the movements to a musical score composed by Mira Calix.

The setup demonstrates how a single board can coordinate multiple outputs — light, sound, and motion — while responding to real-time instructions from a central controller.

## Lessons learned

- » Arduino can scale up to large, coordinated systems when combined with solid engineering.

- » Simple circuits (LEDs, motors, relays) become powerful when used in repetition and synchrony.
- » Collaboration between artists, programmers, and engineers creates experiences that none could achieve alone.

## Build your own

You can experiment with similar principles on a smaller scale using servos, LEDs, and an Arduino Uno R4. Try the motor and LED examples in Chapters 10 and 12, combining motion and light to create your own mini installation. You could even synchronize multiple boards using serial communication (Chapter 7) or shared timing signals.

## Further reading

You can find the project page at UVA's website: <https://www.uva.co.uk/works/chorus>. In addition, an excellent paper by Vince Dziekan looks at both UVA's working practice and the Chorus project in detail: <https://fibreculturejournal.org/wp-content/pdfs/FCJ-122Vince%20Dziekan.pdf>.

# Push Snowboarding

Developed by Vitamins Design (now Special Projects) in collaboration with Nokia and Burton Snowboards, Push Snowboarding captured a snowboarder's movement and physical data during a run. It used small, wireless sensor modules to record balance, acceleration, heart rate, and more, translating those readings into visuals overlaid on live video. The result was a complete picture of both performance and physical response. (See Figure 8-2.)

## How it works

Each sensor unit was built around an Arduino Pro Mini for its compact size and low power consumption. Modules included galvanic skin response sensors, heart-rate monitors, pressure sensors, and 3D motion sensors (IMUs). These communicated wirelessly via Bluetooth to a smartphone in the rider's pocket, which processed and visualized the data.



**FIGURE 8-2:**  
Snowboarding  
sensors in  
customized  
sensor boxes.

*Courtesy of Special Projects*

The enclosures were 3D printed, weatherproofed, and fitted with rechargeable lithium batteries — a reminder that even prototypes can look like finished products when designed carefully.

## Lessons learned

- » Sensor data is only as valuable as how you interpret and visualize it.
- » Compact microcontrollers like the Nano or Seeed studio's XIAO make wearables and embedded systems practical.
- » Wireless communication unlocks new kinds of feedback and interaction.

## Build your own

You can use similar sensors from Chapter 9 — such as accelerometers, temperature sensors, or pulse sensors — to build your own data logger and add serial communication as shown in Chapter 7. Even a simple project, such as tracking motion or environment data, follows the same principles as Push Snowboarding.

## Further reading

You can find more details about the project on the Special Projects' website at [specialprojects.studio/project/push-snowboarding/](http://specialprojects.studio/project/push-snowboarding/).

## Baker Tweet

When the Albion Café in London opened, creative agency Poke built Baker Tweet — a small box that let bakers announce freshly baked goods on Twitter at the push of a button. It was an early example of an Internet of Things (IoT) device that turned a mundane task into something delightful and useful. (See Figure 8-3.)



**FIGURE 8-3:**  
Baker Tweet  
in the Albion  
Bakery.

*Courtesy of Poke London*

## How it works

The device had a simple physical interface: a rotary dial to select an item, a button to send the tweet, and an LCD screen to show confirmation. Inside, an Arduino controlled these inputs and connected to the internet via an Ethernet shield or Wi-Fi adapter. A small web app allowed café staff to update the menu remotely.

When the baker pressed the button, the Arduino sent the message through the network to Twitter’s API — no keyboard, phone, or computer required.

## Lessons learned

- » Arduino can bring the internet into everyday tools.
- » Tangible interfaces are often more intuitive than screens.
- » A simple, well-made prototype can become a finished product when it meets a real need.

## Build your own

Combine the input and output examples from Chapters 6 and 7 to make your own connected notifier. You could post updates to a web dashboard, trigger an email, or control a cloud-connected LED from a button press. The concept remains the same: physical action leads to online response.

## Further reading

Sadly there are only a few sources of documentation on the Internet, such as the MoMA (<https://www.moma.org/interactives/exhibitions/2011/talktome/objects/146207/>). But luckily you can still take a look at a few excellent prototyping photos that show the development of the project from breadboard to bakery ([flickr.com/photos/aszoly/sets/72157614293377430/](https://www.flickr.com/photos/aszoly/sets/72157614293377430/)).

# Interactive Plan Chests and Compass Cards

London design studio Kin created *The Compass Lounge* as part of a new gallery at the National Maritime Museum. The installation let visitors browse digital collections through large touchscreens and interact with physical drawers, lights, and scanning points — blending physical and digital discovery. (See Figure 8-4.)

**FIGURE 8-4:**  
The hidden LED display lights up underneath the wallpaper.



Courtesy of Kin

## How it works

Each drawer in the plan chests contained a microswitch connected to an Arduino. When opened, the switch activated the screen; when closed, it turned the screen off to prevent image burn. Hidden LED grids behind wallpaper displayed reference numbers for each artifact, glowing through the fabric only when active.

Visitors also received *Compass Cards*, physical passes with barcodes. Scanning them at collection points (each containing an Arduino with a barcode reader and Ethernet shield) logged items to a central server and stamped the card physically. Visitors could later explore their digital collection online. (See Figure 8-5.)

## Lessons learned

- » Arduinos can serve as reliable controllers in public installations when built with care.
- » Combining simple sensors and actuators can produce highly refined experiences.
- » Blending physical and digital interfaces encourages engagement and learning.

**FIGURE 8-5:**  
A look inside the  
Compass card  
collection points.



*Courtesy of Kin*

## Build your own

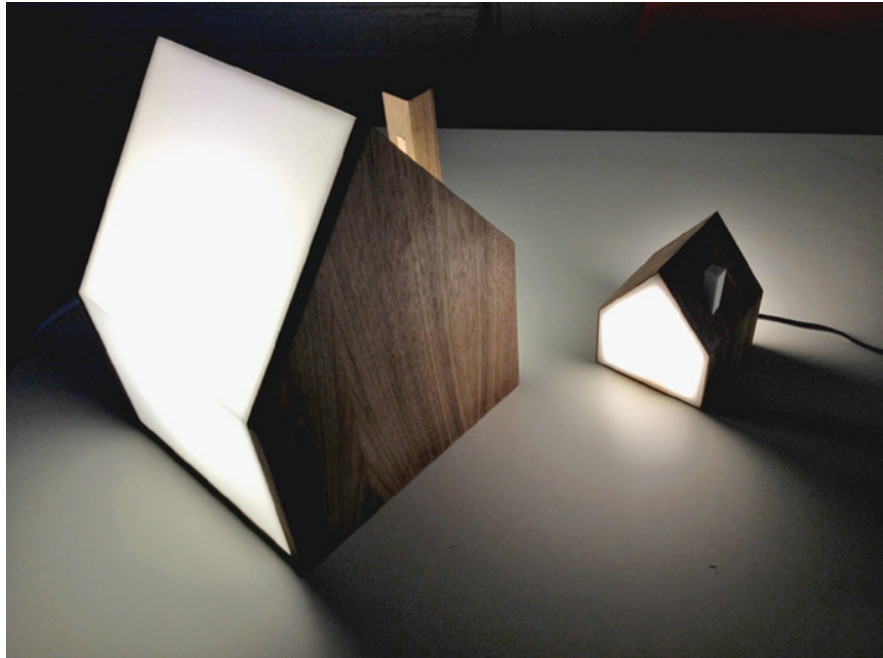
Use the button and LED projects from Chapter 6 to experiment with interactive triggers. You could make your own digital collection system using RFID or barcode scanners, covered later in Chapter 12, where you explore expansion boards and peripherals. Think of your installation as a conversation between the visitor and the system.

## Further reading

You can find much more information as well as illustrations on the Kin project page at [kin-design.com/commissioned-work/arts\\_culture/compass-card-system-national-maritime-museum/](http://kin-design.com/commissioned-work/arts_culture/compass-card-system-national-maritime-museum/).

# The Good Night Lamp

*The Good Night Lamp* is a family of connected lamps that let loved ones communicate a simple message: “I’m home.” Turning on the larger “Big Lamp” causes all linked “Little Lamps” elsewhere in the world to light up, too. (See Figure 8-6.) Designed by Alexandra Deschamps-Sonsino and her team, it became one of the earliest consumer IoT products based on Arduino.



**FIGURE 8-6:** Whenever the Big Lamp is turned on, the Little Lamp turns on as well, wherever it is in the world.

*Courtesy of Good Night Lamp*

## How it works

Each lamp used a Wi-Fi-enabled Arduino connected to a web server. When the Big Lamp was switched on, it sent its ID and on/off state to the server. The Little Lamps periodically checked the same server and mirrored the Big Lamp's state. The electronics were minimal — just a pushbutton, an LED driver, and a Wi-Fi shield — but the concept was emotionally powerful.

## Lessons learned

- » The best connected products often do one thing simply and beautifully.
- » Wi-Fi and cloud communication allow distant physical devices to stay in sync.
- » Emotional design can make technology more human.

## Build your own

Using the Physical Pixel or Graph example in Chapter 7, try linking an Arduino to software using a variable, such as a button press or sensor reading. You could

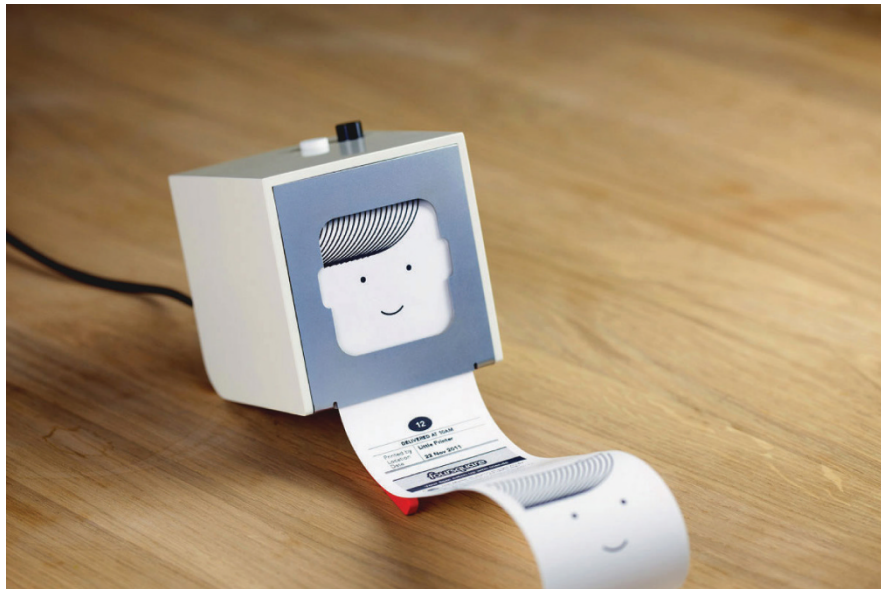
create a pair of connected nightlights or desk indicators that change color when one is pressed. This simple communication loop captures the spirit of *The Good Night Lamp*.

## Further reading

If you would like to read more about the Good Night Lamp, go to the product home page at [goodnightlamp.com](http://goodnightlamp.com).

# Little Printer

Berg, a London design consultancy, created Little Printer as a playful home companion. Using your smartphone, you could subscribe to content feeds — weather, puzzles, to-do lists, or messages from friends — which printed out as a tiny, personalized newspaper. (See Figure 8-7.) It showed how the Internet of Things could be charming and social rather than purely functional.



**FIGURE 8-7:**  
A Little Printer  
ready to print  
whatever data  
you'd like.

*Courtesy of Berg*

## How it works

The device used a small thermal printer (like those in receipt machines) connected to an Arduino-based prototype. Data was transmitted through Wi-Fi to a small bridge device, which handled the cloud communication. The early prototypes used XBee wireless modules before moving to custom hardware. Berg also built “Berg Cloud,” a backend platform that delivered the data.

The project started life as an Arduino proof of concept and grew into a full-fledged consumer product — one of the first to merge physical interaction with online data in a friendly, approachable way.

## Lessons learned

- » Arduino is ideal for testing and refining ideas before investing in custom hardware.
- » Physical output can make digital data more meaningful.
- » Even serious technology can have a playful personality.

## Build your own

In Chapter 13, you explore code and data visualization. Swap a display for a small thermal printer (available from Adafruit or Pimoroni) and create a mini news ticker or weather printer. The same code structure applies: gather data, format it, and send it to a visual output.

## Further reading

Sadly, Little Printer is no more, but it remains a textbook example of what’s possible with Arduino. To find out more about their journey, head over to [littleprinterblog.tumblr.com](http://littleprinterblog.tumblr.com).

# What These Projects Teach You

Across these examples, a few clear themes emerge:

- » **Tangibility matters.** Physical interaction makes digital experiences easier to understand and more rewarding to use.

- » **Connectivity adds value.** Projects become richer when they communicate — between devices, with users, or across the Internet.
- » **Design is as important as code.** A well-made enclosure or interface can make a simple circuit feel like a finished product.
- » **Iteration leads to innovation.** All these projects began as prototypes. Their creators learned through building, testing, and refining.

## Your Turn

You can apply these lessons using projects from earlier chapters:

- » Combine **motion and light** from the motor and LED examples (Chapters 10 and 12) to create your own kinetic sculpture like Chorus.
- » Use **sensor data and wireless modules** (Chapters 9 and 14) to make your own wearable or environment monitor inspired by Push Snowboarding.
- » Create **connected home objects** using buttons, LEDs, and cloud communication (Chapters 6 and 14), in the spirit of Baker Tweet or Good Night Lamp.
- » Build **interactive displays** (Chapter 13) that show or print live data, echoing Little Printer or Compass Lounge.

The important thing isn't to copy these projects exactly but to understand how they combine the same fundamentals you've already learned.

## Where to Go Next

If these projects spark ideas, you'll find thousands more on the Arduino Project Hub ([create.arduino.cc/projecthub](http://create.arduino.cc/projecthub)), which hosts tutorials from beginners and professionals alike. You can also explore Hackster.io ([hackster.io](http://hackster.io)), Instructables ([instructables.com](http://instructables.com)), or local maker communities to share your own builds.

Each of the projects in this chapter started with an idea, a few components, and curiosity — the same starting point you have now. Whether you're making something playful, practical, or poetic, Arduino gives you the tools to bring it to life.

- » Learning all about soldering
- » Getting all the right kits for the job
- » Assembling a shield
- » Moving from the breadboard to strip board
- » Preparing your project for the real world

# Chapter 9

# Soldering On

In previous chapters, I cover in great detail how to assemble circuits on a breadboard. If you read those chapters, you most likely already have a few ideas that build on or combine a few of the basic examples, so you may be asking, “What do I do next?”

This chapter takes you through the process, or art, of soldering. You discover all the tools you need to get your project ready for the real world. No more precariously balanced breadboards or flailing wires. From this point on, you’ll know what you need to solder circuit boards that last.

## Understanding Soldering

*Soldering* is a technique for joining metals. By melting metal with a much lower melting point than the metal you’re joining, you can link pieces of metal to form your circuit. Mechanical joints are great for prototyping, allowing you to change your mind and quickly change your circuit, but after you’re sure of what you’re making, it’s time to commit.

You use a soldering iron or solder gun to melt *solder*, a metal alloy (mixture of metals) with a low melting point, and apply it to the joint. When the solder has

cooled around the pieces that are being connected, it forms a secure chemical bond rather than a mechanical bond. This method is a far superior way to fix components in place, and bonded areas can still be melted and resoldered, if needed.

But why do you need to mess with soldering at all? Picture this: You have your circuit on a breadboard and you're ready to use it, but every time you do, the wires fall out. You could persevere and keep replacing the wires, but you risk replacing the wrong wire and damaging the Arduino or yourself. The best solution is to make a soldered circuit board that's robust and can survive in the real world.

A solderless breadboard allows you to quickly and easily build and change your circuit, but after you know that it works, you need to solder the circuit to keep it intact.

Creating your own circuit board is also an opportunity to refine your circuit by making circuit boards that fit the components. After you know what you want to do, the process of miniaturization can start and you're eventually left with a circuit board that takes up only the required space and no more.

## Gathering What You Need for Soldering

Before you dive in to soldering, make sure that you have what you need to get the job done. Read on to find out more.

### Creating a workspace

For your soldering adventures, what you need above all is a good workspace. Having a good workspace can make all the difference between a successful project and hours spent on your hands and knees, swearing at cracks in the floorboards. A large desk or workbench would be perfect, but even the kitchen table, if clear, will work. Because you're dealing with hot soldering irons and molten metal, it's a good idea to cover the surface with something you don't mind damaging. A cutting mat, piece of wood, or piece of cardboard will do fine for this purpose.

Your workspace should be well lit as well. Make sure that you have ample daylight by day and a good work light at night to help find those tiny components.

It's also good to have easy access to a power source. If your soldering iron functions at a fixed temperature and with a short lead connected directly to a plug, it can be especially important to have a plug nearby. If you overstretch your lead,

you run the risk of pulling the iron off the table and burning anything it touches. A tabletop power strip or multi-plug is the best solution because it provides power for your laptop, your lamp, and your soldering iron.

A comfortable chair is always important. Also remember to stand up every half hour or so to prevent back cramp. You can easily get drawn into soldering and forget what a horrible posture you're in.

Solder fumes, although not lethal, are not good for your lungs, so make every attempt to avoid breathing them. Always work in a well-ventilated area. It's also advisable to work with lead-free solder, as mentioned later in this section.



TIP

If you're working at home and are under pressure from other people in your house to not cover every surface with bits of metal, you should designate a soldering surface. This surface could be a rigid, wooden surface that can fit all your kit and can be moved, neatly packed away, or covered when not in use. This arrangement saves you the chore of unboxing and packing up every time you want to solder — and keeps everyone else happy as well.

## Choosing a soldering iron

The most important tool for soldering is, obviously, a soldering iron or solder station. You have a huge variety to choose from, but they're generally divided into four types: fixed temperature, portable, and temperature-controlled soldering irons, and complete solder stations. I describe each type in the following sections and provide a rough price from my local retailers. When you have an idea of what you want, shop around locally to see what deals you can find. If you're lucky, you may even find some high-quality second-hand gear on eBay!

### Fixed-temperature soldering iron

A *fixed-temperature soldering iron* (see an example in Figure 9-1) is normally sold as just an iron on a piece of electrical cable with a plug at the other end. They are usually sold with a sponge and a flimsy piece of bent metal as a stand. Others that are slightly better have a plastic stand with a place to put your sponge and a decent spring-like holster for your iron.

Fixed-temperature irons are adequate but offer no control over the temperature of your iron, other than on or off. They are sold with a power rating or wattage, which is of little help to most people. On these fixed-temperature irons, a higher wattage means a higher temperature, although that can vary wildly from manufacturer to manufacturer.



**FIGURE 9-1:**  
A basic fixed-temperature soldering iron.



**WARNING**

This variation can cause trouble with more delicate components because a high temperature is quickly conducted and melts integrated circuits and plastic components.

A quick study of RadioShack shows soldering irons in a range of powers, from 15W to 60W, which could cover a temperature range of approximately 400° F to 750° F. The difficulty is in finding an iron that is hot enough so that it heats the part that you want to heat quickly and allows you to melt the solder before the heat spreads. For this reason, a low-temperature iron can often do more damage than a higher temperature one. If you go too high, you encounter other problems, such as having the tips erode faster and running a higher risk of overheating parts.

If you want to go with a fixed-temperature iron, my advice is to start with a mid-range iron. I recommend a 25W iron as a good starter, which costs in the region of \$22 from RadioShack.

## USB-C soldering iron

A USB-C soldering iron (like the one shown in Figure 9-2) offers a modern, portable option for soldering without needing a wall socket. These irons are powered by a USB-C cable connected to a power source that supports USB Power Delivery (PD), such as a laptop charger or a high-capacity power bank. They're compact, heat up quickly, and are popular with hobbyists and professionals alike for field work and emergency repairs.

USB-C irons often include adjustable temperature control, usually via buttons or a small digital display. Some allow you to swap tips depending on the job. Despite their small size, many models heat to soldering temperatures in under 30 seconds and offer decent thermal stability — as long as you're using a compatible power supply.



WARNING

These irons are only as good as the power source you plug them into. If your USB-C power supply doesn't support high enough voltage (usually 9V or 12V) and sufficient current (at least 2–3 amps), the iron may underperform or not heat at all.

USB-C soldering irons are especially handy when you're working in tight spaces or traveling, and they're generally allowed in carry-on luggage — unlike gas-powered ones. You can expect to pay between \$30 and \$70 for a good model, plus another \$20–\$40 if you need to buy a compatible USB-C PD power supply or power bank.



**FIGURE 9-2:**  
A sleek and surprisingly powerful USB-C soldering iron. Yes, that tiny thing really works!

## Butane soldering iron

A *butane soldering iron* does away with cables and electricity in favor of gas power. It burns butane gas (more commonly known as lighter fuel) to produce its heat. A portable iron has a wattage rating to allow comparison with other irons, but unlike the fixed-temperature ones described in the preceding section, this rating indicates the maximum temperature, which can be lowered by use of a valve.



WARNING

The flames burn around the edge of the iron, which can make them awkward to use on precise joints, so I recommend using a portable iron only if necessary.

A portable iron (shown in Figure 9-3) is great for tight situations when an extension lead just won't reach, but it's a bit too expensive and wasteful to be used

often. It's also considered a lot more dangerous than conventional irons by most airport security because of the use of gas. So if you're planning on doing any soldering abroad, take an electric iron.



**FIGURE 9-3:**  
A butane soldering iron. Watch those eyebrows!

Butane soldering irons vary in price, but you can usually find them in the range of \$45 to \$90. You also need to buy additional butane refill cans for \$7.50 each.

## Temperature-controlled soldering iron

A *temperature-controlled soldering iron*, shown in Figure 9-4, is preferable to the fixed-temperature variety because it gives you more control at a reasonable price. This increase in control can make all the difference between melting and burning. Temperature-controlled irons should have a wattage rating, but it will be the maximum power possible. For this type of iron, a higher wattage is preferable because it should give you a better range of higher temperatures.



**FIGURE 9-4:**  
A temperature-controlled soldering iron.

A temperature control dial allows you to scale your temperature range up or down as needed. The difference between this type of temperature control and the control when using more accurate solder stations is that you don't have a reading of the current temperature. Most temperature-controlled irons have a color wheel indicating warm to hot, so a bit of trial and error may be necessary to get just the right temperature.

You can get an affordable temperature-controlled soldering iron for around \$30. Because you have more control over the temperature of the iron, you have greater control and more longevity than you have with a fixed-temperature iron.

## Solder stations

After you gain some experience (and can justify the expense), a solder station is what you'll want. A *solder station* is usually made up of an iron, a stand or cage for the iron, a temperature display, a dial or buttons for temperature adjustment, and a tray for a sponge. It can also include various other accessories for de-soldering or reworking, such as a hot air gun or vacuum pickup tool, but these are generally for more professional applications and not immediately necessary.

You can find many available brands of solder stations, but one of the most reputable and widely used is Weller (see Figure 9-5). Of that company's stations, I recommend the WES51 (120V AC), the WESD51 (120V AC), or the WESD51D (240V AC). My WES51 is still performing excellently after four years of use. Note that the 120V irons need a transformer to step the voltage down in countries that use 240V AC; the transformer can often be heavier than the iron itself!



**FIGURE 9-5:**  
A Weller  
solder station.

Before using a Weller solder station, I owned a cheaper temperature-controlled soldering iron that did very well for a number of years. As I cover in more detail later in this chapter, the best way to maintain and get the most out of your iron is to use it correctly. Leave a little melted solder on the tip to protect it when it's not in use. Then, before using the iron again, use the sponge to remove the solder.

## Soldering iron tips

Regardless of which iron you buy, I also recommend buying a few spare tips because they eventually degrade. You'll find a variety of tips for different purposes, too, so it's good to have a selection to cover different needs.

## Solder

Solder is what you use to join your circuits. Although many different types of solder are used with different combinations of metals, they commonly fall into two categories: lead and lead free. Many people prefer lead solder, finding it easier to work with, perhaps because it has a lower melting point, so your iron can be cooler and less likely to damage components.

Lead poisoning has been known about for a long time, and attitudes about using it have begun to change. Lead pipes were switched to copper as recently as the 1980s, and the use of lead solder in consumer electronics was addressed in 1996, when the Restriction of Hazardous Substances Directive (RoHS) and the European Union Waste Electrical and Electronic Equipment Directive (WEEE) addressed the use and disposal of certain materials in electronics.

You commonly find RoHS on components that comply with their guidelines, which should, therefore, be better for the environment when disposed of. Commercially, US and European companies enjoy tax benefits for using lead-free solder (shown in Figure 9-6), but lead is still widely used in the rest of the world. Steven Tai, a colleague of mine, visited China to complete a project we were working on. When he asked where he could buy lead-free solder, he was laughed at outright because lead-free solder was not only unheard of in most cases but not even available! For the more conscientious Arduinists, most electronics supplier and shops offer lead-free solder that contains other metals, such as tin, copper, silver, and zinc. From my experience, lead-free solder works just fine for any Arduino projects, so if you want to do your bit for the environment and avoid using lead in your work, please do!



**FIGURE 9-6:**  
Lead-free solder.

Another variety of solder is flux-cored solder. *Flux* is used to reduce oxidization, the reaction on the surface of a metal when it reacts with the air (oxygen), as with rust on an iron anchor. Reducing the oxidization allows a better, more reliable connection on your solder joint and allows the solder to flow more easily and fill the joint. Some solders have flux in the core of the solder, dispensing flux to the joint as the solder is melted. You sometimes see smoke as you melt your solder; in most cases, the smoke is from the flux burning off. You can be sure that you have flux core if, when you cut the solder, you see a black center surrounded by a tube of solder.

Always work in a well-ventilated area and avoid breathing the solder fumes no matter which solder you're using. Solder fumes are not good for you, and neither is eating solder. Always wash your hands and face thoroughly after soldering. Sometimes the flux in solder can spit, so wear clothes that are not precious — and definitely use eye protection.

## Third hand (helping hand)

Sometimes you just don't have enough hands to hold the fiddly electronics that you're trying to solder. It would be great to have someone nearby with asbestos hands to hold the tiny red-hot pieces of metal, but failing that, you can use a little device known as a *third hand* or a *helping hand*, as shown in Figure 9-7. A third hand is a set of crocodile clips on an adjustable arm. You can arrange the clips to help you get your component, circuit board, and solder into place.



**FIGURE 9-7:**  
Give the  
man a hand!

A third hand costs from \$5 to \$50 and can be extremely useful for holding a circuit board at an angle or holding components together while you work on the soldering. The drawback is that setting it up can be tricky. If you're doing lots of solder joints, you may spend a lot of time loosening, adjusting, and retightening. If you do purchase a third hand, make sure that the parts are all metal. Plastic parts, such as the grips on the vices, will not stand up to much use.

## Adhesive putty

A good alternative to a third hand is adhesive putty, such as Bostik's Blu-Tack, UHU's White Tack, or Loctite's Fun-Tak. Rather than use a mechanical device to grip a component or a circuit board, you use the adhesive putty to hold the objects you're soldering in place on one side of the board, leaving you free to work on the other side of the board without the components or circuit board moving. You can also use the adhesive putty to tack your work to your work surface, stopping it from moving around the surface as you solder. After the solder joints are done, you remove the adhesive putty, which you can reuse. Note that putty goes extremely soft if it is heated and takes a while to return to its usual tackiness. After it cools, you can roll the ball of putty along the circuit board to remove any remaining bits.

## Wire cutters

A good pair of wire cutters or snips is invaluable. Many pairs of wire cutters have a rounded, claw-like shape. These are tough but can be difficult to use when cutting in confined spaces or pinpointing a specific wire. Precision wire cutters have a more precise, pointed shape that is far more useful for the vast majority of electronics work.

Note that wire cutters are good for soft metal such as copper but do not stand up to tougher metals such as paper clips or staples. Figure 9-8 shows a pair of pointed wire cutters.



**FIGURE 9-8:**  
Pointed wire cutters are good for getting into and out of tight spots.

## Wire strippers

To connect wires to your project, you need to strip back the plastic insulation. You can do this stripping with a knife if you're careful or don't value your fingertips, but the quickest, easiest, and safest way to strip a wire is to use a wire stripper. The two kinds of wire strippers are manual and mechanical (see Figure 9-9). Manual wire strippers are like clippers but have semicircular notches made to various diameters. When the wire stripper is closed on wire, it cuts just deep enough to cut through the plastic sheath but stops before it hits the wire. Mechanical wire strippers work with a trigger action to remove the insulation on the wire without any need to pull on the wire.

Mechanical wire strippers are a great timesaver but can be less reliable in the long run because the mechanisms are more likely to fail than are those in the simple manual ones.

**FIGURE 9-9:**  
Manual and  
mechanical wire  
strippers.



## Needle-nose pliers

Needle-nose pliers, as with a solderless breadboard, are a great help for getting to those hard-to-reach places. They're especially useful when soldering because they spare your fingers from any excess heat. I cover needle-nose pliers in more detail in Chapter 4.

## Multimeter

A multimeter is a great help for testing your circuits. When you're soldering connections, the continuity-testing function can be a great help for verifying that the solder joints are as they should be and are connected to the right places. See Chapter 4 for more about working with a multimeter.



REMEMBER

When testing the continuity, always unplug any power supplies connected to your circuit to avoid false bleeps.

## Solder sucker

Everyone makes mistakes, and they can be more difficult to undo when you're dealing with hot metal. One tool to keep handy for fixing mistakes is a solder sucker, shown in Figure 9-10, or a de-soldering gun. Each of these tools is a pneumatic pump that sucks molten solder straight off the surface of a circuit board. Most have a piston that can be pressed down and will lock when all the air is pressed out. When you hit the trigger, a spring pushes the piston back out and sucks any molten solder into the piston chamber. Next time you press the piston

down, it pushes out any solder that was removed. Using this type of tool takes a bit of practice because you need to heat the solder with your soldering iron in one hand and suck it away with the sucker in the other.



**FIGURE 9-10:**  
A solder sucker  
can save  
your project.

## Solder wick

Another method of removing solder is to use solder wick (see Figure 9-11), also known as copper braid or de-soldering wire. Solder wick is a copper wire that has been braided, and you buy it in reels. It provides lots of surface area for solder to grip into, to remove it from other surfaces. Place the braid on the joint or hole that has too much solder, and hold your soldering iron on top of it to heat the solder wick. Apply some pressure, and the solder continues to melt and fill the gaps in between the braids of the copper wire. Remove the wick and the soldering iron together, and the solder should be cleared. If it's not, repeat as necessary. After the solder is cleared, you can cut off the used solder wick and dispose of it.



**FIGURE 9-11:**  
Solder wick is  
great for those  
more stubborn  
mistakes.



WARNING

Do not pull the solder wick away if the solder has cooled. You risk pulling off the metal pads on your board, making it unusable, because the solder wick is attached to the board.



TIP

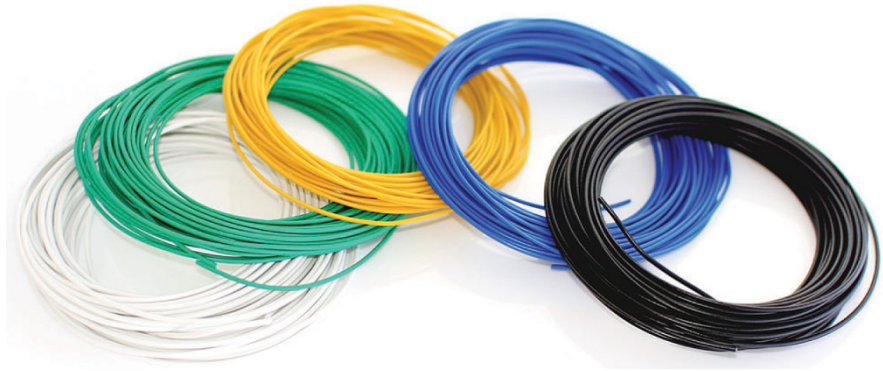
Solder suckers and solder wick are both equally effective at removing solder, but they are each suited to different situations and require an element of dexterity and skill to use. If you're worried about overheating components, a solder sucker is more suitable; if you can't remove all the solder with the sucker or can't get close enough to the components with it, the solder wick may be a better option. I advise getting both to prepare yourself for any situation.

## Equipment wire

Equipment wire is the general name given to electronics wire. It's the same as the jump wires you may have in your kit but is unfinished. You buy it in reels. Equipment wire can be either single-core or multicore. Single-core wire is made up of one solid piece of wire and is malleable, so it holds its shape if bent, but it snaps if it is bent too much. Multicore wire is made up of many fine wires and can withstand a great deal more flexing than single-core, but it does not keep its shape if bent. To use equipment wire, you need to cut it to length and strip the insulation off the ends of to reveal the wire underneath.

Wire also comes in different diameters, indicated with numbers such as 7/0.2 or 1/0.6. In this format, the first digit is the number of wires in the bundle and the second is the diameter of those individual wires. So 7/0.2 is a bundle of 7 wires, each measuring 0.2mm, making it multicore; 1/0.6 is one single-core wire with a diameter of 0.6mm.

When you're starting out, it can be difficult to know what type of wire to invest in. Wire is cheaper when bought by the reel, but you don't want to be stuck with wire that isn't fit for your purposes. As a general guideline, I have found that multicore wire is the most versatile and robust for most applications. The 7/0.2 diameter should fit most PCB holes. I also recommend having three colors — red, black, and a color to signify your signal wires. With three reels of this type and size, you should be able to complete most projects. Some hobby electronic shops also supply lengths in various colors, such as the ones by Oomlout shown in Figure 9-12.



**FIGURE 9-12:**  
Equipment wire  
in various colors.

## Staying Safe while Soldering

With a few simple precautions, you can solder safely. Remember that soldering is not dangerous if you take proper care — but it can be if you don't. Please keep the tips in this section in mind whenever you're soldering.

### Handling your soldering iron

A soldering iron is safe if used correctly but is still potentially dangerous. The iron has two ends, the hot end and the handle. Don't hold it by the hot end! The correct way to hold a soldering iron is like a pen, between your thumb and index finger, resting on your middle finger. When you're not using the iron, keep it in its holster or cage, which helps to dissipate heat and prevent accidental burns.

### Keeping your eyes protected

You must wear proper eye protection when soldering. Solder, especially the flux-cored kind, has a tendency to spit when heated. In addition, when you use clippers to neaten your circuit board, the small bits of metal you cut off often shoot around the room if they're not held down. Also, if you're working in a group, you need to protect yourself from the person next to you. Safety goggles are relatively inexpensive depending on the amount of comfort you want, and they're a lot cheaper than eye surgery.

## Working in a ventilated environment

Breathing in fumes of any kind is bad for you, so it's important to always solder in a well-ventilated environment. Also make sure that you're not working under any smoke alarms because the fumes from soldering can set them off.

## Cleaning your iron

Your soldering iron should come with a sponge, which you use to wipe away excess solder. You should dampen the sponge but not use it soaking wet, so make sure to squeeze out excess water. When you're heating solder, it oxidizes on the tip of the iron. If the tip itself oxidizes, it can degrade over time. To prevent oxidizing the tip, leave a blob of solder on the end of the iron while it's in its cage. Doing so makes the blob of solder oxidize rather than the tip, and you just wipe off it off using the sponge the next time you need the iron.

## Don't eat the solder!

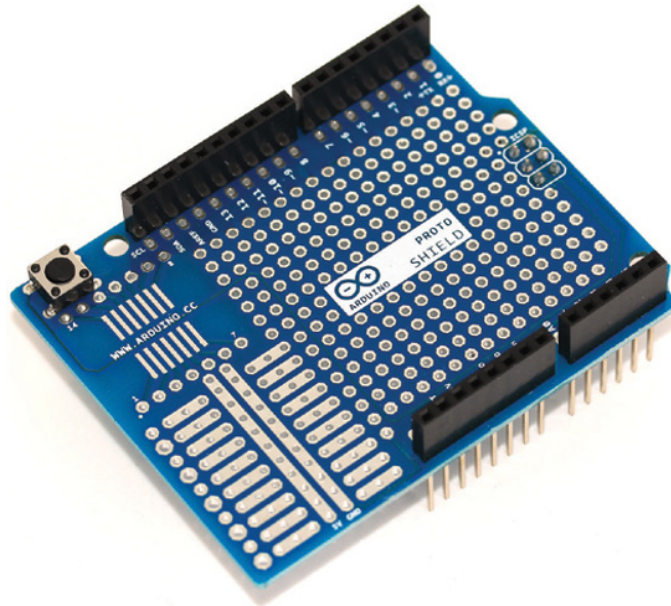
Although the chemicals and metals in solder are not deadly, they are definitely not healthy and can cause irritation. While soldering, avoid touching your face and getting solder around your eyes and mouth. It's also a good idea to wash your hands (and face if needed) after soldering.

# Assembling a Shield

Soldering requires learning a great amount of technique, and you develop good technique with practice. In this example, you find out how to assemble an Arduino shield. A *shield* is a specific printed circuit board (PCB) that sits on top of the Arduino to give it a function. (You learn more about shields in Chapter 12.)

There are different shields for different functions. The one used in the example is the proto shield kit (shown assembled in Figure 9-13), which is essentially a blank canvas to solder your project onto, after prototyping it on a breadboard. In this example, you see how to assemble the bare minimum of the kit to attach it to your Arduino and then how to build a simple circuit on it.

As with many Arduino kits, you need to assemble this shield yourself. The basic principles of soldering remain the same but may vary in difficulty as you encounter smaller or more sensitive components.



**FIGURE 9-13:**  
A complete  
proto shield.

## Laying out all the pieces of the circuit

When assembling a circuit, your first step should always be to lay out all the pieces to check that you have everything you need. Your work surface should be clear and have a solid-colored cover to make things easy to find.

### USING STRIPBOARD RATHER THAN A PCB

Specially designed shields are made to fit your Arduino perfectly but can often be relatively expensive. Stripboard, or perfbboard as it's sometimes known, provides a cheap and highly versatile alternative. *Stripboard* is a circuit board with strips of copper and a grid of perforated holes that you can use to lay out your circuit in a similar way as on a breadboard. An example of stripboard appears in the following figure.

The pitch of the holes and the layout of the copper strips can vary. The most useful pitch for Arduino-related applications is the same as the pitch on the Arduino pins, 0.1 inches (2.54mm), because that pitch allows you to build on the layout of your Arduino

*(continued)*

(continued)

to make your own custom shields. You can buy stripboard in various arrangements of copper strip as well, commonly either long copper columns that run the length of the board or sets of columns three rows deep (usually called tri-board).

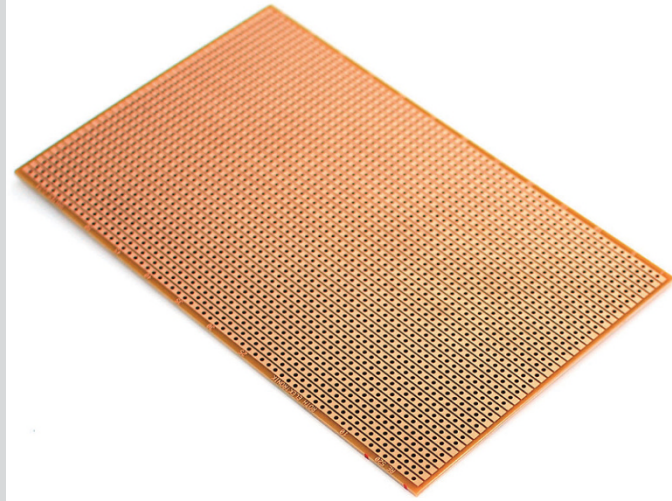
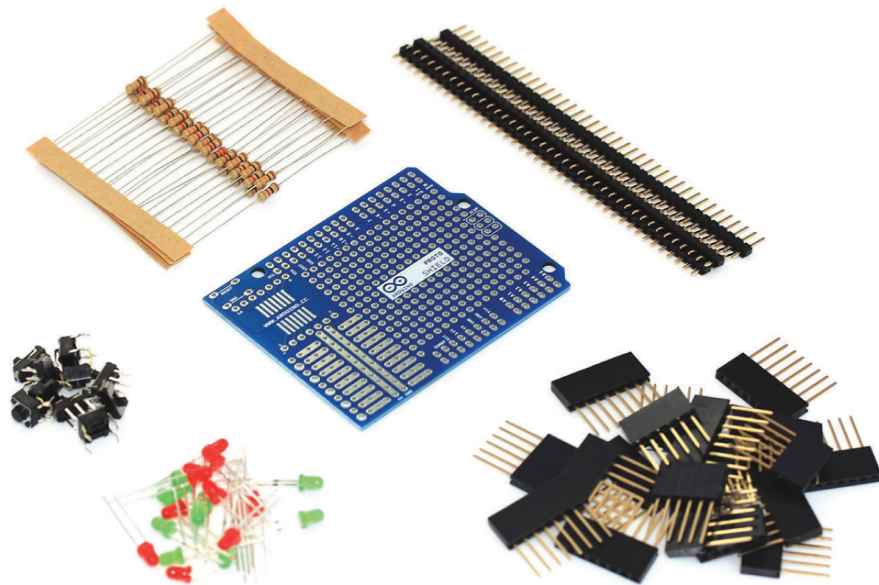


Figure 9-14 shows the Arduino proto kit laid out in an orderly fashion. It contains the following:

- » Header pins (40 × 1)
- » Header sockets (3 × 2)
- » Pushbuttons
- » LEDs (various)
- » Resistors (various)

Some kits may ship the PCB only and leave you to choose the headers that are connected. Remember that there is no right or wrong way as long as the assembly suits your purpose.

To assemble this shield, you can work from a picture to see the layout of the components, but for more difficult ones, you usually have instructions. In this example, I walk you through the construction of this shield step by step and point out various techniques for soldering along the way.



**FIGURE 9-14:**  
All the parts  
of the shield  
laid out.

## Assembly

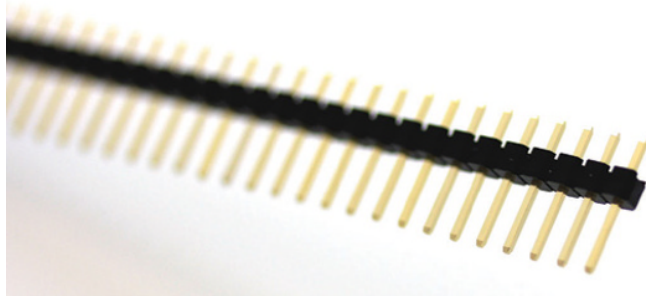
To assemble this kit, you need to solder the header pins and the pushbutton. Soldering these pieces allows the shield to sit in the header sockets on your Arduino, extending all the pin connections to the proto shield. Note that some versions of the proto board have header sockets (or stackable headers) rather than header pins. Header sockets have long legs so that they can sit on top of an Arduino in the same way as header pins but also allow sockets for another shield to be placed on top. The benefit of header pins is that your shield is shorter and needs less space for any enclosure. Header sockets are used in the assembled shield shown in Figure 9-14. They are the black sockets that run down either side on the top of the board, with pins extending underneath.

In this example, I use header pins and do not connect the ICSP (in-circuit serial programming) connector, which is the  $3 \times 2$  connector in the center right of the Uno (refer to the first figure in Chapter 2). The ICSP is used as an alternative for uploading sketches with an external programmer as opposed to the Arduino and is for advanced users.

## Header pins

First you will need to cut the header pins to length (see Figure 9-15). This kit uses a length of  $1 \times 40$ , which is 1 row of 40 pins. The plastic strip has a notch between each pair of pins that you can cut to divide the pins neatly. To secure the shield,

you need lengths of 6 pins (for the analog-in pins), 8 pins (for the power pins), 8 pins (for the shorter row of digital pins), and 10 pins (for the longer row of digital pins). Use your clippers to cut the header pins to the correct length; you should have 8 pins remaining. (Put these in a box for future use!) The pins should fit exactly because there is a 2.54mm (0.1 inch) pitch between them, which matches the board. You need to look for this same pitch if you're buying header pins of any other connectors separately.



**FIGURE 9-15:**  
Close up of the  
header pins.

Now that you know where the header pins go, you can solder them in place. In the next section, I talk you through soldering technique. Have a read through before you start.

## Acquiring Your Soldering Technique

The first step with soldering is to make sure that your components are secure. It's common to try balancing your circuit board on whatever objects are closest to hand to arrange it in an accessible position. But if you do, the board is destined to fall over at some point, most likely when you have a hot soldering iron in your hand.

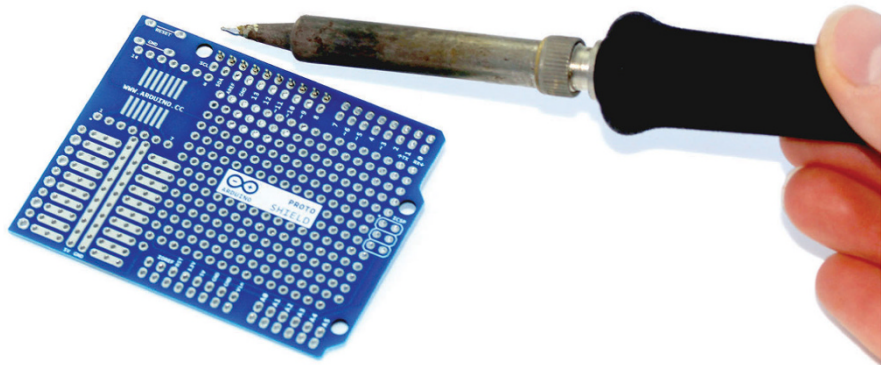
As mentioned in the “Gathering What You Need for Soldering” section, earlier in this chapter, two good ways to secure your work are to use a third hand or adhesive putty. You can use the crocodile clips on the third hand to grip the plastic of the pins and the circuit board as well, holding them firmly at 90 degrees from one another. Similarly, you can hold the pins at 90 degrees and press adhesive putty into the underside to hold them together. You can then press them into a rigid, weighty support to bring the board closer to you. I sometimes use a big reel of solder for the support.



WARNING

A third way to secure your components might occur to you, and because it's tempting, I describe it here so that you won't do it! You might think that the perfect way to lay out the pins would be to place them in the Arduino itself (with the long ends sticking into the sockets) and then place the circuit board on top. This approach holds everything at 90 degrees; however, because the pins are designed to connect to the Arduino and conduct electrical current, they can also conduct other things, such as heat. If they conduct the heat of your soldering iron, that heat can be passed through the board to the legs of the very sensitive microcontroller chip and damage it irreparably.

When you have your circuit board supported and ready, you can rotate it to a comfortable working angle, most likely the same angle as your iron, as shown in Figure 9-16.



**FIGURE 9-16:**  
Always arrange your work in a comfortable position.

Fire up your soldering iron. My Weller WES51 has a temperature range of 35–85 and the units show this as  $^{\circ}\text{F} \times 10$ , so the real range is 350–850  $^{\circ}\text{F}$ ! The hotter you set it, the quicker it will melt solder to make joints, but it will also melt everything else faster, such as plastic parts and silicon chips. Always set your iron to the lowest convenient temperature. A good way to test the temperature of the iron is to melt some solder. If it's taking a long time to melt, make sure that you have a good amount of the surface area of the soldering iron's tip in contact with the solder. If it's still not working, increase the temperature gradually, allowing time for it to get up to temperature. I normally set it to 650  $^{\circ}\text{F}$  (340  $^{\circ}\text{C}$ ), which is hot enough but not too hot.

Some quality soldering irons have a thermostat that tells you when the iron is up to temperature. The really fancy ones have a digital display. With the cheaper ones, you have to use your best judgment.

While your iron is getting up to temperature, you can wet your sponge. On some irons, the sponge is stuck down for convenience, but it's particularly inconvenient when you need to wet it. I recommend unsticking it and taking it over to a basin rather than spilling water over your soldering iron and the surrounding area. The sponge should be damp but not full of water. The dampness stops the sponge from burning when you pass your iron across it. If the sponge is too wet, however, it can lower the temperature of the iron and the solder, meaning that the solder hardens or solidifies and can't be removed until it's up to temperature again.

Now you're ready you can start soldering. Follow these steps:

**1. Melt a small amount of solder on the tip of your iron (called *tinning the tip*); see Figure 9-17.**

The solder should stick to the edge and smoke. Generally, with a new or well-maintained iron, the solder latches onto the tip with no problem. If it doesn't stick to the edge of the tip, you may need to try rotating it to find a good patch that isn't oxidized. Failing that, you can use some tip cleaner to remove any built-up layer. By pressing your hot iron into the tip cleaner and wiping away any buildup as it loosens, you can restore your iron to its former glory.

Tip cleaner is generally a nasty, toxic substance. Make sure you do not ingest or inhale any of it.



WARNING

**2. When you have a blob of solder on your iron, wipe it off on a sponge to reveal a bright metallic tip on your iron.**

The aim is to apply this freshly tinned edge (not the point) to the area that you're joining. Using the edge gives you more surface area and allows the joint to heat up faster. It's also important to note that a variety of tips are available for soldering irons, and if you choose good ones, they are easily interchangeable, allowing you to suit them to different situations. Some are pointed, as in Figure 9-17; others are screwdriver tips, chisel tips, and beveled tips to provide different amounts of surface area for different situations.

**3. Starting on the first pin in one of the rows, apply the iron to the metal plate on the circuit board and the pin connecting to it.**

This step heats the board and pin, preparing them for the solder.

**4. With your other hand (the one not holding the iron), apply the solder to the point where the iron, the pin, and the metal plate all meet.**

As the solder is heated, it melts and spreads to fill the gap, sealing the pin and the board together. To apply more solder, press it into the joint where it is melting; to apply less, simply pull it back out. You may need only a few millimeters of solder for small joints.

- 5. When the solder has filled the area, remove it, but keep your iron there for a second or two longer.**

This extra time allows the solder to melt fully and fill any gaps for a solid joint.

- 6. Remove your soldering iron by wiping it up the leg of the pin.**

Any excess solder is directed upward into a point that can be cut off rather than sitting in a blob.

**FIGURE 9-17:**  
Tinning the tip helps to preserve your iron and makes soldering easier.



This entire process should take around 2–3 seconds. That probably sounds impossible, but after you get used to the rhythm, it's totally achievable. All it requires is practice.

Following the preceding steps should leave you with a neat, pyramid-shaped solder joint, with the metal pad on the circuit board completely covered. If you can see the hole that the pin has come through or have a blob of solder on the pin that is not connected to the circuit board, you need to reapply heat using the soldering iron, and maybe reapply a little solder as well.

After you solder the first pin, it's a good idea to do the pin at the other end of the row. By doing these two, you secure the row in place, and if it is not level or at 90 degrees, you can still straighten the row by heating up the solder at either end. If you find that you have too much solder, first try reapplying heat. Watch as the solder melts, and you should see it filling all the gaps. If you still have too much solder, you can use a solder sucker or copper braid (described in “Gathering What You Need to Solder,” earlier in this chapter) to remove any excess and try again.

For examples of well-soldered joints versus bad ones, see the image at [learn.sparkfun.com/tutorials/how-to-solder-through-hole-soldering#soldering-your-first-component-](http://learn.sparkfun.com/tutorials/how-to-solder-through-hole-soldering#soldering-your-first-component-).

Place the shield on top of your Arduino and check that it fits correctly. If it does, you're ready to solder your first circuit, which you find out about next.

# Building Your Circuit

After you have a shield to work on (see “Assembling a Shield,” earlier in this chapter), you can think about what circuit you want to build on it. Before you actually build that circuit, though, you should do some proper planning to avoid having to undo what you’ve done, which can be difficult. The best way to start is by prototyping the circuit on a solder-less breadboard. This process is quick, easy, and — more importantly — not permanent. As covered in Chapters 6 and 7, throwing together a circuit and making sure that it works are simple tasks. For this example, I use the AnalogInOutSerial example (see Chapter 6) to demonstrate how you can transform a solderless breadboard into a soldered one.

## Knowing your circuit

First, recreate the AnalogInOutSerial circuit on the breadboard as shown at the end of Chapter 6. Upload the sketch by choosing File ⇨ Examples ⇨ 03.Analog ⇨ AnalogInOutSerial. This should give you an LED that can be faded by twisting the potentiometer.

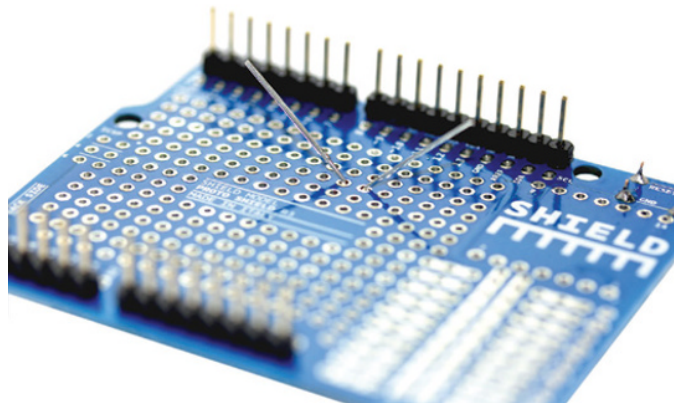
When you have this circuit working, take a look at the AnalogInOutSerial circuit again on the breadboard. One difference that you can immediately see is that the proto shield doesn’t have the same rows and columns as the breadboard, except in one corner. This corner area is designed for ICs (integrated circuits) or chips but can be used for anything. The rest of the proto shield has individual holes into which components can be soldered. You can connect these with wires to join the various components.

The easiest way to convert the circuit to one that will work on the shield is to look at the circuit diagram. The lines that connect the components can be substituted for real wires and soldered directly to the correct pins. For the potentiometer, you need three wires: 5V, GND, and analog 0. For the LED and resistor, you need two more: GND and pin 9. It’s always a good idea to draw the circuit first for clarity. It doesn’t need to be as neat as the circuit diagram in the examples, but planning your circuit will save a lot of painful de-soldering later. As the old carpentry saying goes, *measure twice, cut once* (or in this case, *sketch twice, solder once*).

Note that the wires all go to holes next to the wire with which they need to connect. Because you don’t have enough space to comfortably fit the wires and the component legs in one hole, you need to get them close and then bridge the gap by using the ends of the wire or the legs of the components.

## Laying out your circuit

Now that you've drawn the circuit, you should lay it out so that you know what length to cut your wires. To secure the components to the board, insert them and bend to 45 degrees the parts of the legs that protrude through the board. They should look something like that shown in Figure 9-18.



**FIGURE 9-18:**  
An LED secured in  
a circuit board,  
seen from below.

## Preparing your wire

You can see that the lengths of wire required are relatively short. If you have solid-core wire, you can bend it neatly into shape to sit flat on the circuit board by using needle-nose pliers. If you have multicore wire, making an arc of wire up out of one hole and into the next is easier. Remember to measure or estimate the distance and add a small amount on each end to fit into the hole before you cut your lengths. Strip back the wire and sit it in place to make sure that the length is correct. Note that sometimes the end of multicore wire can become frayed. If this happens, grip it between your thumb and forefinger and twist the end into a point. If you lightly coat the end in solder, you can prevent the wire from unravelling and tin it in preparation for soldering. When you're happy with the length, place the wires to one side.

## Soldering your circuit

Now that you have all the components and wires, it's time to solder them in place. Earlier, you bent the component leg at 45 degrees, so the wires should still be hanging on. Because the resistor and LED are next to each other, you can use their legs to connect them, avoiding the use of another wire. Remember to check that your LED is the right way around. If required, use adhesive putty to further secure the components in place and then solder them as with the pin headers.

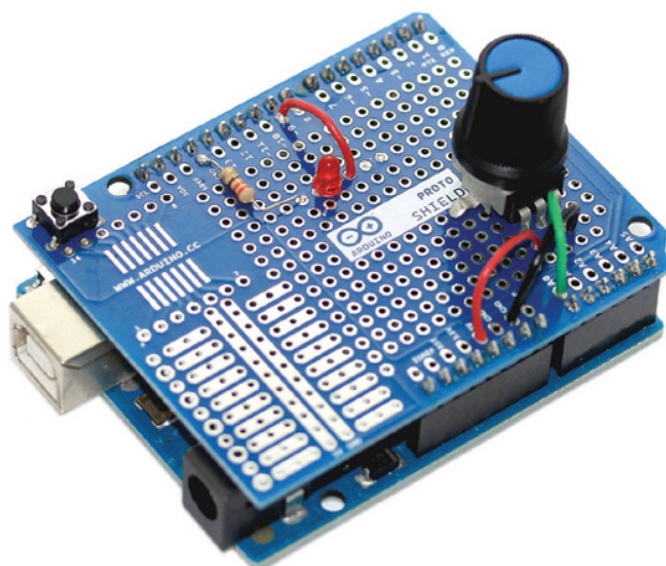
When all the components are in place, you can solder the wire. Insert the wire and bend to 45 degrees as before. When it is secured, solder it in place or bend it further to meet the component legs, bridging the gap as with the resistor and LED. As always, there is no right or wrong in the appearance of your result; it depends how neat you want to be. Some people prefer to wind the wire around the legs of the component to get a good grip; others prefer to join them side by side to get a clean joint. The choice is yours. As long as you have a good connection, you're in good shape.

## Cleaning up

When you finish soldering, give the board a good check for any loose connections. If all seems well, you can start neatening the board. Using your clippers, carefully remove the legs of the components from just above the solder joint, at the top of the pyramid. You can cut lower, but the solder is thicker and you risk tearing the metal contacts off the circuit board, which cannot be fixed. Remember, always hold or cover the piece of metal that you're clipping. If you don't, it can fly a great distance and seriously injure someone.

## Testing your shield

Now that you have your shield with a completed circuit assembled on it, it's time to plug it in and try it. If everything is working correctly, you should have a neatly packaged dimmer circuit shield resembling something like Figure 9-19.



**FIGURE 9-19:**  
My new dimmer shield, ready to be taken out into the world!

# Packaging Your Project

Now that your circuit is no longer at risk of falling apart, it's a good idea to look at protecting it from the outside world by boxing it up.

## Enclosures

The simplest way to protect your circuit is by putting it in a box. In electronics terms, such a box is called an enclosure or a project box. You can find a variety of plastic or metal enclosures in a vast array of shapes, sizes, and finishes. The only task is to find the one that's right for you.

Many of the online suppliers (RS, Farnell, Digi-Key, and others) have huge lists of possible enclosures, but you can't tell whether it's right without holding it in your hands. Many enclosures have accurate measurements for internal and external dimensions, but even with that information there are usually omissions, such as the molded plastic for the screw fixings. My advice is to find a retail store, such as Radio Shack, and take an Arduino with you to see whether it will fit correctly with enough space for wires and any shields. Then you'll know what works and what to order next time.

Keep the following considerations in mind when boxing up your project.

- » **The ability to access the USB for code changes:** You may have to unscrew your enclosure to update the code. If this process is too time-consuming, you may need to drill a hole big enough to plug the USB in from the outside.
- » **Power to run the Arduino:** If your Arduino is not powered by USB, how is it powered? It could be an external power supply that plugs into the power jack, which needs a big enough hole for the plug. You could remove the plug and solder the bare wires to the VIN and GND pins if you're looking for something more permanent. Or you could even run it off a battery pack and only open it every week or so to charge it.
- » **The ability to access the inputs and outputs:** What use is an LED or a button if it's inside the box? Most projects need some contact with the outside world, if only to tell you that they're still working, and most components are designed with this need in mind. The lip on an LED means that you can drill a 1.9-inch (5 mm) hole and push the front end through without it going all the way; if you take the plastic or metal knob off a radio, you'll see a similar fitting too.

Always think carefully about the needs of your circuit before soldering it in place and boxing it up. Examine other cheap electronics around you to see what tricks the industry has been using. For example, you may be surprised to find out that most remote controls for remote control cars that give you forward, backward, left, and right are just a few simple pushbuttons underneath those complex-looking control sticks.

## Wiring

To give you more flexibility with your wiring, consider attaching inputs, outputs, or power to flexible lengths of cable by using terminal blocks, which are sometimes also known as connector strips, screw terminals, or chocolate (choc) blocks. By doing this, you can fix the inputs and outputs to the enclosure rather than to the circuit board. This approach gives you more flexibility — and if you drill a hole slightly out of alignment, you won't need to de-solder or re-solder your circuit.

Following is a little more detail on adding wires to your project and selecting terminal blocks, which adds flexibility to your project and makes assembling and disassembling your project a lot easier.

### Twisting and braiding

When you make wire connectors, I recommend twisting or braiding the wires together. The braiding gives the wires extra strength if pulled on and, as a bonus, looks nice! To twist two wires together, cut them to length and grip one end in a power drill. Hold onto the other end of the two wires with your hand or a vice. Pull the wire taut enough to have no slack but not too tight. Spin the drill until the wires are neatly twisted together, and you should be left with one twisted wire.

If you have three wires, you can braid them in the same way that you braid hair. With three wires in your hand, hold them at one end facing forward. Pass the leftmost wire over the middle and under the rightmost. Keep repeating this process with the new left wire until you have braided the full length. Your project will be more robust and look extra professional. I advise you to use different colors for the wires; if you use the same color wire for all three, you need to use the continuity tester on your multimeter to figure out which is which.

### Terminal blocks

Terminal blocks come in a variety of sizes, depending on the amount of current passing through them, and are usually marked with the upper limit that they can maintain. When selecting one for your power supply, always read the current

rating and choose a size with a bit of tolerance. If you have a 3A supply, choose a 5A terminal block. When connecting wires, especially multicore wires, you should tin the tip of the wire with a small amount of solder or fold the wire back under the insulation layer so that the screw grips the insulated side. This technique prevents the screw from cutting any of the strands of wire as it is tightened.

## **Securing the board and other elements**

When you're happy with your cabling and have all the required holes, it's a good idea to secure your items so that they don't rattle around inside the box. To secure your Arduino, screw terminals, or stripboard, you can use Velcro-type tape or hot glue. If you have any loose wires, you can use cable ties to neatly tie them together.



- » Understanding timers
- » Debouncing your buttons
- » Getting more from your buttons
- » Averaging your results
- » Adjusting the sensitivity of sensors

## Chapter **10**

# Getting Clever with Code

**A**s you find different uses and needs for Arduino, you can refine your code to make it more accurate, responsive, and efficient. Also, by thinking about the code in your project, you may be able to avoid or minimize many of the unexpected results that can occur when dealing with physical hardware and the real world. In this chapter, you look at a few sketches that will help you fine-tune your project.

## Blinking Better

Blink is most likely the first sketch you encountered. It's a magical moment when that first LED lights up, isn't it? But what if I told you that it can get even better? The basic Blink sketch presented in Chapter 3 performs its task well, with one significant drawback: It can't do anything else while blinking.

Take a look at the Blink sketch again:

```
void setup() {           // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {           // the loop function runs over and over again forever
  digitalWrite(LED_BUILTIN, HIGH); // turn on LED (HIGH is the voltage level)
```

```
delay(1000);           // wait for a second
digitalWrite(LED_BUILTIN, LOW); // turn off LED by making the voltage LOW
delay(1000);           // wait for a second
}
```

The loops can be summarized this way:

1. Turn on the LED.
2. Wait for a second.
3. Turn off the LED.
4. Wait for a second.

This delay, or waiting, can be problematic for many people when they try to integrate the Blink sketch with another bit of code. When the sketch uses the delay function, it waits for the amount of time specified (in this case, a second), during which it doesn't do anything else. Effectively, the sketch is twiddling its thumbs.

If you wanted to change something — for example, you wanted the LED to blink only when a light sensor was dark — you might think of writing the code in the loop section something like this:

```
void loop() {

  sensorValue = analogRead(sensorPin);

  if (sensorValue < darkValue) {
    digitalWrite(LED_BUILTIN, HIGH); // turn on LED (HIGH is the voltage level)
    delay(1000);                       // wait for a second
    digitalWrite(LED_BUILTIN, LOW);  // turn off LED by making the voltage LOW
    delay(1000);                       // wait for a second
  }
}
```

This code almost works. When the value threshold for `darkValue` is crossed, the `if` loop starts and turns on the LED, waits for one second, and then turns it off and waits for one second. But because the sketch is occupied doing this blink for two seconds, the sketch can't check to see if the light level becomes brighter again until the blink has finished.

The solution is to use a timer rather than pause the program. A timer or counter is like a clock that can be used to time events. For example, the timer can count from 0 to 1,000, and when it reaches 1,000 it can *do something*, and then start

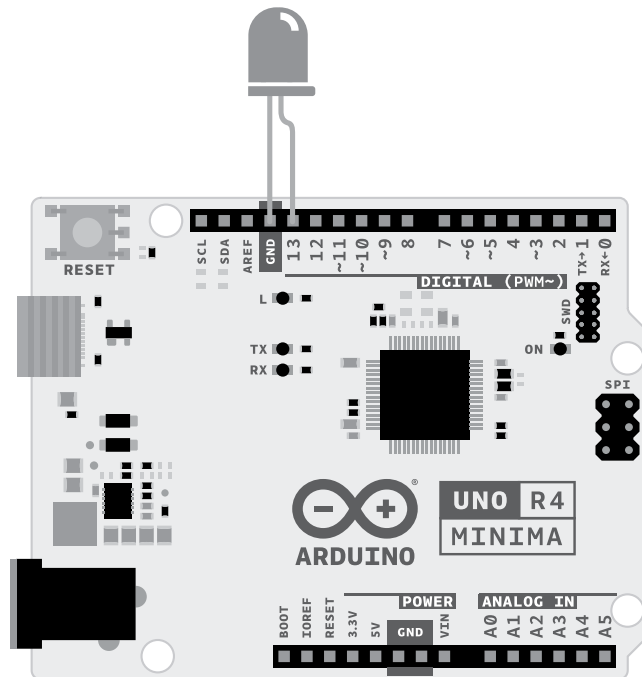
counting from 0 again. Timers can be especially useful for regularly timed events, such as checking a sensor every second — or in this case triggering an LED every second.

## Setting up the BlinkWithoutDelay sketch

To complete this project, you need:

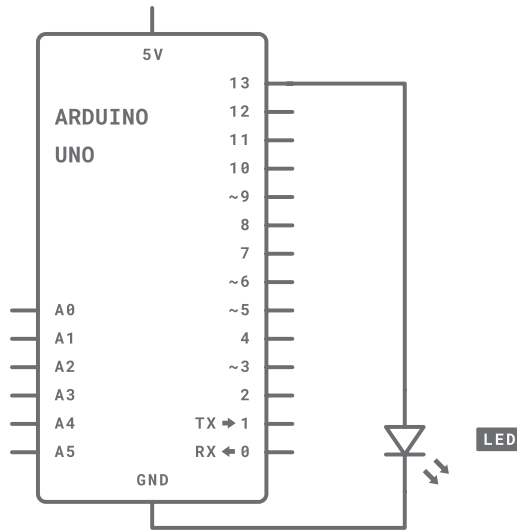
- » An Arduino Uno
- » An LED

Place the legs of the LED between pin 13 (long leg) and GND (short leg), as shown in Figures 10-1 and 10-2. This placement makes it a bit easier to see the blink in action. If you don't have an LED, look for the one fixed to your Arduino marked L. Upload the sketch to the correct serial port to see the LED blinking away as it would with the standard Blink sketch.



**FIGURE 10-1:**  
All you need is an LED in pin 13.

Find the BlinkWithoutDelay sketch by choosing File ⇨ Examples ⇨ 02.Digital ⇨ BlinkWithoutDelay and then open it.



**FIGURE 10-2:**  
A circuit diagram showing the LED in pin 13.

The complete code for the BlinkWithoutDelay sketch is as follows:

```

/*
  Blink without Delay

  Turns on and off a light emitting diode (LED) connected to a digital pin,
  without using the delay() function. This means that other code can run at
  the same time without being interrupted by the LED code.

  The circuit:
  - Use the onboard LED.
  - Note: Most Arduinos have an on-board LED you can control. On the UNO, MEGA
    and ZERO it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN
    is set to the correct LED pin independent of which board is used.
    If you want to know what pin the on-board LED is connected to on your
    Arduino model, check the Technical Specs of your board at:
    https://docs.arduino.cc/hardware/

  created 2005
  by David A. Mellis
  modified 8 Feb 2010
  by Paul Stoffregen
  modified 11 Nov 2013
  by Scott Fitzgerald
  modified 9 Jan 2017
  by Arturo Guadalupi
  This example code is in the public domain.

```

```

https://docs.arduino.cc/built-in-examples/digital/BlinkWithoutDelay/
*/

// constants won't change. Used here to set a pin number:
const int ledPin = LED_BUILTIN; // the number of the LED pin

// Variables will change:
int ledState = LOW; // ledState used to set the LED

// Generally, you should use "unsigned long" for variables that hold time
// The value will quickly become too large for an int to store
unsigned long previousMillis = 0; // will store last time LED was updated

// constants won't change:
const long interval = 1000; // interval at which to blink (milliseconds)

void setup() {
  // set the digital pin as output:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // here is where you'd put code that needs to be running all the time.

  // check to see if it's time to blink the LED; that is, if the difference
  // between the current time and last time you blinked the LED is bigger than
  // the interval at which you want to blink the LED.
  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= interval) {
    // save the last time you blinked the LED
    previousMillis = currentMillis;

    // if the LED is off turn it on and vice-versa:
    if (ledState == LOW) {
      ledState = HIGH;
    } else {
      ledState = LOW;
    }
    // set the LED with the ledState of the variable:
    digitalWrite(ledPin, ledState);
  }
}

```

This sketch is quite a bit longer than Blink and may seem more confusing, so we'll walk through it one line at a time to see what's happening.

## Understanding the BlinkWithoutDelay sketch

First, in the declarations, a `const int` is used to set `ledPin` to the built-in LED (also known as pin 13) because it is a constant integer and does not change:

```
const int ledPin = LED_BUILTIN;
```

Next are the variables. `ledState` is set to `LOW` so that our LED starts the sketch in an off state:

```
int ledState = LOW;
```

Then there is a new variable referred to as a `long` rather than an `int`. See the “Long and Unsigned Long” sidebar later in this chapter for more about `longs`. The first instance, `previousMillis`, stores the time in milliseconds so that you can monitor how much time has passed each time you do a loop:

```
unsigned long previousMillis = 0;
```

The second value, a constant named `interval`, is the time in milliseconds between each blink, which is set to 1000 milliseconds, or 1 second:

```
const long interval = 1000;
```

## LONG AND UNSIGNED LONG

`Longs` are for extra-long number storage and can store a value from  $-2,147,483,648$  to  $2,147,483,647$ , whereas an `int` can store only  $-32,768$  to  $32,767$ . When measuring time in milliseconds, you need access to big numbers because every second is stored as 1,000 milliseconds. To get an idea of just how big a `long` value is, imagine the maximum amount of time that it could store: 2,147,483.6 seconds or 35791.4 minutes or 596.5 hours or approximately 24.9 days!

In some cases, you have no need for a negative range, so to avoid unnecessary calculations, use an `unsigned long` instead. An `unsigned long` is similar to a regular `long` but has no negative values, which gives your `unsigned long` a whopping range of 0 to 4,294,967,295.

In the `setup` function, you have only one pin to define as `OUTPUT`. Pin 13 is referred to as `ledPin`, just as it is in the declarations:

```
void setup() {
  // set the digital pin as output:
  pinMode(ledPin, OUTPUT);
}
```

In the loop, things start to get more complicated. Because the code for the timer can be run at the end of every loop, you can add your own code at the start of the loop so that it doesn't interfere with the timer. Following your code, the timer code begins, which declares another variable: an `unsigned long` to store the current value of the timer in milliseconds. The timer code uses the function `millis()`, which returns the number of milliseconds since the current Arduino program began running. After approximately 50 days, this value resets to 0, but this is more than enough time for most applications.

```
unsigned long currentMillis = millis();
```



TIP

Variables declared inside a loop or other functions are known as *local* variables. They exist only within the function in which they are declared (and other sub-functions contained inside), and cease to exist after the function is completed. They are re-declared the next time the function is called. If you have a variable that needs to be either read or written to by other functions or pieces of code, you should use a *global* variable and declare it at the start of the sketch before the `setup` function.

Next you need to check the current `millis()` value to see how much time has passed. You do so by using a simple `if` loop that subtracts the previous value from the current value to get the difference. If that difference is greater than the interval value, the sketch knows that it's time to blink. It's important that you also tell the code to reset `previousMillis`; otherwise, it will measure the interval only once. This is what setting `previousMillis = currentMillis` does:

```
if(currentMillis - previousMillis > interval) {
  // save the last time you blinked the LED
  previousMillis = currentMillis;
}
```

Because the LED could already be on or off, the code needs to check the state of the LED to know what to do. The state is stored in `ledState`, so another simple `if` statement can check the state and do the opposite. If `LOW`, make `HIGH`; or if `HIGH`, make `LOW`. The following code updates the variable `ledState`:

```
// if the LED is off turn it on and vice-versa:
if (ledState == LOW){
```

```
    ledState = HIGH;
  } else {
    ledState = LOW;
  }
```

Now, all that is left to do is to write the newly updated state to the LED by using `digitalWrite`:

```
// set the LED with the ledState of the variable:
digitalWrite(ledPin, ledState);
}
```

This code allows you to happily blink your LED while performing any number of other functions.

## Taking the Bounce Out of Your Button

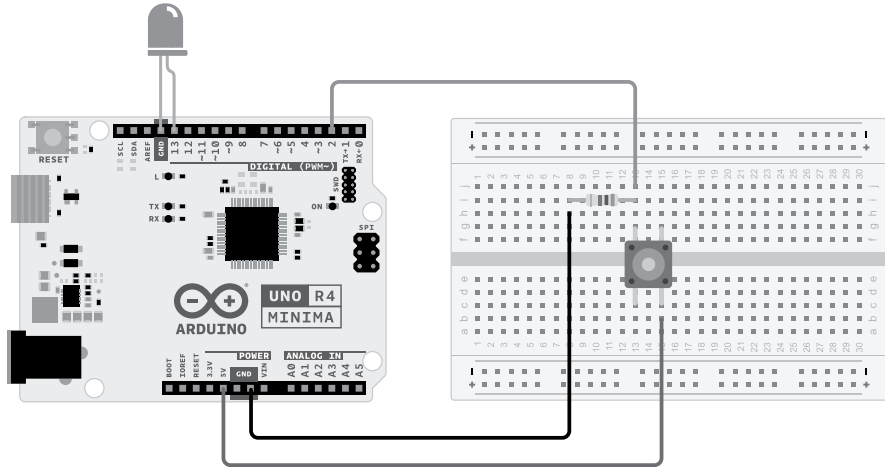
A strange occurrence that happens with pushbuttons is bouncing. The microcontroller on your Arduino can read a switch thousands of times per second, much faster than you can operate it. This speed ensures that the reading is instantaneous (as far as human perception can tell), but sometimes there is a moment of fuzziness when the contact on a switch is neither fully down nor fully up, which causes it to read on and off rapidly in quick succession until it reaches the correct state. This behavior is called *bouncing*. To remove this peculiarity, you have to ignore any sudden changes when the switch state changes by using a timer. It's relatively simple and can greatly improve the reliability of your buttons. Using timers works for your inputs as well as outputs, as noted in the preceding section, "Blinking Better."

### Setting up the Debounce sketch

Complete the circuit in Figure 10-3 to try out the Debounce sketch. You need:

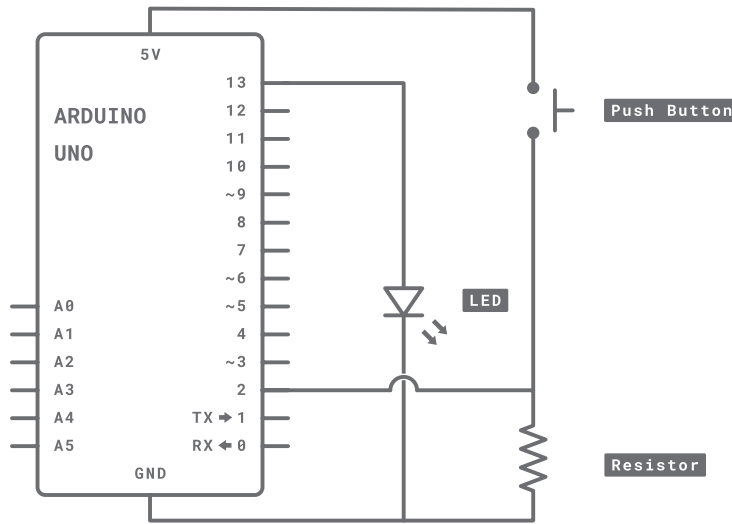
- » An Arduino Uno
- » A breadboard
- » A pushbutton
- » An LED

- » A 10k ohm resistor
- » Jump wires



**FIGURE 10-3:**  
The pushbutton circuit layout.

Complete the circuit shown in Figures 10-3 and 10-4, using a breadboard to mount the pushbutton part of the circuit. The LED can be inserted straight into pin 13 and its neighboring GND pin.



**FIGURE 10-4:**  
A circuit diagram of the pushbutton circuit.

Build the circuit and choose File ⇨ Examples ⇨ 02.Digital ⇨ Debounce to find the pushbutton Debounce sketch and open it. The complete code for the Debounce sketch is as follows:

```
/*
  Debounce

  Each time the input pin goes from LOW to HIGH (e.g. because of a push-button
  press), the output pin is toggled from LOW to HIGH or HIGH to LOW. There's a
  minimum delay between toggles to debounce the circuit (i.e. to ignore noise).

  The circuit:
  - LED attached from pin 13 to ground
  - pushbutton attached from pin 2 to +5V
  - 10 kilohm resistor attached from pin 2 to ground

  - Note: On most Arduino boards, there is already an LED on the board connected
  to pin 13, so you don't need any extra components for this example.

  created 21 Nov 2006
  by David A. Mellis
  modified 30 Aug 2011
  by Limor Fried
  modified 28 Dec 2012
  by Mike Walters
  modified 30 Aug 2016
  by Arturo Guadalupi

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/digital/Debounce/
  */

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;           // the number of the pushbutton pin
const int ledPin = 13;            // the number of the LED pin

// Variables will change:
int ledState = HIGH;              // the current state of the output pin
int buttonState;                  // the current reading from the input pin
int lastButtonState = LOW;        // the previous reading from the input pin

// the following variables are unsigned longs because the time, measured in
```

```

// milliseconds, will quickly become a bigger number than can be stored
// in an int.
unsigned long lastDebounceTime = 0; // the last time the output pin was toggled
unsigned long debounceDelay = 50; // the debounce time; increase if the output
// flickers

void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);
  // set initial LED state
  digitalWrite (ledPin, ledState);
}

void loop() {
  // read the state of the switch into a local variable:
  int reading = digitalRead(buttonPin);

  // check to see if you just pressed the button
  // (i.e. the input went from LOW to HIGH), and you've waited
  // long enough since the last press to ignore any noise:
  // If the switch changed, due to noise or pressing:
  if (reading != lastButtonState) {
    // reset the debouncing timer
    lastDebounceTime = millis();
  }

  if ((millis() - lastDebounceTime) > debounceDelay) {
    // whatever the reading is at, it's been there for longer
    // than the debounce delay, so take it as the actual current state:

    // if the button state has changed:
    if (reading != buttonState) {
      buttonState = reading;

      // only toggle the LED if the new button state is HIGH
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
    }
  }
  // set the LED:
  digitalWrite(ledPin, ledState);

  // save the reading. Next time through the loop, it'll be the lastButtonState:
  lastButtonState = reading;
}

```

When you've uploaded the sketch, you should have a reliable, debounced button. It can be difficult to see the effects — when everything's working correctly, you see just accurate button presses and responses from your LED.

## Understanding the Debounce sketch

The Debounce sketch starts by declaring two constants that are used to define the input and output pins:

```
// constants won't change. They're used here to set pin numbers:  
const int buttonPin = 2;    // the number of the pushbutton pin  
const int ledPin = 13;     // the number of the LED pin
```

The next set of variables hold details about the button state. `ledState` is set to `HIGH` so that the LED starts as being turned on; `buttonState` is left empty and holds the current state; `lastButtonState` holds the previous button state so that it can be compared with the current state:

```
// Variables will change:  
int ledState = HIGH;       // the current state of the output pin  
int buttonState;           // the current reading from the input pin  
int lastButtonState = LOW; // the previous reading from the input pin
```

Finally, two long variables store time values. These are used in a timer to monitor the time between readings and prevent any sudden changes in values, such as those that occur during bounces:

```
// the following variables are long's because the time, measured in milliseconds,  
// will quickly become a bigger number than can be stored in an int.  
long lastDebounceTime = 0; // the last time the output pin was toggled  
long debounceDelay = 50;  // the debounce time; increase if the output  
                           // flickers
```

The `setup` function is straightforward and sets only the input and output pins:

```
void setup() {  
    pinMode(buttonPin, INPUT);  
    pinMode(ledPin, OUTPUT);  
}
```

In the loop, a reading is taken from the button pin and stored in the reading variable:

```
void loop() {
  // read the state of the switch into a local variable:
  int reading = digitalRead(buttonPin);
```

The reading variable is then checked against `lastButtonState`. The first time this runs, `lastButtonState` is LOW because it was set in the variable declarations at the beginning of the sketch. In the `if` statement, the comparison symbol `!=` is used. This means: “If reading is *not* equal to `lastButtonState`, do something.” If this change has occurred, `lastDebounceTime` is updated so that a fresh comparison can be made the next time the loop runs:

```
// If the switch changed, due to noise or pressing:
if (reading != lastButtonState) {
  // reset the debouncing timer
  lastDebounceTime = millis();
}
```

If reading has been the same for longer than the debounce delay of 50 milliseconds, it can be assumed that the value is not erratic and can be forwarded to the `buttonState` variable:

```
if ((millis() - lastDebounceTime) > debounceDelay) {
  // whatever the reading is at, it's been there for longer
  // than the debounce delay, so take it as the actual current state:
  buttonState = reading;
}
```

The trusted value can then be used to trigger the LED directly. In this case, if the button is HIGH, it is closed, so the same HIGH value can be written to the LED to turn it on:

```
digitalWrite(ledPin, buttonState);
```

The current `buttonState` becomes `lastButtonState` for the next loop, and then the code returns to the start of the `loop` function:

```
lastButtonState = reading;
}
```

Some pushbuttons and triggers can be more or less reliable than others, depending on the way they’re made or used. By using bits of code like this, you can sort out any inconsistencies and create more reliable results.

# Making a Better Button

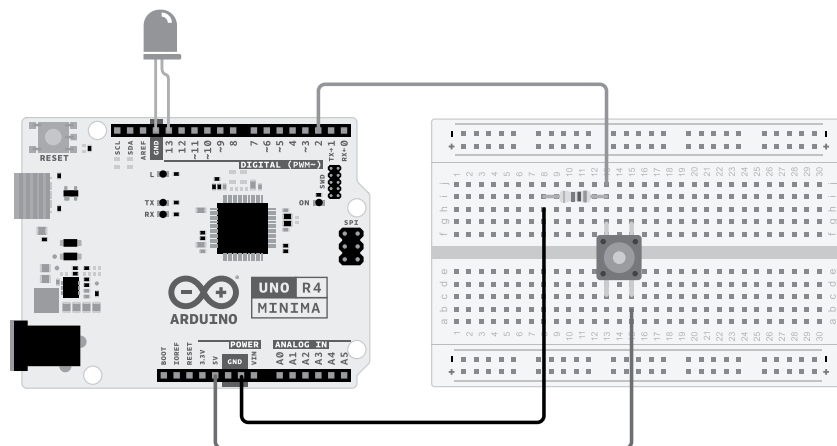
Buttons are usually simple things. They are either on or off, depending on whether or not you're pressing them. You can monitor these changes and interpret them to make a button more intelligent. If you can tell when a button has been pressed, you don't need to constantly read its value and can instead just look for this change of state. This practice is much better when connecting your Arduino to a computer, because it efficiently sends the appropriate data as it is needed, rather than hogging the serial port.

## Setting up the StateChangeDetection sketch

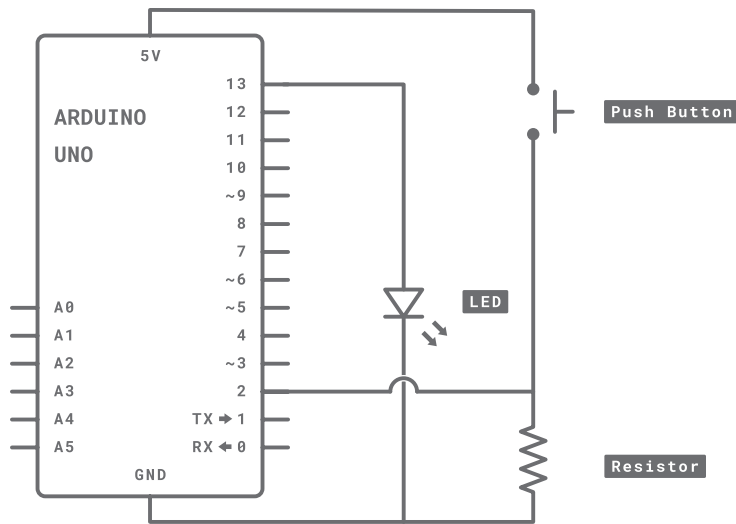
To build the StateChangeDetection circuit, you need the following:

- » An Arduino Uno
- » A breadboard
- » A pushbutton
- » A 10k ohm resistor
- » An LED
- » Jump wires

Using the layout and circuit diagrams shown in Figures 10-5 and 10-6, you can lay out a simple button circuit with an LED as an output. The hardware in this circuit is the same as the basic Button sketch, but with the use of some simple code, you can make a button a lot more intelligent.



**FIGURE 10-5:**  
A button  
circuit layout.



**FIGURE 10-6:**  
A button  
circuit diagram.

Complete the circuit and choose **File** ⇨ **Examples** ⇨ **02.Digital** ⇨ **StateChange-Detection** from the Arduino menu to load the sketch:

```

/*
State change detection (edge detection)

Often, you don't need to know the state of a digital input all the time, but
you just need to know when the input changes from one state to another.
For example, you want to know when a button goes from OFF to ON. This is called
state change detection, or edge detection.

This example shows how to detect when a button or button changes from off
to on and on to off.
The circuit:
- pushbutton attached to pin 2 from +5V
- 10 kilohm resistor attached to pin 2 from ground
- LED attached from pin 13 to ground (or use the built-in LED on most
  Arduino boards)

created 27 Sep 2005
modified 30 Aug 2011
by Tom Igoe

This example code is in the public domain.

```

```

http://www.arduino.cc/en/Tutorial/ButtonStateChange
*/

// this constant won't change:
const int  buttonPin = 2;    // the pin that the pushbutton is attached to
const int  ledPin = 13;     // the pin that the LED is attached to

// Variables will change:
int  buttonPushCounter = 0; // counter for the number of button presses
int  buttonState = 0;       // current state of the button
int  lastButtonState = 0;   // previous state of the button

void setup() {
  // initialize the button pin as a input:
  pinMode(buttonPin, INPUT);
  // initialize the LED as an output:
  pinMode(ledPin, OUTPUT);
  // initialize serial communication:
  Serial.begin(9600);
}

void loop() {
  // read the pushbutton input pin:
  buttonState = digitalRead(buttonPin);

  // compare the buttonState to its previous state
  if (buttonState != lastButtonState) {
    // if the state has changed, increment the counter
    if (buttonState == HIGH) {
      // if the current state is HIGH then the button
      // went from off to on:
      buttonPushCounter++;
      Serial.println("on");
      Serial.print("number of button pushes:  ");
      Serial.println(buttonPushCounter);
    }
    else {
      // if the current state is LOW then the button went from on to off:
      Serial.println("off");
    }
  }
  // save the current state as the last state, for next time through the loop
  lastButtonState = buttonState;
}

```

```
// turns on the LED every four button pushes by
// checking the modulo of the button push counter.
// the modulo function gives you the remainder of
// the division of two numbers:
if (buttonPushCounter % 4 == 0) {
  digitalWrite(ledPin, HIGH);
} else {
  digitalWrite(ledPin, LOW);
}
}
```

Click the Compile button to check your code. Compiling should highlight any grammatical errors and display a red text box if any are discovered. If the sketch compiles correctly, click Upload to send the sketch to your board. When the sketch has finished uploading, choose the serial monitor, and you should be presented with a readout showing when the button was turned on and off as well as how many times it was pressed. Also, the LED should illuminate every four button pushes to show that it's counting.

If nothing happens, try the following:

- » Double-check your wiring.
- » Make sure that you're using the correct pin number.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the StateChangeDetection sketch

In the StateChangeDetection sketch, the first action is to declare the constants for the sketch. The input and output pins won't change, so they are declared as constant integers, pin 2 for the pushbutton and pin 13 for the LED.

```
// this constant won't change:
const int buttonPin = 2;    // the pin that the pushbutton is attached to
const int ledPin = 13;     // the pin that the LED is attached to
```

Other variables are needed to keep track of the pushbutton's behavior. One variable is a counter that keeps a running total of the number of button presses, and

two other variables track the current and previous states of the pushbutton. These are used to monitor the button presses as the signal goes from HIGH to LOW or LOW to HIGH:

```
// Variables will change:
int buttonPushCounter = 0; // counter for the number of button presses
int buttonState = 0;      // current state of the button
int lastButtonState = 0;  // previous state of the button
```

In the setup function, the pins are set to INPUT and OUTPUT accordingly. The serial port is opened for communication to display changes in the pushbutton:

```
void setup() {
  // initialize the button pin as a input:
  pinMode(buttonPin, INPUT);
  // initialize the LED as an output:
  pinMode(ledPin, OUTPUT);
  // initialize serial communication:
  Serial.begin(9600);
}
```

The first stage in the main loop is to read the state of the pushbutton:

```
void loop() {
  // read the pushbutton input pin:
  buttonState = digitalRead(buttonPin);
```

If this state is not equal to the previous value, which happens when the pushbutton is pressed, the program progresses to the next `if ( )` statement.

```
// compare the buttonState to its previous state
if (buttonState != lastButtonState) {
```

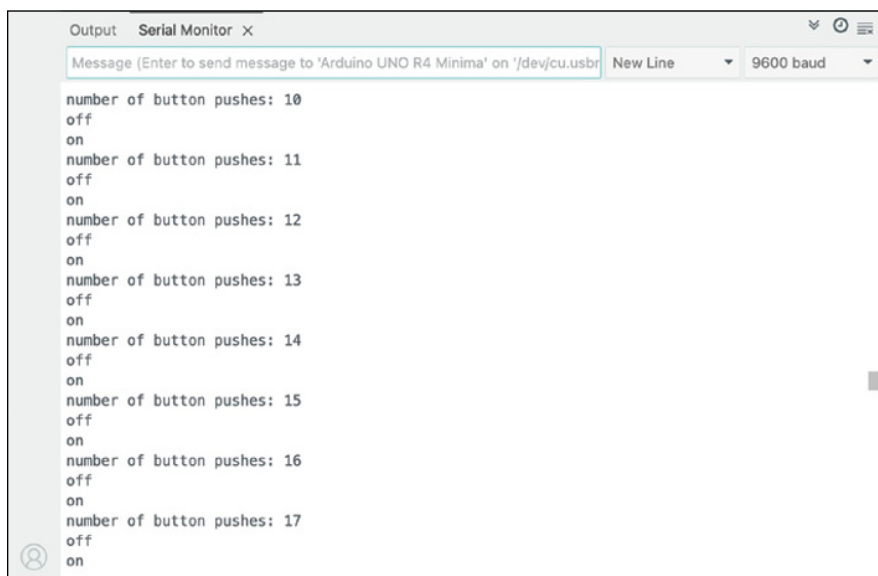
The next condition is to check whether the button is HIGH or LOW. If it is HIGH, the pushbutton has changed to on:

```
// if the state has changed, increment the counter
if (buttonState == HIGH) {
  // if the current state is HIGH then the button
  // went from off to on:
```

This bit of code increments the counter and then prints a line on the serial monitor to show the state and the number of pushes. The counter is incremented on the downward part of the button press rather than the release:

```
    buttonPushCounter++;  
    Serial.println("on");  
    Serial.print("number of button pushes: ");  
    Serial.println(buttonPushCounter);  
}
```

If the pushbutton went from HIGH to LOW, the button state is off, and this change of state is printed on the serial monitor, as shown in Figure 10-7.



**FIGURE 10-7:**  
The serial monitor provides a window into what your Arduino is experiencing.

```
    else {  
        // if the current state is LOW then the button  
        // went from on to off:  
        Serial.println("off");  
    }  
}
```

This piece of code allows you to click a pushbutton rather than hold down on it; the code also leaves plenty of room to add your own functionality.

Because a change has occurred, the current state becomes the last state in preparation for the next loop:

```
// save the current state as the last state,  
//for next time through the loop  
lastButtonState = buttonState;
```

At the end of the loop, a check is done to make sure that four button presses have occurred. If the total number of presses divided by four gives a remainder of 0, the LED pin is set to HIGH; if not the pin is set to LOW again:

```
// turns on the LED every four button pushes by  
// checking the modulo of the button push counter.  
// the modulo function gives you the remainder of  
// the division of two numbers:  
if (buttonPushCounter % 4 == 0) {  
  digitalWrite(ledPin, HIGH);  
} else {  
  digitalWrite(ledPin, LOW);  
}  
}
```



TIP

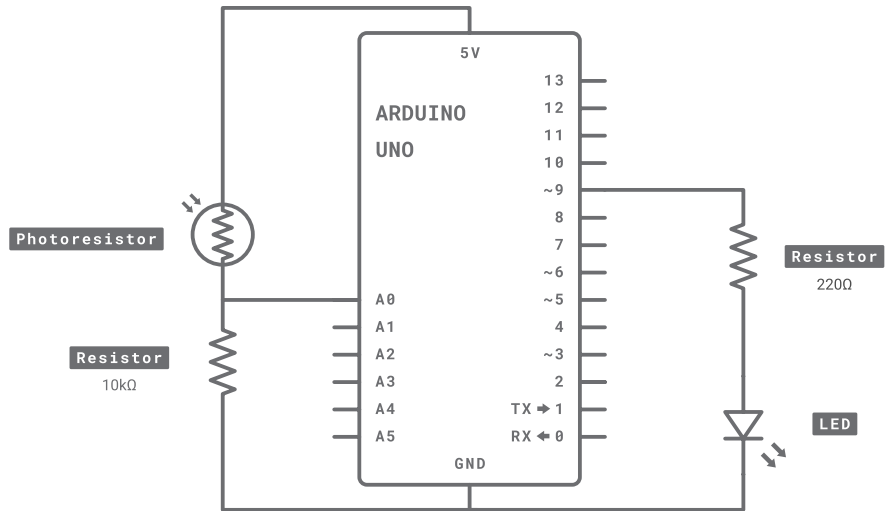
You may find that the counter in this sketch can occasionally jump up, depending on the type and quality of pushbutton that you're using, causing you to quite rightly ask, "Why doesn't this sketch include debouncing?" The sketches in the Arduino examples are designed to help you understand many individual principles easily, to equip you for any situation. To include two or more techniques in one sketch may be good for the application, but would make it more difficult for you, the person learning, to understand how each element works.

It is possible to combine numerous examples to reap the benefits of each, but this topic is beyond the scope of this book. If you want to try your hand at including two techniques in a single sketch, open two sketch examples side by side and combine them into one sketch, one line at a time, checking as you go that there are no repetitions of variables or omissions. The Compile shortcut (Ctrl+R or ⌘+R) is helpful for this task. Good luck!

## Smoothing Your Sensors

Analog sensors measure light levels or distance to a high degree of accuracy. Sometimes, however, they can be overly sensitive and flinch at the slightest change. If that's what you're looking for, great. If not, you may want to smooth





**FIGURE 10-9:** A circuit diagram of the light sensor circuit.



TIP

The sketch indicates that a potentiometer works for testing. This is true, but it is more difficult to see the effects of smoothing with a potentiometer because the mechanics of the device already provide a smooth analog input. Light, distance, and movement sensors are far more likely to need smoothing.

```

/*
  Smoothing

  Reads repeatedly from an analog input, calculating a running average and
  printing it to the computer. Keeps ten readings in an array and continually
  averages them.

  The circuit:
  - analog sensor (potentiometer will do) attached to analog input 0

  created 22 Apr 2007
  by David A. Mellis <dam@mellis.org>
  modified 9 Apr 2012
  by Tom Igoe

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/analog/Smoothing/
*/

// Define the number of samples to keep track of. The higher the number, the
// more the readings will be smoothed, but the slower the output will respond to
// the input. Using a constant rather than a normal variable lets us use this

```

```

// value to determine the size of the readings array.
const int numReadings = 10;

int readings[numReadings]; // the readings from the analog input
int readIndex = 0;         // the index of the current reading
int total = 0;             // the running total
int average = 0;          // the average

int inputPin = A0;

void setup() {
  // initialize serial communication with computer:
  Serial.begin(9600);
  // initialize all the readings to 0:
  for (int thisReading = 0; thisReading < numReadings; thisReading++) {
    readings[thisReading] = 0;
  }
}

void loop() {
  // subtract the last reading:
  total = total - readings[readIndex];
  // read from the sensor:
  readings[readIndex] = analogRead(inputPin);
  // add the reading to the total:
  total = total + readings[readIndex];
  // advance to the next position in the array:
  readIndex = readIndex + 1;

  // if we're at the end of the array...
  if (readIndex >= numReadings) {
    // ...wrap around to the beginning:
    readIndex = 0;
  }

  // calculate the average:
  average = total / numReadings;
  // send it to the computer as ASCII digits
  Serial.println(average);
  delay(1); // delay in between reads for stability
}

```

This sketch gives you a nicely smoothed reading based on what the sensor is detecting. The smoothing is achieved by averaging a number of readings. The

averaging process may slow the number of readings per second, but because the Arduino is capable of reading these changes far faster than you are, the slowness doesn't affect how the sensor works in any noticeable way.

## Understanding the Smoothing sketch

The start of the Smoothing sketch declares the constants and variables. First is the number of readings to average, declared as `numReadings` with a value of 10:

```
const int numReadings = 10;
```

The next four variables keep track of how many readings have been stored and average them. These sensor readings are added to an array (or list), which is defined here as `readings`. The number of items in the `readings` array is defined in the square brackets. Because `numReadings` has already been declared, it can be used to set the array length to 10 (which is numbered, or indexed, from 0 to 9):

```
int readings[numReadings]; // the readings from the analog input
```

*Index* is the common term for the current value and is used to keep track of how many loops or readings are taken. Because the index increases every time a reading is taken, it can be used to store the results of that reading in the correct place in your array, before increasing to store the next reading in the next position in the array:

```
int readIndex = 0; // the index of the current reading
```

The `total` variable provides a running total that is added to as readings are made. When the total is processed, the average value is stored in the `average` variable:

```
int total = 0; // the running total
int average = 0; // the average
```

The last variable is `inputPin`, the analog in pin that is being read:

```
int inputPin = A0;
```

In the `setup` function, the serial port is initialized to allow you to view the readings from the light sensor:

```
void setup()
{
  // initialize serial communication with computer:
  Serial.begin(9600);
```

Next in the code is a for loop, which is used to effectively reset the array. In the loop, `thisReading`, a new local variable, is initialized and made equal to zero. The `thisReading` variable is then compared to the length of the array. If it's less than the length of the array, the current reading of the value in that part of the array is made equal to zero:

```
// initialize all the readings to 0:
for (int thisReading = 0; thisReading < numReadings; thisReading++)
    readings[thisReading] = 0;
}
```

In layman's terms, the code reads like this: "Make a variable equal to 0, and if that variable is less than 10, make that same value in the array equal to 0; then increase the variable by 1." As you can see, the code works through all the numbers (0 to 9) and sets that same position in the array to a 0 value. After it reaches 10, the for loop ceases running and the code moves on to the main loop.



TIP

This type of automation is great for setting arrays. The alternative is to write all the numbers in the array as individual integer variables, which is a lot less efficient, both for you and the Arduino.

The first line of code in the main loop subtracts any reading in the current index of the array from the total. That value is replaced in this loop, so it is essential to remove it from the total first:

```
void loop() {
    // subtract the last reading:
    total= total - readings[index];
```

The next line obtains a new reading using `analogRead`, which is stored in the current index of the array, overwriting the previous value:

```
// read from the sensor:
readings[index] = analogRead(inputPin);
```

This reading is then added to the total to correct it:

```
// add the reading to the total:
total= total + readings[index];
```

```
// advance to the next position in the array:
index = index + 1;
```

It's important to check when the end of the array is reached so that the program doesn't loop forever without telling you your results. You can do this check with a

simple `if` statement: If the `index` value is greater than or equal to the number of readings that the sketch is looking for, set `index` back to 0. This `if` statement counts the `index` value from 0 to 9, as in `setup`, and then resets as soon as it reaches 10:

```
// if we're at the end of the array...
if (index >= numReadings)
  // ...wrap around to the beginning:
  index = 0;
```

To get the average from all the data in the array, the total is simply divided by the number of readings. The average is then displayed on the serial monitor for you to check. Because of the command used to display the message, this action could also be referred to as “printing to the serial port.” A 1-millisecond delay at the end slows the program considerably as well as helps to prevent erratic readings:

```
// calculate the average:
average = total / numReadings;
// send it to the computer as ASCII digits
Serial.println(average);
delay(1); // delay in between reads for stability
}
```

Using simple procedures like this to average your results helps control unpredictable behavior in your projects. Averaging is especially useful if the sensor readings are directly linked to your output.

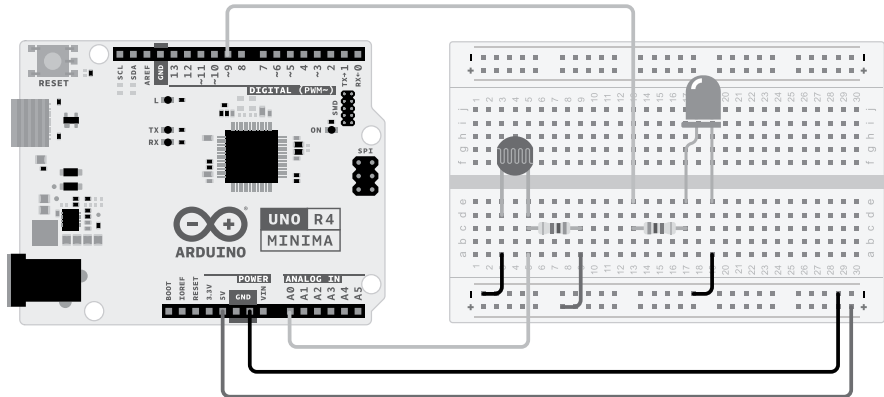
## Calibrating Your Inputs

Think of calibrating your circuit like setting up a dimmer switch with adjustable limits. A dimmer can smoothly fade a light from fully off to fully on, but only if it knows where those end points are. In a bright room, “fully on” might feel different than it does in a dark one.

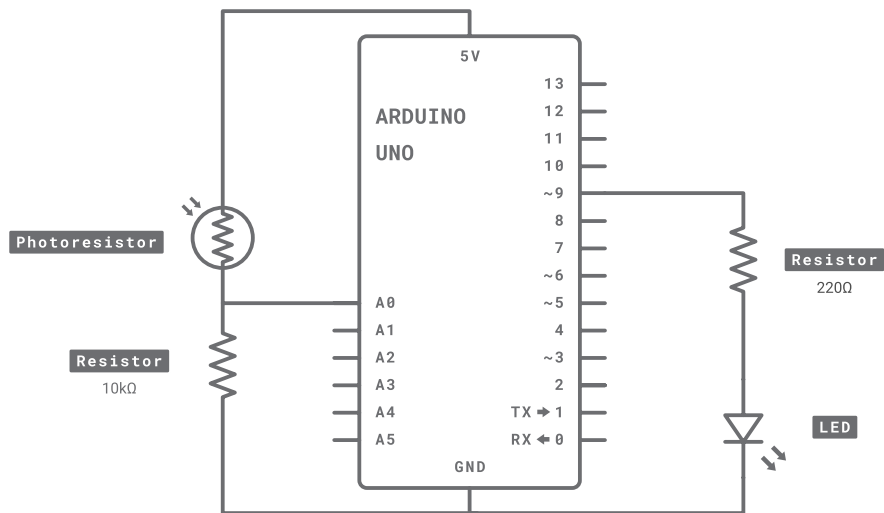
By calibrating the sensors in your Arduino project, you tailor them to their surroundings in the same way. In this example, you learn how to calibrate a light sensor. Light, of course, is highly variable, whether you’re inside, outside, in a well-lit room, or working by candlelight. As long as the Arduino understands the range of light it is likely to encounter, it can interpret those readings correctly and map them to the full brightness range of an LED. The following sketch shows you how to calibrate a light sensor to its surroundings.

# Setting up the Calibration sketch

To try the Calibration example, complete the circuit shown in Figures 10-10 and 10-11 to calibrate your light sensor automatically.



**FIGURE 10-10:** The light sensor circuit layout.



**FIGURE 10-11:** A circuit diagram of the light sensor circuit.

You need:

- » An Arduino Uno
- » A breadboard
- » An LED

- » A light sensor
- » A 10k ohm resistor
- » A 220 ohm resistor
- » Jump wires

Build the circuit and go to File ⇨ Examples ⇨ 03.Analog ⇨ Calibration to find the sketch. The code for this example is as follows:

```
/*
  Calibration

  Demonstrates one technique for calibrating sensor input. The sensor readings
  during the first five seconds of the sketch execution define the minimum and
  maximum of expected values attached to the sensor pin.

  The sensor minimum and maximum initial values may seem backwards. Initially,
  you set the minimum high and listen for anything lower, saving it as the new
  minimum. Likewise, you set the maximum low and listen for anything higher as
  the new maximum.

  The circuit:
  - analog sensor (potentiometer will do) attached to analog input 0
  - LED attached from digital pin 9 to ground

  created 29 Oct 2008
  by David A Mellis
  modified 30 Aug 2011
  by Tom Igoe

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/analog/Calibration/
*/

// These constants won't change:
const int sensorPin = A0; // pin that the sensor is attached to
const int ledPin = 9; // pin that the LED is attached to

// variables:
int sensorValue = 0; // the sensor value
int sensorMin = 1023; // minimum sensor value
```

```

int sensorMax = 0; // maximum sensor value

void setup() {
  // turn on LED to signal the start of the calibration period:
  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);
  // calibrate during the first five seconds
  while (millis() < 5000) {
    sensorValue = analogRead(sensorPin);

    // record the maximum sensor value
    if (sensorValue > sensorMax) {
      sensorMax = sensorValue;
    }

    // record the minimum sensor value
    if (sensorValue < sensorMin) {
      sensorMin = sensorValue;
    }
  }

  // signal the end of the calibration period
  digitalWrite(13, LOW);
}

void loop() {
  // read the sensor:
  sensorValue = analogRead(sensorPin);

  // apply the calibration to the sensor reading
  sensorValue = map(sensorValue, sensorMin, sensorMax, 0, 255);

  // in case the sensor value is outside the range seen during calibration
  sensorValue = constrain(sensorValue, 0, 255);

  // fade the LED using the calibrated value:
  analogWrite(ledPin, sensorValue);
}

```

Upload the sketch, and let your Arduino settle with your normal ambient light levels for five seconds. Then try moving your hand over the light sensor. You should find it a lot more responsive than it is when it's just reading the analog value normally, and the LED should have a range from fully on when the light sensor is uncovered to fully off when the light sensor is covered.

## Understanding the Calibration sketch

The first part of the Calibration sketch lays out all the constants and variables. The constants are the pins for the light sensor and the LED. Note that the LED should fade up and down, so it must use a PWM pin:

```
// These constants won't change:
const int sensorPin = A0; // pin that the sensor is attached to
const int ledPin = 9; // pin that the LED is attached to
```

The variables are used for the current sensor value and the minimum and maximum values of the sensor. You can see that `sensorMin` is initially set to a high value and `sensorMax` is set to a low one because they must work down and up, respectively, to set the minimum and maximum values:

```
// variables:
int sensorValue = 0; // the sensor value
int sensorMin = 1023; // minimum sensor value
int sensorMax = 0; // maximum sensor value
```

In `setup`, quite a bit is going on. First, the usual `pinMode` sets pin 13 as an `OUTPUT`. Next is a `digitalWrite` to pin 13 to set it `HIGH`, which signals that the sensor is in its calibration phase:

```
void setup() {
  // turn on LED to signal the start of the calibration period:
  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);
}
```

For the first 5 seconds the sensor will calibrate. Because `millis()` starts counting the (milli)second that the program starts, the easiest way to count for 5 seconds is to use a `while` loop. The following code continues to check the value of `millis()`. As long as this value is less than 5000 (5 seconds), it carries out the code inside the curly brackets:

```
// calibrate during the first five seconds
while (millis() < 5000) {
```

The brackets contain the calibration code. `sensorValue` stores the current sensor reading. If this reading is more than the maximum or less than the minimum (or both), the values are updated. Because this happens during five seconds, you get a number of readings, and they all help to better define the expected range:

```

sensorValue = analogRead(sensorPin);

// record the maximum sensor value
if (sensorValue > sensorMax) {
    sensorMax = sensorValue;
}

// record the minimum sensor value
if (sensorValue < sensorMin) {
    sensorMin = sensorValue;
}
}

```

The LED pin is then written LOW to indicate that the calibration phase is over:

```

// signal the end of the calibration period
digitalWrite(13, LOW);
}

```

Now that the range is known, it just needs to be applied to the output LED. A reading is taken from `sensorPin`. Because the reading is in the range 0 to 1023, it must be mapped to the LED's range of 0 to 255. `sensorValue` is converted to this new range by using the `map()` function, which uses the `sensorMin` and `sensorMax` values from the calibration to specify the current range rather than using the full range of 0 to 1023:

```

void loop() {
    // read the sensor:
    sensorValue = analogRead(sensorPin);

    // apply the calibration to the sensor reading
    sensorValue = map(sensorValue, sensorMin, sensorMax, 0, 255);
}

```

It is still possible for the sensor to read values outside those of the calibration, so `sensorValue` must be restricted by using the `constrain()` function. Any values outside 0 to 255 are ignored. The calibration gives a good idea of the range of values, so any larger or smaller values are likely to be anomalies:

```

// in case the sensor value is outside the range seen during calibration
sensorValue = constrain(sensorValue, 0, 255);
}

```

All that is left to do is to update the LED with the mapped and constrained value by using the `analogWrite` function on `ledPin`:

```
// fade the LED using the calibrated value:  
  analogWrite(ledPin, sensorValue);  
}
```

This code should give you a better representation of your sensor's changing values relative to your environment. The calibration runs only once when the program is started, so if the range still seems off, it's best to restart it or calibrate over a longer period. Calibration is designed to remove *noise* — erratic variations in the readings — so you should also make sure that the environment being measured doesn't have anything in it that you don't want to measure.

- » Learning about sensors
- » Understanding the complexities of different inputs
- » Paying the right amount
- » Knowing where to use sensors
- » Wiring some examples

## Chapter **11**

# Common Sense with Common Sensors

In my experience of teaching, I often find that when people first have an idea, they get caught up in how to carry it out using a specific piece of hardware they've found instead of focusing on what they want to achieve. If Arduino is a toolbox with the potential of solving numerous problems, using the right tool for the right job is key.

If you go to any Arduino-related site, you're likely to see a list of sensors, and finding the right ones for your project can be baffling. A common next step is to search the web for projects similar to the one you want to do to see what other people have built. Their efforts and successes can be a great source of inspiration and knowledge, but they can also plunge you into a black hole of too many possible solutions as well as solutions that are overkill for your needs.

In this chapter, you discover not only more about different sensors and how to use them but also — and more important — *why* to use them.

Note that all prices are approximate for buying an individual sensor to give you a rough idea of the cost. If you buy in bulk or do some thorough shopping around, you should be able to make considerable savings. To find some places to start shopping, read Chapter 16.

## Making Buttons Easier

The first sensor described in this book (in Chapter 6), and arguably the best, is the pushbutton. Many kinds of pushbuttons are available. Note that switches are also included in this category. Generally, switches stick in their position in the same way that a light switch does, whereas buttons pop back. Some exceptions to this general rule are microswitches and toggle buttons. They are essentially the same electrically, and only differ mechanically.

If you plan to use a button for your project, run through the following considerations:

» **Complexity:** In its simplest form, a pushbutton can be two metal contacts that are pushed together. At its most complex, it can be a set of carefully engineered contacts in an enclosed pushbutton. Pushbuttons tend to be mounted in enclosures designed for different uses. A pushbutton like the one in your kit is perfect for breadboard layouts and suited to prototyping. If it were used in the real world, it would need protecting. Take apart an old game console controller and you may well find an enclosed pushbutton inside. If you need a more industrial button, such as an emergency stop button, the switch may be larger and more robust, and may even contain a bigger spring to handle the force of someone hitting or kicking it.

The great thing about pushbuttons is that they never get complicated. However, the right spring or click to a button can make all the difference in the quality of your project, so choose wisely.

» **Cost:** The cost of a pushbutton varies greatly depending on the quality of the enclosure and the materials used. The prices for RS Components range from 10 cents for a microswitch to around \$150 for an industrial stop button in an enclosure. It's possible to use cheaper buttons for most applications.

» **Location:** You can use buttons to detect presses from intentional human contact (or even unintentional, if you are clever with how you house the buttons). Museum exhibits are a great example of using buttons to register intentional human contact. People “get” how to use buttons. They're



TIP

everywhere, and people use them every day without thinking. Sometimes it may seem clever to use subtler methods, but if in doubt, a button is always a safe option.

Also consider how you might apply the use of a button to what's already in place. For example, maybe you're monitoring how often a door is opened in your house. If you put a highly sensitive microswitch against the door when it's closed, that switch tells you every time the door moves away from it.

In Chapters 6 and 10, you learn how to wire a button circuit and how to refine it, respectively. In the example in the following section, you learn how to simplify the hardware of your button. By using a hidden feature of your Arduino, you can use a button with no additional hardware.

## Implementing the DigitalInputPullup sketch

The basic button circuit is a simple one, but it can be made simpler still by using a little-known function on your microcontroller. On the basic button example in Chapter 6, a *pull-down* resistor is connected to ground to make the button pin read LOW. Whenever pressed, the button connects to 5V and goes HIGH. This behavior allows you to read the button's state as an input.

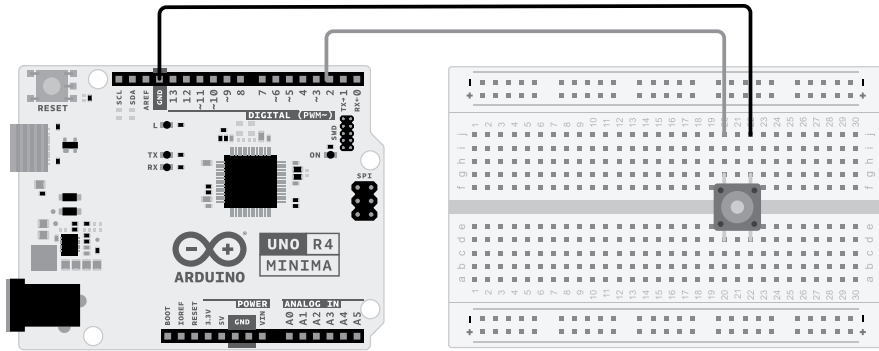
In the microcontroller, an internal *pull-up* resistor can be activated to give you a constant HIGH value. When a button connected to ground is pressed, it grounds the current and sets the pin to LOW. This design gives you the same functionality as the basic example from Chapter 6, but the logic is inverted: HIGH is an open switch, and LOW is a closed switch. The wiring is, therefore, simpler because you eliminate the need for an extra wire and resistor.

To complete this example, you need the following:

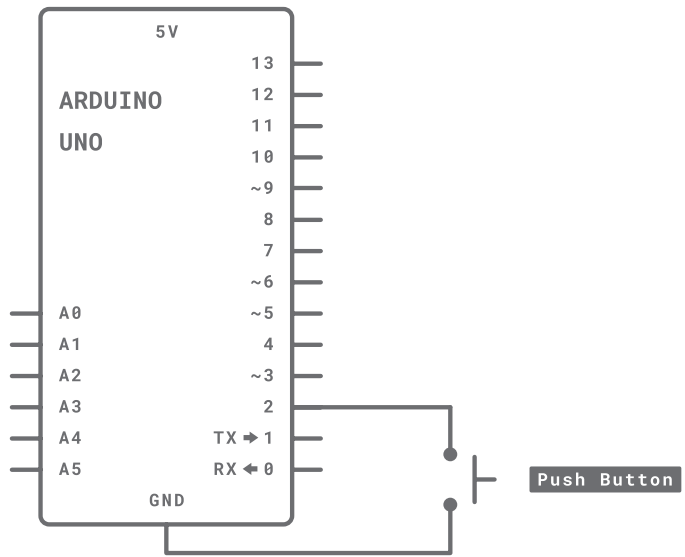
- » An Arduino Uno
- » A breadboard
- » A pushbutton
- » An LED (optional)
- » Jump wires

Complete the circuit shown in Figures 11-1 and 11-2 to try the new, simpler push-button using the DigitalInputPullup sketch.

**FIGURE 11-1:**  
The pushbutton circuit layout.



**FIGURE 11-2:**  
A circuit diagram of the pushbutton circuit.



**REMEMBER**

An LED on the board is already linked to pin 13, but if you want to accentuate the output, you can insert an LED straight into pin 13 and its neighboring GND pin.

Complete the circuit and choose **File** ⇨ **Examples** ⇨ **02.Digital** ⇨ **DigitalInput-Pullup** to find the `DigitalInputPullup` sketch.

```

/*

Input Pull-up Serial

This example demonstrates the use of pinMode(INPUT_PULLUP). It reads a
Digital input on pin 2 and prints the results to the Serial Monitor.

```

The circuit:

- momentary switch attached from pin 2 to ground
- built-in LED on pin 13

Unlike `pinMode(INPUT)`, there is no pull-down resistor necessary. An internal 20K-ohm resistor is pulled to 5V. This configuration causes the input to read HIGH when the switch is open, and LOW when it is closed.

created 14 Mar 2012

by Scott Fitzgerald

This example code is in the public domain.

<https://docs.arduino.cc/built-in-examples/digital/InputPullupSerial/>

```
*/  
  
void setup(){  
  //start serial connection  
  Serial.begin(9600);  
  //configure pin2 as an input and enable the internal pull-up resistor  
  pinMode(2, INPUT_PULLUP);  
  pinMode(13, OUTPUT);  
  
}  
  
void loop(){  
  //read the pushbutton value into a variable  
  int sensorVal = digitalRead(2);  
  //print out the value of the pushbutton  
  Serial.println(sensorVal);  
  
  // Keep in mind the pullup means the pushbutton's  
  // logic is inverted. It goes HIGH when it's open,  
  // and LOW when it's pressed. Turn on pin 13 when the  
  // button's pressed, and off when it's not:  
  if (sensorVal == HIGH) {  
    digitalWrite(13, LOW);  
  }  
  else {  
    digitalWrite(13, HIGH);  
  }  
}
```

# Understanding the DigitalInputPullup sketch

The DigitalInputPullup sketch is similar to the standard button sketch but with a few changes. In `setup`, serial communication is started to monitor the state of the button. Next, the `pinMode` of the inputs and outputs is set. Pin 2 is your button pin, but instead of setting it to `INPUT`, you use `INPUT_PULLUP`. Doing so activates the internal pull-up resistor. Pin 13 is set to be an output as an LED control pin:

```
void setup(){
  //start serial connection
  Serial.begin(9600);
  //configure pin2 as an input and enable the internal pull-up resistor
  pinMode(2, INPUT_PULLUP);
  pinMode(13, OUTPUT);
}
```

In the main loop, you read the value of the pull-up pin and store it in the `sensorVal` variable. You then print the variable to the serial monitor to show you what value is being read:

```
void loop(){
  //read the pushbutton value into a variable
  int sensorVal = digitalRead(2);
  //print out the value of the pushbutton
  Serial.println(sensorVal);
}
```

But because the logic is inverted, you need to invert your `if ( )` statement to make it correct. A `HIGH` value is open and a `LOW` value is closed. Inside the `if ( )` statement, you can write any actions to perform. In this case, the LED is being turned off, or set `LOW`, whenever the button pin is open, or pulled `HIGH`:

```
// Keep in mind the pullup means the pushbutton's
// logic is inverted. It goes HIGH when it's open,
// and LOW when it's pressed. Turn on pin 13 when the
// button's pressed, and off when it's not:
if (sensorVal == HIGH) {
  digitalWrite(13, LOW);
}
else {
  digitalWrite(13, HIGH);
}
}
```

This method is great when you don't have enough spare components, because it allows you to make a switch with only a couple of wires, if necessary. This functionality can be used on any digital pin, but only for inputs.

## Exploring Piezo Sensors

In Chapter 7, you learn how to make sound using a piezo buzzer, but you should know that you have another way to use the same hardware as an input rather than as an output. To make a sound with a piezo, you put a current through it and it vibrates, so it follows that if you vibrate the same piezo, you generate a small amount of electrical current. Using a piezo in this way allows you to create a *knock sensor* and is used to measure vibrations on the surface to which it is fixed.

Piezos vary in size, and that determines the scale of the vibrations that they can detect. Small piezos are extremely sensitive to vibrations and need little to max out their range. Bigger piezos have a broader range, but more vibration is necessary to register a reading. In addition, specialized piezo sensors can act as inputs to detect flex, touch, vibration, and shock. These cost slightly more than a basic piezo element, and are usually made from a flexible film, which makes them a lot more robust.

When using a piezo, consider the following:

- » **Complexity:** Piezos are relatively simple to wire, needing only a resistor to function in a circuit. Because the top half is made of a fragile ceramic, it is often enclosed in a plastic case, which makes it easy to mount and avoids any direct contact with the fragile solder joints on the surface of the piezo.
- » **Cost:** Piezo elements are inexpensive, costing from around 20 cents for the cheapest elements without a casing to \$15 for high-power piezo buzzers. As an input, a piezo element is preferable to the more specific piezo buzzer. The usual difference is a smaller form factor for buzzers, whereas elements usually have a broader base. The latter is preferable for the knock sensor because it gives you more area on the piezo as well as more contact with the surface being monitored.

Piezos are much cheaper to purchase from the major electronics companies, but because these require you to browse their vast online catalogues, you might find it more useful to buy a selection from retail stores first, such as

RadioShack, where you can see the product in real life and get a feel for the different shapes, styles, and housings.

- » **Location:** Knock sensors are not usually used as a direct input. Because they are so fragile, having people tapping on them all the time is risky. Instead, fix your piezo to a rigid surface such as wood, plastic, or metal, and let that surface take the punishment. For example, a knock sensor mounted on a staircase could be discreet and unobtrusive but still give highly accurate readings.

Piezos are simple and inexpensive sensors with a large variety of uses. You can use them to detect vibrations or more directly in a homemade electric drum kit. This section's example shows you how to wire your own set of piezo knock sensors.



WARNING

When you're using a bare piezo disk, strong knocks can generate voltages above 5V. To protect your analog input, always include a resistor (typically 1 M  $\Omega$ ) in parallel or series with the sensor.

## Implementing the Knock sketch

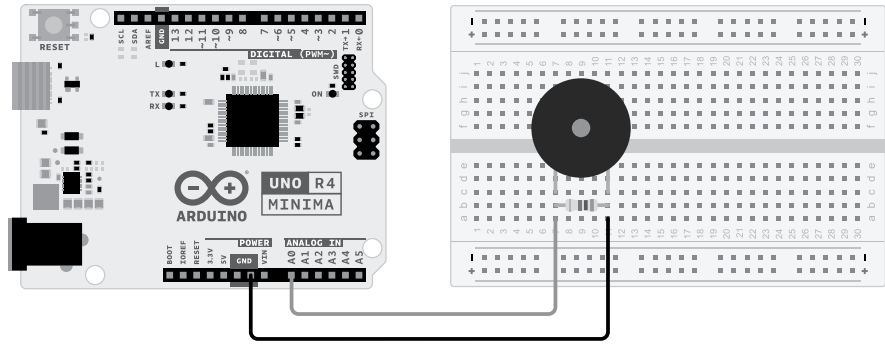
Knock sensors use a piezo element to measure vibration. When a piezo vibrates, it produces a voltage that can be interpreted by your Arduino as an analog signal. Piezo elements are more commonly used as buzzers, which inversely make a vibration when a current is passed through them.

You need the following:

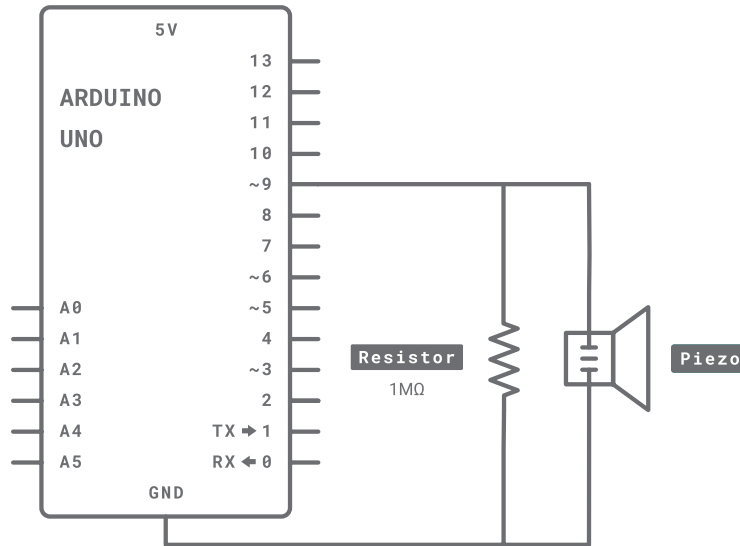
- » An Arduino Uno
- » A breadboard
- » A piezo
- » A 1M ohm resistor
- » Jump wires

Using the layout and circuit diagrams in Figures 11-3 and 11-4, assemble the circuit for the knock sensor. The hardware in this circuit is similar to the piezo buzzer sketch in Chapter 7, but with a few changes you can make this piezo element into an input as well.

**FIGURE 11-3:**  
A knock sensor  
circuit layout.



**FIGURE 11-4:**  
A knock sensor  
circuit diagram.



Complete the circuit and choose File ⇨ Examples ⇨ 06.Sensors ⇨ Knock from the Arduino menu to load the sketch.

```

/*
Knock Sensor

This sketch reads a piezo element to detect a knocking sound.
It reads an analog pin and compares the result to a set threshold.
If the result is greater than the threshold, it writes "knock" to the serial
port, and toggles the LED on pin 13.

```

The circuit:

- positive connection of the piezo attached to analog in 0
- negative connection of the piezo attached to ground
- 1 megohm resistor attached from analog in 0 to ground

created 25 Mar 2007

by David Cuartielles <<http://www.0j0.org>>

modified 30 Aug 2011

by Tom Igoe

This example code is in the public domain.

<https://docs.arduino.cc/built-in-examples/sensors/Knock/>\*

```
// these constants won't change:
const int ledPin = 13;      // led connected to digital pin 13
const int knockSensor = A0; // the piezo is connected to analog pin 0
const int threshold = 100; // threshold value to decide when the detected
                          // sound is a knock or not

// these variables will change:
int sensorReading = 0;      // variable to store the value read from the
                          // sensor pin
int ledState = LOW;        // variable used to store the last LED status,
                          // to toggle the light

void setup() {
  pinMode(ledPin, OUTPUT); // declare the ledPin as OUTPUT
  Serial.begin(9600);      // use the serial port
}

void loop() {
  // read the sensor and store it in the variable sensorReading:
  sensorReading = analogRead(knockSensor);

  // if the sensor reading is greater than the threshold:
  if (sensorReading >= threshold) {
    // toggle the status of the ledPin:
    ledState = !ledState;
    // update the LED pin itself:
    digitalWrite(ledPin, ledState);
  }
}
```

```
// send the string "Knock!" back to the computer, followed by newline
Serial.println("Knock!");
}
delay(100); // delay to avoid overloading the serial port buffer
}
```

Press the Compile button to check your code. Doing so highlights any grammatical errors and lights them up in red. If the sketch compiles correctly, click Upload to send the sketch to your board. When it is finished uploading, choose the serial monitor and give the surface that your piezo is on a good knock. If it's working, *Knock!* appears on the serial monitor and the LED changes with each successful knock.

If nothing happens, double-check your wiring:

- » Make sure that you're using the correct pin number.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the Knock sketch

In the Knock sketch, the first declarations are constant values, the LED pin number, the knock sensor pin number, and the threshold for a knock value. These are set and don't change throughout the sketch:

```
// these constants won't change:
const int ledPin = 13; // led connected to digital pin 13
const int knockSensor = A0; // the piezo is connected to analog pin 0
const int threshold = 100; // threshold value to decide when the
// detected sound is a knock or not
```

Two variables do change — the current sensor reading and the state of the LED:

```
// these variables will change:
int sensorReading = 0; // variable to store the value read from the
// sensor pin
int ledState = LOW; // variable used to store the last LED status,
// to toggle the light
```

In setup, the LED pin is set to be an output and the serial port is opened for communication:

```
void setup() {
  pinMode(ledPin, OUTPUT); // declare the ledPin as OUTPUT
  Serial.begin(9600);      // use the serial port
}
```

The first line in the loop reads the analog value from the knock sensor pin:

```
void loop() {
  // read the sensor and store it in the variable sensorReading:
  sensorReading = analogRead(knockSensor);
```

This value is compared to the threshold value:

```
// if the sensor reading is greater than the threshold:
if (sensorReading >= threshold) {
```

If the value of `sensorReading` is greater than or equal to the `threshold` value, the LED's state is switched between 0 and 1 using `!`, the NOT symbol. The `!` in this case is used to return the opposite Boolean value of the `ledState` variable's current value. As you know, Booleans are either 1 or 0 (true or false), the same as the possible values of `ledState`. This line of code could be written as "make `ledState` equal to whatever value it is not:"

```
// toggle the status of the ledPin:
ledState = !ledState;
```

The `ledState` value is then sent to the LED pin using `digitalWrite`. The `digitalWrite` function interprets a value of 0 as LOW and 1 as HIGH:

```
// update the LED pin itself:
digitalWrite(ledPin, ledState);
```

Finally, *Knock!* is sent to the serial port with a short delay, for stability:

```
// send the string "Knock!" back to the computer, followed by newline
Serial.println("Knock!");
}
delay(100); // delay to avoid overloading the serial port buffer
}
```



TIP

If you want more robust vibration measurement, you can also use low-cost modules such as the KY-031 knock sensor or modern accelerometers like the MPU-6050, which work in a similar way but are less fragile.

## Utilizing Pressure, Force, and Load Sensors

Three closely related kinds of sensors are commonly confused: pressure, force, and load sensors. These three sensors are extremely different in how they behave and what data they can give you, so it's important to know the difference so that you choose the one that's right for your situation. In this section, you learn about the different definitions of each of these sensors, as well as where you use them and why you use one over the other.

Consider the following as you plan:

- » **Complexity:** As you might expect, complexity increases depending on how accurate you need to be:
  - *Pressure pads* are designed to detect when pressure is applied to an area, and they come in a variety of both quality and accuracy. The simplest pressure pads are often misnamed and are really the equivalent of big switches. Inside a simple pressure pad are two layers of foil separated by a layer of foam with holes in it. When the foam is squashed, the metal contacts touch through the foam and complete the circuit. Instead of measuring pressure or weight, the pad is detecting when there is enough weight to squash the foam. These pads do a fine job and are similar to the mechanisms found inside dance mats — ample proof that you don't need to overthink your sensors!
  - For more precision, you may want to use *force sensors*, which measure the force applied by whatever is put on them within their range. Although force sensors are accurate enough to detect a change in weight, they are not accurate enough to provide a precise measurement. Force sensors are usually flexible, force-sensitive resistors — that is, resistors made on a flexible PCB that change their resistance when force is applied. The resistor itself is on the flexible circuit board, and although the board can tolerate extremely high forces and loads, protecting it from direct contact is a good idea to prevent it from bending, folding, or tearing.
  - With pressure pads at one end of the spectrum, at the other are load sensors. An example of a load sensor is found in your bathroom scale. *Load sensors* can accurately measure weight up to their limit. They work in much the same way as force sensors by changing resistance as they bend.

In most cases, a load sensor is fixed to a rigid piece of metal and monitors changes as the metal is put under strain. The changes are so minute that they often require an amplification circuit known as a Wheatstone bridge. Incorporating this kind of sensor is more complex than the others, but you can find material on the Internet that will walk you through the process.

» **Cost:** The cost of each sensor is relatively low, even for the most sensitive ones. All the materials to make a cheap pressure pad will set you back \$3; the cost for an inexpensive, entry-level pressure mat available from most electronics stores and suppliers is \$12. Force-sensitive resistors range from \$8 to \$23, but cover a much smaller area than a pressure pad, so you may need quite a few of them to cover a large area. At around \$11, load sensors are also relatively cheap, most likely because they are so widespread that mass production has knocked the price down. There may be an extra cost in time to plan and make the additional circuitry.

» **Location:** The challenge with all these sensors is housing them to prevent damage. In the case of the pressure pad and force-sensitive resistors, placing a thick layer of upholstery foam on the side that the force is coming from is a good idea. Depending on the density of the foam, it should dampen enough of the force to protect the sensor but still compress it enough for a good reading. Underneath the sensors, you want to have a solid base to give you something to push against. This base could be the floor or surface that the sensor is placed on, or you could attach a piece of MDF/plywood to the underside of the sensor. It's a good idea to protect the exterior of your pressure sensor as well, so consider something a bit sturdier than foam on the exterior. For a soft finish, upholstery vinyl is a great option. If you plan to have people walk on the surface, for example, a layer of wood on the top to sandwich the foam will spread the load and can easily be replaced, if needed.

Load sensors require little movement and should be connected to or placed in direct contact with a rigid surface. If you use a load sensor in a bathroom scale, it may take some trial and error to place the sensor in the correct location to get an accurate reading. Sometimes multiple sensors are used to get an average reading across the surface.

After you choose a sensor, you need to figure out how to use it:

- » *Pressure pads* are an extremely simple circuit, the same as for a pushbutton. The hardware of a pressure pad is also easy enough that you can make one yourself using two sheets of foil (or the more flexible conductive fabric or conductive thread), a sheet of foam, a cover, and a couple of wires.
- » *Force sensors* are also relatively easy to use and can take the place of other analog sensors, such as light or temperature sensors, in simple Arduino

circuits. The ranges of force may vary, but whatever the range, scaling the force to your needs in the code is simple.

- » *Load sensors* are probably the most complex sensor if they're being used for accurate reading, as with a set of weight scales. They require extra circuitry and an amplifier for the Arduino to read the minute changes in resistance. This topic is outside the scope of this book, so if you want to know more, get friendly with Google.

Force sensors are just like any other variable resistor and can easily be switched with potentiometer or light-dependent resistors as needed. In this section's example, you use force-sensitive resistors and the `toneKeyboard` sketch to make an Arduino piano keyboard.



TIP

For precise weight measurements, a simple HX711 amplifier board can be used with load cells. It amplifies the minute voltage changes from the strain gauge and communicates directly with the Arduino over digital pins, removing the need to build a Wheatstone bridge yourself.

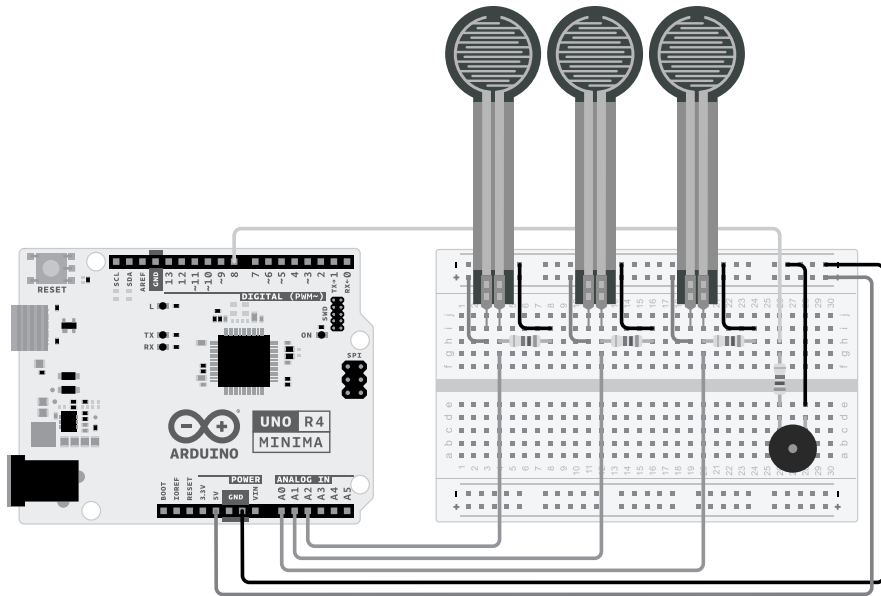
## Implementing the `toneKeyboard` sketch

You may think of pushbuttons as the perfect input for a keyboard, but force-sensitive resistors give you much more touch sensitivity. Rather than detect just a press, you can detect also the intensity of the keypress, in the same way as on a traditional piano.

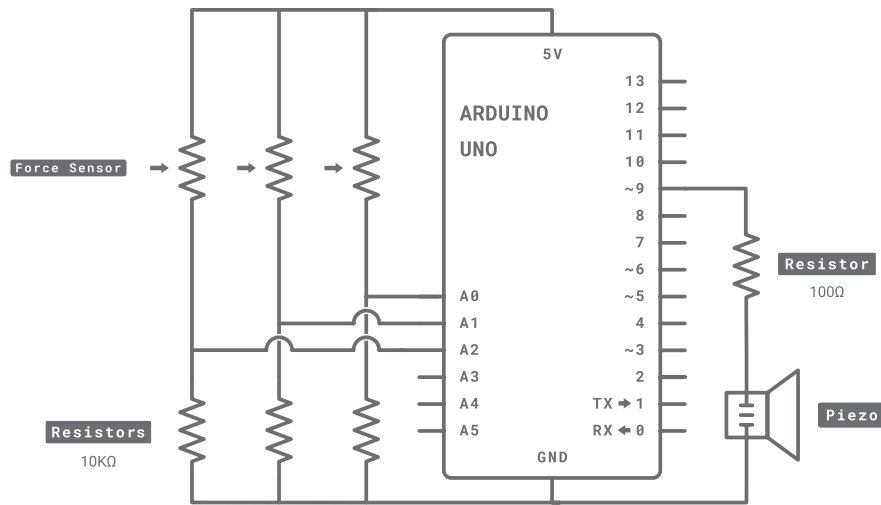
You need the following:

- » An Arduino Uno
- » A breadboard
- » Three force-sensitive resistors
- » Three 10k ohm resistors
- » One 100 ohm resistor
- » A piezo element
- » Jump wires

Using the layout and circuit diagrams in Figures 11-5 and 11-6, lay out the force-sensitive resistors and the piezo to make your own keyboard.



**FIGURE 11-5:**  
A keyboard circuit layout.



**FIGURE 11-6:**  
A keyboard circuit diagram.

Complete the circuit and choose File ⇨ Examples ⇨ 02.Digital ⇨ toneKeyboard from the Arduino menu to load the sketch.

```

/*
  Keyboard

  Plays a pitch that changes based on a changing analog input
  */

```

```

circuit:
- three force-sensing resistors from +5V to analog in 0 through 5
- three 10 kilohm resistors from analog in 0 through 5 to ground
- 8 ohm speaker on digital pin 8

created 21 Jan 2010
modified 9 Apr 2012
by Tom Igoe

This example code is in the public domain.

https://docs.arduino.cc/built-in-examples/digital/toneKeyboard/
*/

#include "pitches.h"

const int threshold = 10;    // minimum reading of the sensors that
                             // generates a note

// notes to play, corresponding to the 3 sensors:
int notes[] = {
  NOTE_A4, NOTE_B4, NOTE_C3
};

void setup() {

}

void loop() {
  for (int thisSensor = 0; thisSensor < 3; thisSensor++) {
    // get a sensor reading:
    int sensorReading = analogRead(thisSensor);

    // if the sensor is pressed hard enough:
    if (sensorReading > threshold) {
      // play the note corresponding to this sensor:
      tone(8, notes[thisSensor], 20);
    }
  }
}

```

Press the Compile button to check your code. The compiler should highlight any grammatical errors and light them up in red. If the sketch compiles correctly, click Upload to send the sketch to your board. When it has finished uploading, try the keys to make sure they're working. If they are, you're ready to play.

If nothing happens, double-check your wiring:

- » Make sure that you're using the correct pin number.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they won't work.

## Understanding the toneKeyboard sketch

The toneKeyboard sketch uses the same table of notes as the Melody sketch in Chapter 7. The first line includes `pitches.h`, which should be open in a separate tab next to the main sketch:

```
#include "pitches.h"
```

A low threshold of 10 (out of a possible 1024) is set to avoid any low readings resulting from background vibrations:

```
const int threshold = 10;    // minimum reading of the sensors that
                             // generates a note
```

The notes for each sensor are stored in an array with values (0, 1, and 2) that correspond to the analog input pin numbers (A0, A1, and A2). You can change the note values in the array manually by using the look-up table on `pitches.h`. Simply copy and paste new note values in the change the note of each sensor.

```
// notes to play, corresponding to the 3 sensors:
int notes[] = {
    NOTE_A4, NOTE_B4, NOTE_C3
};
```

In setup, you have nothing to define because the analog input pins are set to be inputs by default:

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

In the main loop, a for ( ) loop cycles through the numbers 0 to 2:

```
void loop() {
    for (int thisSensor = 0; thisSensor < 3; thisSensor++) {
```

The value of the `for ()` loop, which is used as the pin number, is stored temporarily to `sensorReading`:

```
// get a sensor reading:  
int sensorReading = analogRead(thisSensor);
```

If the reading is greater than the threshold, it is used to trigger the correct note assigned to that input:

```
// if the sensor is pressed hard enough:  
if (sensorReading > threshold) {  
    // play the note corresponding to this sensor:  
    tone(8, notes[thisSensor], 20);  
}  
}  
}
```

Because the loop happens so quickly, any delay in reading each sensor is unnoticeable.

## Sensing with Style

*Capacitive sensors* detect changes in electromagnetic fields. Every living thing — even you — has an electromagnetic field. Capacitive sensors are extremely useful because they can detect human contact and ignore other environmental factors. You're probably familiar with high-end capacitive sensors because they are present in nearly all smartphones, but they have been around since the late 1920s.

You can find dedicated capacitive-touch breakout boards such as the TTP223 and QTouch modules, but it's just as easy to make your own using the `CapacitiveSensor` library and a simple piece of wire or copper tape as the antenna.

Consider the following in your plans:

- » **Complexity:** Because all that is required is an antenna, you can be creative with what the antenna is and where it is placed. Short pieces of wire or copper tape are great for simple touch sensors. The piece of copper tape suddenly becomes a touch switch, meaning that you don't even need a pushbutton to get the same functionality. You could even connect the antenna to a bigger metal object such as a lamp, turning it into a touch lamp.

If the antenna is made from a reel of wire or a piece of foil, you can extend the range of the sensor beyond touch, which is known as a *projected capacitive*

*sensor.* Using a capacitive sensor you can detect a person's hand a few inches away from the antenna, which creates a lot of new possibilities for hiding sensors behind other materials. These discreet capacitive sensors are now commonly seen in many recent consumer electronics to remove physical buttons and maintain the sleek shape of the product. The electronics can also be placed under layers of other material, protected from the outside world.

Capacitive touch sensors are easy to make. The difficulty with projected field sensors is to determine the range of the field. The best way to determine this range is by experimentation, testing to see whether the field that you're generating is far-reaching enough.

- » **Cost:** Simple breakout modules such as the TTP223 cost \$1–\$2 each, and the open-source `CapacitiveSensor` library is free to install from the Arduino IDE's Library Manager. You only need a resistor (1 M $\Omega$  to 10 M $\Omega$ ) and a short wire or copper surface to create your own touchpad.
- » **Location:** Capacitive sensors work through many thin non-conductive materials. You can hide the antenna behind wood, acrylic, or plastic, giving the finished project a clean exterior. Thin plywood or veneer allows the touch to register easily. By covering the antenna with a non-conductive surface, you also give it a seemingly magical property, ensuring that people are left guessing at how it works.

If you're using an Arduino Uno R4 (Minima or WiFi), the easiest way to add capacitive touch is with the `Arduino_CapacitiveTouch` library, which uses the R4's built-in hardware. It needs just one I/O pin connected to a piece of foil or copper tape — no resistor required.



TIP

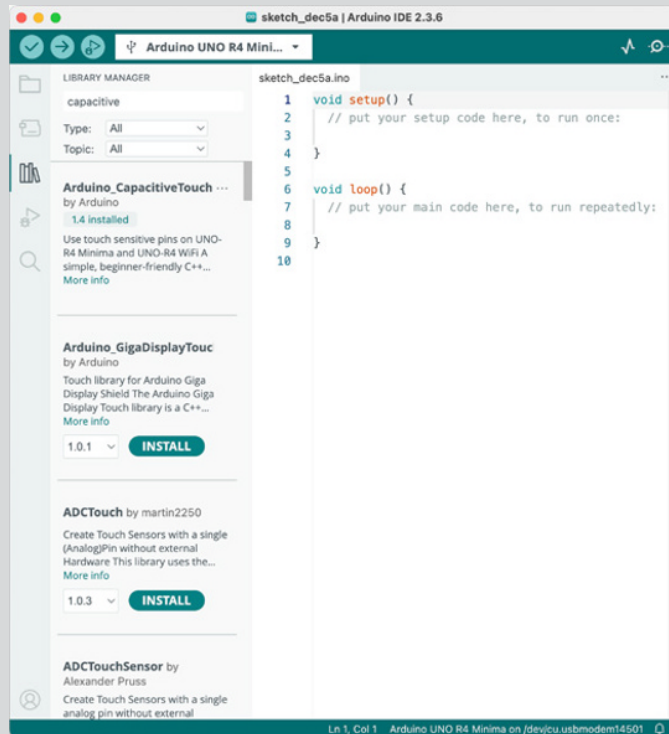
The one-pin `Arduino_CapacitiveTouch` approach is specific to Uno R4 boards. If you're on a classic Uno (R3) or another 5V AVR board, use a two-pin method (such as [github.com/moderndevise/CapacitiveSensor](https://github.com/moderndevise/CapacitiveSensor)) with a high-value resistor. For R4, stick to the one-pin approach below.

## GETTING THE ARDUINO\_ CAPACITIVETOUCH LIBRARY

Installing libraries is an essential skill, especially as you move on to more advanced projects. The Arduino IDE makes this easy through its built-in Library Manager.

1. In the Arduino IDE, choose **Sketch** ⇨ **Include Library** ⇨ **Manage Libraries**.
2. In the Library Manager window, search for **Arduino\_CapacitiveTouch (by Arduino)**.

3. Click Install.
4. After installation, you can access example sketches from  
File ⇨ Examples ⇨ Arduino\_CapacitiveTouchUNO.
5. Restart the IDE if you don't immediately see it listed under Contributed Libraries.

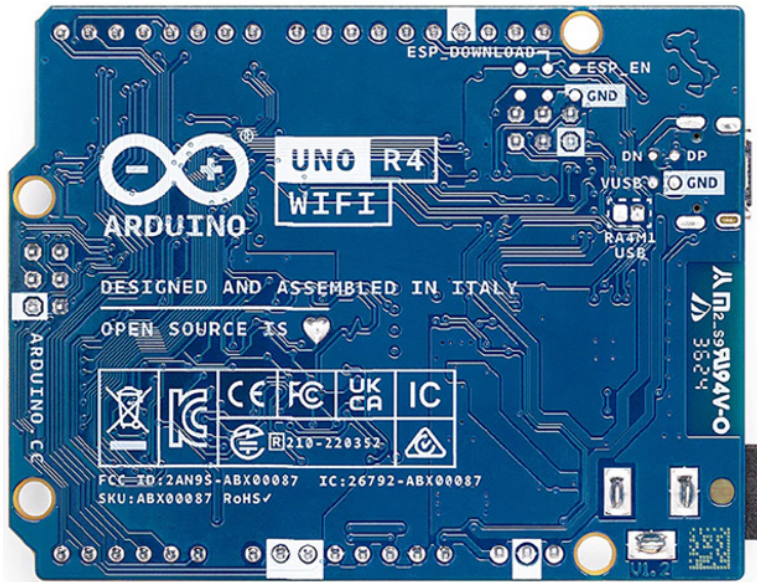


## Implementing the CapacitiveTouch sketch

For this project, you need the following:

- » An Arduino Uno R4 Minima (or R4 WiFi)
- » Your index finger

With this example you can test the code using the onboard LOVE touch pad, located on the rear of the R4 boards with the text “OPEN SOURCE IS” and a little heart icon covered in solder (as shown in Figure 11-7).



**FIGURE 11-7:** The Love Button is a heart-shaped solder pad on the rear of Uno R4 boards.

Build the CapSense circuit, and choose File ⇨ Examples ⇨ Arduino\_Capacitive-TouchUNO ⇨ LoveButton from the Arduino menu to load the sketch.

```
#include "Arduino_CapacitiveTouch.h"

CapacitiveTouch touchButton = CapacitiveTouch(LOVE_BUTTON);

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

  if(touchButton.begin()){
    Serial.println("Capacitive touch sensor initialized.");
  } else {
    Serial.println("Failed to initialize capacitive touch
sensor. Please use a supported pin.");
    while(true);
  }

  touchButton.setThreshold(2000);
}

void loop() {
  // Read the raw value from the capacitive touch sensor.
  int sensorValue = touchButton.read();
```

```
Serial.print("Raw value: ");
Serial.println(sensorValue);

// Check if the sensor is touched (raw value exceeds the
// threshold).
if (touchButton.isTouched()) {
  Serial.println("Button touched!");
}

delay(100);
}
```

Upload the sketch and open the Serial Monitor at 9600 baud. Touch the Love Button pad — you'll see the readings increase sharply when you touch it. Adjust sensitivity by changing the threshold value in `setThreshold()`. Higher thresholds require a stronger touch; lower values make the pad more sensitive to smaller changes.

If you'd prefer a custom surface or an alligator clip for testing, you can change `LOVE_BUTTON` to your pin of choice — D0, D1, D2, D3, D8, D9, D11, D13, A1, A2 on the Minima. More detail is available on the [CapacitiveTouch GitHub page](https://github.com/arduino-libraries/Arduino_CapacitiveTouch): [https://github.com/arduino-libraries/Arduino\\_CapacitiveTouch](https://github.com/arduino-libraries/Arduino_CapacitiveTouch).



TIP

If your readings seem inconsistent, check that your foil pad or copper surface has a solid connection to the pin and isn't touching any grounded metal parts. Environmental factors such as humidity or long wire length can also affect capacitance readings.

## Understanding the CapacitiveTouch sketch

At the start of the sketch, a new `CapacitiveTouch` object is declared:

```
CapacitiveTouch touchButton = CapacitiveTouch(LOVE_BUTTON);
```

This tells the library to use a specific capacitive touch point — in this case, the `LOVE_BUTTON` built into the Uno R4 boards — as the touch electrode. The Uno R4's hardware measures tiny changes in capacitance on that pin as your hand approaches or touches the pad.

The command:

```
touchButton.begin();
```

initializes the built-in capacitive-touch hardware. The line:

```
touchButton.setThreshold(2000);
```

defines the value above which a touch is considered “on.” You can adjust this number depending on your setup — if the pad is large or the environment is noisy (for example, in dry air or with long wires), increase it. If touches don’t register, lower it slightly.

The method:

```
touchButton.read();
```

returns a raw integer that reflects the measured capacitance, whereas:

```
touchButton.isTouched();
```

compares that reading with the threshold and returns `true` whenever a touch is detected. Use `isTouched()` to trigger outputs such as LEDs, relays, or sound, and keep `read()` visible in the Serial Monitor while you calibrate.

**Calibration tip:** Watch the raw values at rest, then touch the pad and note the peak. Set the threshold roughly halfway between those two numbers, leaving a little margin to avoid false triggers.



TIP

If you’re using a board with built-in touch capability, such as the Nano 33 BLE Sense or an ESP32-based board, you can replace this library with the simpler `touchRead(pin)` function, which returns capacitance directly.



REMEMBER

If your readings seem inconsistent, make sure your foil pad or copper surface has a solid connection to the selected pin and isn’t touching any grounded metal parts. Environmental factors like humidity or cable length can also affect results.

## WHAT IS A FLOAT?

A *float*, or *floating-point number*, is any number that includes a decimal point. Floats are useful when you need to record or calculate very precise measurements — for example, when reading capacitance or sensor data that changes in small increments.

However, because floats take more processing time and memory than integers, you should avoid them unless your project genuinely needs that level of precision. When you can, use whole numbers (*integers*) to keep your sketches running quickly and efficiently.

# Measuring Distance

Two sensors for measuring distance that are extremely popular are the infrared proximity sensor and the ultrasonic range finder. They work in similar ways and achieve pretty much the same thing, but it's important to choose the right sensor for the environment you're in. An *infrared proximity sensor* has a light source and a sensor. The light source bounces infrared light off an object and back to the sensor, and the time it takes the light to return is measured to indicate the object's distance.

An *ultrasonic range finder* fires out high-frequency sound waves and listens for an echo when they hit a solid surface. By measuring the time it takes a signal to bounce back, the ultrasonic range finder can determine the distance traveled.

Infrared proximity sensors are not as accurate and have a much shorter range than ultrasonic range finders.

Consider the following during planning:

- » **Complexity:** Both sensors are designed to be easy to integrate with Arduino projects. In the real world, they're used for similar electronics applications, such as proximity meters on the back of cars that beep as you approach the curb. Again, the main complexity is housing them effectively. Many infrared modules (such as the Sharp/GP2Y series) and ultrasonic modules (such as the HC-SR04) include simple mounting holes or bezels for easy panel mounting.
- » **Cost:** Infrared proximity sensors start around \$10–\$20 with ranges up to ~150 cm. HC-SR04 ultrasonic modules are very affordable (approximately \$2–\$5) with a typical range up to around 400 cm. More robust or weather-resistant ultrasonic sensors are available at higher cost.
- » **Location:** A common application for these sensors is monitoring the presence of a person or an object in a particular floor space, especially when a pressure pad would be too obvious or easy to avoid, or when a PIR sensor would measure too widely. Using a proximity sensor lets you know where someone is in a straight line from that sensor, making it a useful tool.

IR proximity sensors are okay in dark environments but perform terribly in direct sunlight. The HC-SR04 ultrasonic range finder is one of my favorite and most reliable sensors. When using ultrasonic range finders, you can also choose how wide or narrow a beam you want. A large, teardrop-shaped sensor is perfect for detecting large objects moving in a general direction, whereas narrow beams are great for precision measurements.



Ultrasonic sensors work best with hard, flat targets at roughly perpendicular angles. Very soft, angled, or narrow objects may give weaker echoes or erratic readings.

## Implementing the HC-SR04 sketch

In this example, you learn how to measure distance using the widely available HC-SR04 ultrasonic module together with the NewPing library for stable readings.

The range finder needs some minor assembly. To use the range finder in your circuit, you need to solder on either header pins (to use the range finder on a breadboard) or lengths of wire.

You can connect your range finder by using analog, pulse width, or serial communication. In this example, you learn how to measure the pulse width and convert that to distance. The analog output can be read straight into your analog input pins, but it provides less accurate results than pulse width. This example does not cover serial communication.

You need the following for the MaxSonar sketch:

- » An Arduino Uno
- » An HC-SR04 ultrasonic distance sensor
- » Jump wires

Complete the circuit from the layout and circuit diagrams in Figures 11-8 and 11-9.

Wiring:

VCC → 5V

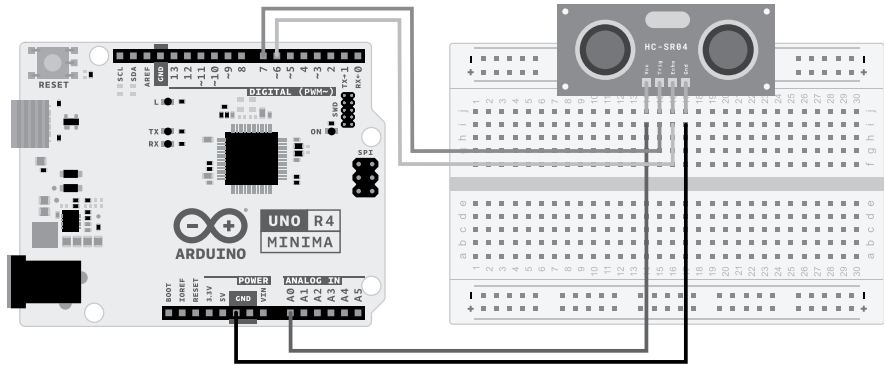
GND → GND

TRIG → D7

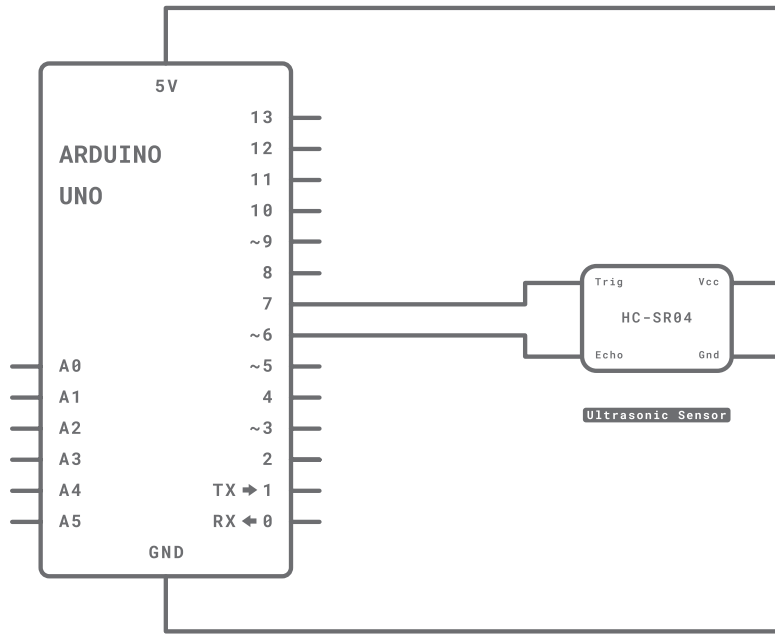
ECHO → D6

Make sure that your distance sensor is affixed to some sort of base pointed in the direction that you want to measure.

**FIGURE 11-8:**  
An HC-SR04  
circuit layout.



**FIGURE 11-9:**  
An HC-SR04  
circuit diagram.



Install the NewPing library (Sketch → Include Library → Manage Libraries, search “NewPing”, Install) and then open the example sketch File ⇄ Examples ⇄ NewPing ⇄ NewPingExample:

```
// -----
// Example NewPing library sketch that does a ping about 20 times per second.
// -----

#include <NewPing.h>

#define TRIGGER_PIN 12 // Arduino pin tied to trigger pin on the ultrasonic sensor.
```

```

#define ECHO_PIN    11 // Arduino pin tied to echo pin on the ultrasonic sensor.
#define MAX_DISTANCE 200 // Maximum distance we want to ping for (in centimeters).
                        // Maximum sensor distance is rated at 400–500cm.

NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE); // NewPing setup of pins and
                                                    // maximum distance.

void setup() {
  Serial.begin(115200); // Open serial monitor at 115200 baud to see ping results.
}

void loop() {
  delay(50); // Wait 50ms between pings (about 20 pings/sec).
            // 29ms should be the shortest delay between pings.
  unsigned int uS = sonar.ping(); // Send ping, get ping time in microseconds (uS).
  Serial.print("Ping: ");
  Serial.print(uS / US_ROUNDTRIP_CM); // Convert ping time to distance in cm and
  // print result (0 = outside set distance range)
  Serial.println("cm");
}

```

Press the Compile button to check your code. The compiler highlights any grammatical errors, turning them red. If the sketch compiles correctly, click Upload to send the sketch to your board. When it has finished uploading, open the serial monitor to see the distance measured in centimeters. If the value is fluctuating, try using an object with a bigger surface.

This sketch allows you to accurately measure distance in a straight line. Check the results with a tape measure and make adjustments to the code if you find discrepancies.

If nothing happens, double-check your wiring:

- » Make sure that you're using the correct pin number.
- » Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

## Understanding the MaxSonar sketch

The NewPingExample sketch is short, but it packs in everything you need to measure distance accurately using an ultrasonic sensor.

At the top of the sketch, three constants define the pin numbers and the maximum range of your sensor:

```
#define TRIGGER_PIN 12
#define ECHO_PIN 11
#define MAX_DISTANCE 200
```

The TRIGGER\_PIN sends out the ultrasonic pulse, while ECHO\_PIN listens for it to bounce back.

The MAX\_DISTANCE value sets the limit for how long the sketch waits before assuming there's no return echo. Setting this to the sensor's rated maximum (for example, 200 cm or 400 cm) helps keep readings quick and consistent.

Next, this line creates an instance of the NewPing class:

```
NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE);
```

The object sonar now manages all the timing required to trigger the pulse and measure the echo automatically.

Without this library, you'd have to calculate the timing manually using pulseIn(), but NewPing simplifies that process and filters out unreliable results.

In the setup() section, the Serial Monitor is started.

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

This opens a connection to your computer so you can view the distance readings in real time.

In the loop(), the following line sends a pulse and returns the echo time in microseconds:

```
unsigned int uS = sonar.ping();
```

That value represents the total time it takes for the ultrasonic pulse to travel to the target and back again. Because sound travels roughly 340 m/s, the library provides a convenient conversion constant, US\_ROUNDTRIP\_CM, which converts the time directly into centimeters:

```
Serial.print(uS / US_ROUNDTRIP_CM);
```

The resulting distance is then printed to the Serial Monitor as a simple number followed by "cm."

Each pass through the loop waits 50 milliseconds before repeating, giving you around 20 readings per second — fast enough for most interactive projects.



TIP

If your readings seem unstable, check that your sensor is pointing directly at a flat surface and that it isn't too close to the target. Ultrasonic sensors struggle with soft, angled, or very narrow objects that don't reflect sound evenly.



REMEMBER

`US_ROUNDTRIP_CM` is a handy constant built into the `NewPing` library. It divides the microsecond value by the time sound takes to travel 1 cm and back, so you can read your results directly in centimeters without doing the math yourself. You can modify the sketch to display distance in inches by replacing `US_ROUNDTRIP_CM` with `US_ROUNDTRIP_IN`.

Or, for smoother results, you can take several readings and average them using the `ping_median()` function:

```
unsigned int cm = sonar.convert_cm(sonar.ping_median(5));
```

This method takes five measurements and uses the median value to reduce noise or sudden spikes from irregular reflections.



TIP

If you need faster updates, you can lower the delay between pings, but keep it above 30 ms to avoid echo overlap — that's when one pulse is still bouncing around while the next one starts.

The `NewPingExample` is a reliable starting point for most distance-based projects, from obstacle-avoidance robots to touchless switches. It's efficient, compact, and works identically on both the Arduino Uno and Uno R4 Minima.

## Taking Sensors Further

By now you've seen that sensors come in many forms, from simple buttons and piezos to sophisticated ultrasonic and capacitive touch devices. Each converts a physical action or change into something your Arduino can understand.

Experiment with these examples, then mix and match them — that's where the really interesting projects begin!

#### IN THIS CHAPTER

- » Understanding what shields are and how they work
- » Exploring the most popular and useful Arduino shields
- » Learning how libraries make complex code easy to use
- » Finding, installing, and managing libraries in the Arduino IDE

## Chapter **12**

# Becoming a Specialist with Shields and Libraries

**T**he further you progress in learning about Arduino, the more you want to do, and it's natural to want to run before you can walk. The areas that interest you may be highly specialized in themselves and require a huge investment of time to understand. Perhaps the most important thing about Arduino is the Arduino community, which is where you can get help when you want to go further.

The traditional viewpoint that is hammered into us in education is to protect our ideas for dear life. Thankfully, many people in the Arduino community have seen past that limitation and are kind enough to share their hard work. By sharing this knowledge, the Arduino community helps the hardware and software become available to more people, who find new and interesting uses for it. If these people in turn share their results, the community continues to grow and eventually makes even the most difficult projects achievable. In this chapter, you discover the power of shared resources such as shields and libraries, even for beginners.

# Looking at Shields

*Shields* are pieces of hardware that sit on top of your Arduino, often to give it a specific purpose. For example, you can use a shield to make it easier to connect and control motors or even to turn your Arduino into something as complex as a mobile phone. A shield may start out as an interesting bit of hardware that an enthusiast has been experimenting with and wants to share with the community. Or an enterprising individual (or company) may design a shield to make an application easier based on demand from the Arduino community.

Shields can be simple or complex. They are sold preassembled or as kits. Kits allow you more freedom to assemble the shield as you need it to be. Some kits require you to assemble the circuitry of the boards, although more complex shields may already be largely assembled, needing only header pins.

Shields enable you to use your Arduino for more than one purpose and to change that purpose easily. They neatly package the electronics for that circuit in the same footprint as an Arduino, and are stackable to combine different functionalities. But they all have to use the same pins on the Arduino, so if you stack shields, watch out for those that need to use the same pins. They always connect the GND pins, too, because any communication by your Arduino and another device needs a common GND.

## Considering combinations

In theory, shields could be stacked on top of each other forever, but you should take some points into consideration before combining them:

- » **Physical size:** Some shields just don't fit on top of one another. Components that are higher than the header sockets may touch the underside of any board on top of it. This situation, which can cause short circuits if a connection is made that shouldn't be, can seriously damage your boards.
- » **Obstruction of inputs and outputs:** If an input or output is obstructed by another shield, it becomes redundant. For example, there's no point having a joystick shield or an LCD shield under another shield because no more than one can be used.
- » **Power requirements:** Some hardware requires a lot of power. Although it is all right for shields to use the same power and ground pins, there is a limit to the amount of current that can flow through the other input/output (I/O) pins: 40mA per pin and 200mA max between all I/O pins. Exceed this, and you run the risk of seriously damaging your board and any other attached shield. In most cases, you can easily remedy this problem by

powering your Arduino and shields from an external power supply so that the current isn't passed through the Arduino. Make sure to use a common GND if you're communicating between a board using I2C, SPI, or serial.

- » **Pins:** Some shields require the use of certain pins. It's important to make sure that shields aren't doubling up on the same pins. In the best case, the hardware will just be confused; in the worst case, you can send voltage to the wrong place and damage your board.
- » **Software:** Some of these shields need specific libraries to work. There can be conflicts in libraries calling on the same functions, so make sure to read up on what's required for your shield.
- » **Interference with radio/Wi-Fi/GPS/GSM:** Wireless devices need space to work. Move antennas or aerials away from the board to get a clear signal. If an antenna is mounted on the board, it's generally a bad idea to cover it. Always try to place wireless shields at the top of the stack.

## Reviewing the field

To give you an idea of available shields, this section covers some of the most interesting and useful shields on the market and shows you where to look for more information.



TIP

Note that all prices were current at the time this book was written and are liable to change, but I've included them to give you an idea of the cost. Links to products may also change, so always try searching for the product if the link is broken. Technical information is gathered from the manufacturers' websites, but you should always check the details yourself to make sure that you're buying what you need. Boards are revised occasionally, so always keep an eye out for the latest versions.

Finally, a lot of feedback on all of these products is available online, so always read the comments and forums to get a good understanding of what you're buying.

This range of shields covers a vast number of different uses and the huge potential of Arduino projects. For many projects, a shield is all you need, but a shield is also an excellent stepping-stone for proving a concept before refining or miniaturizing your project.



REMEMBER

Prices provided are from a range of distributors to show the approximate value of the items. If you are a savvy shopper or are looking to buy in bulk, you may be able to reduce the cost.

# Prototyping shields

Prototyping shields are designed to help you move from loose wires and breadboards to more permanent, reliable builds. They don't add new electronic features on their own; instead, they give you a practical way to connect components, tidy up wiring, and experiment with your own circuits directly on top of an Arduino.

The following shields are especially useful when you want to turn a one-off experiment into something you can reuse or demonstrate.

## Proto shield kit Rev3

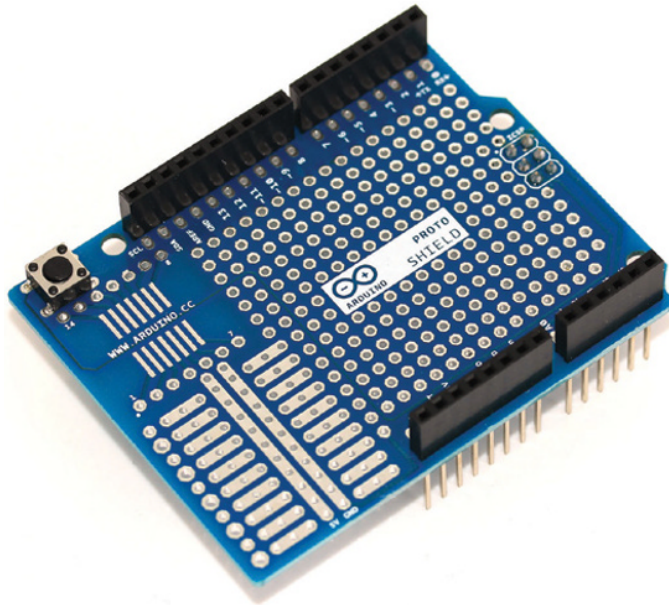
Made by: Arduino

Price: Around \$10

Pins used: Matches all Arduino Uno pins

Link: [store.arduino.cc/products/proto-shield-rev3-uno-size](https://store.arduino.cc/products/proto-shield-rev3-uno-size)

This shield (shown in Figure 12-1) provides a blank area for soldering your own components and creating permanent versions of breadboard circuits. It includes extra power and ground rails, a small SMD area, and a reset button. It's one of the simplest and most useful accessories for anyone who wants to make durable, reliable projects. The Proto shield is sold either fully assembled or as a kit that requires soldering.



**FIGURE 12-1:**  
A fully assembled  
Proto shield.

## Adafruit Proto-Screwshield

Made by: Adafruit Industries

Price: Around \$15

Pins used: Matches all Arduino Uno pins

Link: [adafruit.com/product/196](http://adafruit.com/product/196)

The Proto-Screwshield breaks out every Arduino pin to sturdy screw terminals, making it easy to connect and disconnect wires without soldering. It's ideal for semi-permanent installations or classroom use and includes a small prototyping area for custom circuits. The Proto-Screwshield is sold as a kit and requires soldering.

## Audio & display shields

Audio and display shields add ways for your Arduino to communicate with people. Instead of relying on a computer screen or serial messages, these shields let your projects make sounds, show text, or display graphics directly. They're especially useful for interactive projects, user interfaces, and installations where feedback needs to be immediate and visible.

## Adafruit Music Maker MP3 shield

Made by: Adafruit Industries

Price: Around \$30

Pins used: D2–D7, D10–D13 (SPI), A0–A5, 5V, GND

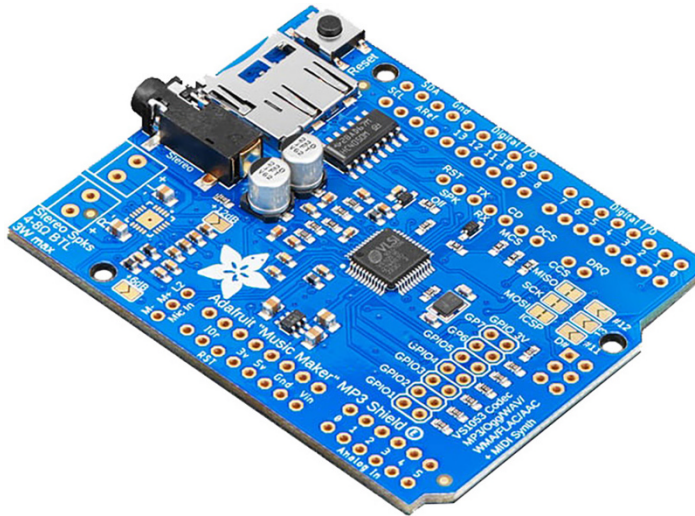
Link: [adafruit.com/product/1788](http://adafruit.com/product/1788)

The Music Maker shield (shown in Figure 12-2) replaces Adafruit's older Wave and MP3 shields with a high-quality VS1053 audio chip. It plays stereo MP3, WAV, OGG, and other formats directly from a microSD card, and it even supports basic recording. A 3.5mm stereo jack provides clear sound output for headphones or speakers, and the built-in level converter makes it safe for 5V boards. It's perfect for talking robots, sound effects, or music-playing installations.



REMEMBER

Always read the comments and forum entries on products and kits. These comments often contain a lot of detail on the ease of (or difficulty with) a product. This is also the place to voice your own problems. Just be sure that you're not repeating something that's solved further down the page; otherwise, you'll be advised to read the manual!



**FIGURE 12-2:**  
The Adafruit  
“Music Maker”  
Shield.

## Adafruit RGB 16x2 LCD shield

Made by: Adafruit Industries

Price: Around \$25

Pins used: D4–D10, 5V, GND

Link: [adafruit.com/product/714](http://adafruit.com/product/714)

This popular shield combines a two-line, 16-character LCD with a five-button keypad. The backlight can change color (red, green, blue, or any mix), and the buttons connect through a single analog input to save pins. It’s great for adding menus, controls, and simple readouts to your projects without needing a separate screen.

The RGB LCD shield is sold as a kit and requires soldering. For more details, check out the Adafruit tutorial at [learn.adafruit.com/rgb-lcd-shield](http://learn.adafruit.com/rgb-lcd-shield).

## Adafruit 2.8" TFT Touch Shield

Made by: Adafruit Industries

Price: Around \$40

Pins used: D8–D13 (SPI), A0–A3, 5V, GND

Link: [adafruit.com/product/1947](http://adafruit.com/product/1947)

This full-color touchscreen lets your Arduino display graphics, draw shapes, and respond to touch inputs. It includes an SD card slot for storing images and works with Adafruit's GFX and TouchScreen libraries. It's ideal for interactive dashboards or data displays.

A newer 3.5" version offers higher resolution, but this one remains reliable and well supported. The TFT touch shield is sold fully assembled and requires no soldering, so you can simply plug it on top of your Arduino. Check out the full tutorial at [learn.adafruit.com/adafruit-2-8-tft-touch-shield-v2](http://learn.adafruit.com/adafruit-2-8-tft-touch-shield-v2).

## Input shields

Input shields give your Arduino physical ways to receive commands. Instead of typing instructions on a keyboard, you can use joysticks, buttons, or controllers to interact with your project in real time.

These shields are ideal for robotics, games, and any project that benefits from hands-on control.

### DFRobot Gamepad shield v2.0

Made by: Adafruit

Price: \$44.95 from Adafruit

Pins used: 4, 8, 9, 10, 11, 12, 13, A4, A5

Link: [dfrobot.com/product-374.html](http://dfrobot.com/product-374.html)

If an LCD display isn't enough for you, try the TFT touch shield to add full color and touch input to your project. This display is a TFT LCD screen — a variation on a standard LCD screen that uses thin-film transistor (TFT) technology to improve the image quality — with a resolution of  $240 \times 320$  pixels and 18-bit colors, giving you 262,144 shades. The screen is also fitted with a resistive touchscreen to register finger presses anywhere on the surface of the screen.

### SparkFun MIDI shield

Made by: SparkFun Electronics

Price: Around \$20

Pins used: D0, D1 (Serial), 5V, GND

Link: [sparkfun.com/products/12898](http://sparkfun.com/products/12898)

This shield enables your Arduino to communicate with musical instruments and sequencers using the MIDI protocol. It provides standard 5-pin DIN connectors for input and output, and a small prototyping area for adding buttons or potentiometers. You can use it to send notes, change parameters, or build custom MIDI controllers that integrate directly with hardware synths. You can find more details on the SparkFun products page ([sparkfun.com/products/12898](http://sparkfun.com/products/12898)).

## Motor & power shields, navigating and sensing shields

Motor and power shields are all about movement and control. They make it possible for your Arduino to drive motors, control speed and direction, and power mechanical systems safely. Because motors can draw a lot of current, these shields also handle the electrical side of things so you don't have to design that circuitry yourself.

Navigation and sensing shields allow your Arduino to measure and understand the world around it. These shields typically work with specialized sensors to gather data such as location, movement, or environmental conditions. They're commonly used in logging, tracking, and data-collection projects.

### Adafruit Motor/Stepper/Servo shield v2.3

Made by: Adafruit Industries

Price: Around \$20

Pins used: A4, A5 (I<sup>2</sup>C bus on Uno)

Link: [adafruit.com/product/1438](http://adafruit.com/product/1438)

A flexible driver board that controls up to four DC motors, two stepper motors, or a mix of motors and servos. Because it communicates over I<sup>2</sup>C, it uses fewer pins and can stack with other shields easily. It's great for robotics or automation projects that require multiple motors. The accompanying Adafruit library includes examples for mixed motor types.

The Adafruit Motor/Stepper/Servo shield is sold as a kit and requires soldering.

You can find more details in the in-depth tutorial ([learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/](http://learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/)).



WARNING

Both the Arduino and Adafruit motor shields can drive motors directly from your board, but only within safe current limits. Always check the rated current of your motors — most small DC motors draw between *100mA* and *500mA*, but can surge much higher when starting or under load. If a motor draws more than *1 amp*,

you'll need a separate power supply and possibly a *heat sink* on the driver chip to prevent overheating.

For a detailed look at motors and power requirements, visit Adafruit's guide: <https://learn.adafruit.com/adafruit-motor-selection-guide/>

## Arduino Motor shield Rev3

Made by: Arduino

Price: Around \$25

Pins used: D3, D11, D12, D13, plus external VIN and GND

Link: [store.arduino.cc/products/arduino-motor-shield-rev3](https://store.arduino.cc/products/arduino-motor-shield-rev3)

This official shield lets you drive two DC motors or one stepper motor with direction and speed control. Each channel can supply up to 2 A from an external power source. It's excellent for small robots, rovers, or mechanical projects and works with the standard Servo and Motor library examples in the IDE.

## Adafruit Ultimate GPS Logger shield

Made by: Adafruit Industries

Price: Around \$30

Pins used: D0, D1 (Serial), D10–D13 (SPI), 5V, GND

Link: [adafruit.com/product/1272](https://adafruit.com/product/1272)

This shield uses the MTK3339 GPS module to provide accurate time and location data. It can log coordinates directly to a microSD card, making it useful for mapping, tracking, or timing projects. Although it remains available, newer GNSS modules now offer faster updates and global satellite support. Treat it as a dependable classic for older hardware.

The Adafruit Ultimate GPS logger shield is sold without attached header pins and requires soldering. You can view an in-depth tutorial on Adafruit's site ([learn.adafruit.com/adafruit-ultimate-gps-logger-shield/](https://learn.adafruit.com/adafruit-ultimate-gps-logger-shield/)), which details everything from construction of the kit to the Arduino code, along with uses for the GPS data.

## Legacy shields

Some shields are no longer widely used or supported, but they're still worth understanding. Legacy and specialty shields show how Arduino has been applied to unusual or experimental projects over the years.

Although you might not choose these for new designs, they're useful examples of what's possible with the platform.

## Adafruit FONA 800 shield — Legacy

Made by: Adafruit Industries

Price: Around \$40 (availability varies)

Pin used: D2, D3, D5–D8, D10, D11 (Serial and control lines), 5V, GND

Link: [adafruit.com/product/2468](https://adafruit.com/product/2468)

This shield adds basic cellular connectivity to your Arduino projects using a SIM800 module. It allows you to make and receive calls, send SMS messages, and connect to the Internet using GPRS. Although once a great way to create stand-alone mobile devices, 2G networks have been retired in many countries, so this shield is now considered legacy. If you need cellular data today, look for LTE-capable boards or shields that use the SIM7600 or Wio LTE modules for modern network support.

Modern alternatives: Although the FONA shield is now considered legacy, newer cellular shields work in much the same way but connect to today's 4G networks. The DFRobot SIM7600G LTE Shield **and** the Seeed Studio Wio LTE Cat.1 Shield are both good modern replacements.

They communicate using the same AT commands as the FONA, so most of your sketches will still work with only minor changes. These newer shields support LTE Cat-M and NB-IoT networks, giving you more reliable coverage, lower power use, and much faster data speeds — all while still working over existing mobile networks.

## Libelium/Cooking-Hacks Geiger-counter shield — Legacy

Made by: Libelium (Cooking-Hacks)

Price: Around \$100 (no longer available)

Pin used: D2, D3, 5V, GND

Link: <https://hackaday.com/2011/04/17/radiation-sensor-shield-for-the-arduino/>

A unique shield that detects ionizing radiation using a Geiger–Müller tube. It outputs counts per minute and can be combined with displays or SD logging to track background radiation levels. It's difficult to source today but remains an interesting reminder of how versatile Arduino shields can be.



WARNING

This shield uses Geiger tubes that operate at dangerously high voltages (400V–1000V), so it requires extreme care. It is best to keep the radiation sensor board in an enclosure to keep it out of human contact. Radiation is dangerous, but so is electricity. If you don't know what you're doing, don't mess around.

## Staying current

New shields, boards, and accessories are released all the time, so it pays to check what's available every few months. Keeping up to date not only helps you discover new hardware but also ensures you're using the latest, best-supported versions of your favorite components.

You can explore the latest products through these reliable online stores. It's a bit like browsing a DIY shop — you never know what new idea might jump out at you:

- » **Arduino Store** ([store.arduino.cc](http://store.arduino.cc))
- » **Adafruit** ([adafruit.com](http://adafruit.com))
- » **SparkFun Electronics** ([sparkfun.com](http://sparkfun.com))
- » **Seeed Studio** ([seeedstudio.com](http://seeedstudio.com))
- » **Pimoroni** ([shop.pimoroni.com](http://shop.pimoroni.com)) — UK-based maker store with great accessories and custom boards)

Alongside shopping, it's worth following a few key blogs and news pages. These sites highlight new projects, community builds, and updates from the Arduino team — all great sources of inspiration:

- » **Arduino Blog** ([arduino.cc/blog](http://arduino.cc/blog))
- » **Adafruit Blog** ([www.adafruit.com/blog](http://www.adafruit.com/blog))
- » **Hackaday** ([hackaday.com](http://hackaday.com))
- » **Make: Makezine** ([makezine.com](http://makezine.com))
- » **Seeed Studio Blog** ([seeedstudio.com/blog](http://seeedstudio.com/blog))
- » **SparkFun News** ([sparkfun.com/news](http://sparkfun.com/news))

Many developers also share their work through GitHub or social platforms like Reddit's **r/arduino** and the **Arduino Forum** ([forum.arduino.cc](http://forum.arduino.cc)). These communities often discover new libraries and shields long before they appear in the official store.

Finally, if you ever wonder whether a shield exists for a certain purpose, check the **Arduino Shield List** at [shieldlist.org](http://shieldlist.org). It's a community-maintained index of hundreds of shields — both classic and new — complete with pin maps and links to documentation.

## Browsing the Libraries

Basic sketches can take you a long way, but as your projects grow, you'll soon find yourself reaching for *libraries*. Libraries are collections of prewritten code that handle complex tasks — everything from controlling displays to connecting to Wi-Fi or the cloud. They save you time and prevent errors by packaging up well-tested functions you can reuse in your own sketches.

You can think of libraries as shortcuts to knowledge. Just as you'd visit a physical library to learn something new, you include libraries in your sketch to “teach” your Arduino a new skill. Adding a library gives your code instant access to proven routines for sensors, motors, communication, and much more.

Getting started with complex hardware can be daunting, but the good news is that many developers have documented their work and published libraries to make it easier for you. Each library usually comes with **example sketches**, showing how to set up the hardware and test the code. By loading and experimenting with these examples, you can get a project running quickly while learning how it works — the essence of Arduino's *learn-by-doing* approach.

In the **Arduino IDE 2.x**, libraries are managed through the **Library Manager**, found under **Sketch** ⇨ **Include Library** ⇨ **Manage Libraries**. The Library Manager lists thousands of approved and community-maintained libraries that you can install with a single click. You can filter by keyword, author, or topic, and view the latest update date to make sure a library is still being maintained.

When you install a library, the IDE automatically stores it in your user **libraries** folder, and its examples appear under **File** ⇨ **Examples** ⇨ **[Library Name]**. To use a library, include a single line at the top of your sketch, such as:

```
#include <Servo.h>
```

This tells the compiler to include the functions and definitions from that library when your sketch is compiled.

If a library you need isn't listed in the Library Manager, you can still install it manually. Choose **Sketch** ⇨ **Include Library** ⇨ **Add .ZIP Library**, select the

downloaded ZIP file, and the IDE will install it automatically. Restarting isn't usually necessary in version 2.x, but it ensures new examples appear in the menu.

For more advanced exploration, many libraries are hosted openly on **GitHub**. From there, you can read documentation, report issues, or even contribute improvements. The Arduino community thrives on sharing, so don't hesitate to learn from — or give back to — the libraries that make your projects possible.

## Reviewing the standard libraries

This section covers a selection of the libraries included with the current release of the Arduino IDE (version 2.x at the time of writing). These “standard” libraries provide core functionality that supports the most common hardware and communication tasks. They're well documented and come with ready-to-run examples, so they're great places to start learning.

You can find these libraries by choosing Sketch ⇨ Include Library ⇨ Manage Libraries from the Arduino IDE menu. Many are installed automatically when you add a new board through Boards Manager.

When you include a library, a single line appears at the top of your sketch, such as:

```
#include <EEPROM.h>
```

Before diving into the code, it's best to open an example first. You'll find examples under File ⇨ Examples, grouped by library name. Each one shows you how to connect the hardware and call the library's main functions.

Here is a brief description of what each library does:

- » **EEPROM** ([arduino.cc/en/Reference/EEPROM](http://arduino.cc/en/Reference/EEPROM)): Your Arduino has electronically erasable programmable read-only memory (EEPROM), which is permanent storage similar to the hard drive in a computer. Data stored in this location stays there even if your Arduino is powered down. Using the EEPROM library, you can read from and write to this memory.
- » **Ethernet** ([arduino.cc/en/Reference/Ethernet](http://arduino.cc/en/Reference/Ethernet)): After you have your Ethernet shield, the Ethernet library allows you to quickly and easily start talking to the Internet. When you use this library, your Arduino can act either as a server that is accessible to other devices or as a client that requests data.
- » **WiFi and WiFinINA / WiFi3** ([arduino.cc/en/Reference/WiFinINA](http://arduino.cc/en/Reference/WiFinINA)): The original WiFi library was designed for the first Arduino WiFi Shield. Modern boards, such as the Uno R4 WiFi, Nano 33 IoT, and MKR WiFi 1010, use

WiFiNINA or WiFiS3 instead. These libraries make it simple to connect to wireless networks and cloud services, send data, and host web servers. They also work smoothly with the Arduino IoT Cloud.

- » **LiquidCrystal** ([arduino.cc/en/Reference/LiquidCrystal](https://arduino.cc/en/Reference/LiquidCrystal)): The LiquidCrystal library helps your Arduino talk to most liquid crystal displays (LCDs). The library is based on the Hitachi HD44780 driver, and you can usually identify these displays by their 16-pin interface.
- » **Servo** ([arduino.cc/en/Reference/Servo](https://arduino.cc/en/Reference/Servo)): The Servo library allows you to control up to 12 servomotor on the Uno R3 (and up to 48 on the Mega). Most hobby servos turn 180 degrees, and using this library, you can specify the degree that you want your servo(s) to turn to.
- » **SD and SdFat** ([arduino.cc/en/Reference/SD](https://arduino.cc/en/Reference/SD)): The SD library allows you to read from and write to SD and microSD cards connected to your Arduino. SD cards need to use SPI to transfer data quickly, which happens on pins 11, 12, and 13. You also need to have another pin to select the SD card when it's needed.
- » **SPI** ([arduino.cc/en/Reference/SPI](https://arduino.cc/en/Reference/SPI)): The Serial Peripheral Interface (SPI) is a method of communication that allows your Arduino to communicate very quickly with one or more devices over a short distance. Example of this communication include receiving data from sensors, talking to peripherals such as an SD card reader, and communicating with another microcontroller.
- » **Wire (I<sup>2</sup>C/TWI)** ([arduino.cc/en/Reference/Wire](https://arduino.cc/en/Reference/Wire)): The Wire library allows your Arduino to communicate with I2C devices (also known as TWI, or two-wire interface). Such devices could be addressable LEDs or a Wii Nunchuk, for example.
- » **SoftwareSerial** ([arduino.cc/en/Reference/SoftwareSerial](https://arduino.cc/en/Reference/SoftwareSerial)): The SoftwareSerial library allows you to use any digital pins to send and receive serial messages instead of, or in addition to, the usual hardware pins (0 and 1). This capability is great if you want to keep the hardware pins free for communication to a computer, allowing you to have a permanent debug connection to your project while still being able to upload new sketches or to send duplicate data to multiple serial devices.
- » **Stepper** ([arduino.cc/en/Reference/Stepper](https://arduino.cc/en/Reference/Stepper)): The Stepper library allows you to control stepper motors from your Arduino. This code also requires the appropriate hardware to work, so make sure to read Tom Igoe's notes on the subject at <https://www.tigoe.net/pcomp/code/circuits/motors/stepper-motors/>.

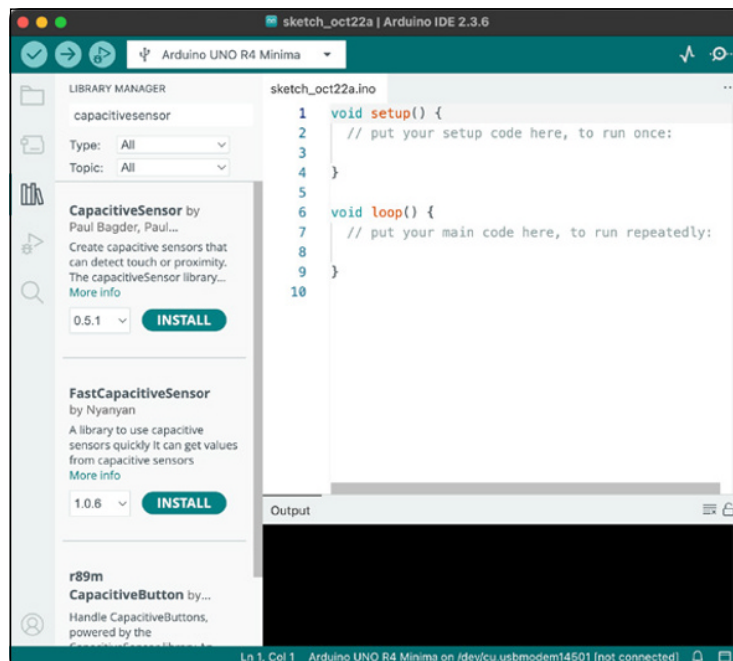
» **Firmata** ([arduino.cc/en/Reference/Firmata](http://arduino.cc/en/Reference/Firmata)): Firmata is one way to control your Arduino from software on a computer. It is a standard communication protocol, so you can use the library to allow easy communication between hardware and software rather than write your own communication software.

These core libraries form the backbone of most sketches. They're a great place to start exploring — try out their examples, modify the code, and see what happens. When you're comfortable with them, you can expand your toolbox with additional libraries for sensors, displays, communication modules, and more.

## Installing additional libraries

The Arduino IDE comes with a handful of useful libraries, but thousands more are available for different sensors, displays, communication modules, and creative projects. Some libraries are official, others are written and maintained by the community — all are there to help you extend what your Arduino can do.

The easiest way to add new libraries is through the **Library Manager** (shown in Figure 12-3). From the Arduino IDE, choose **Sketch** ⇄ **Include Library** ⇄ **Manage Libraries**. A window opens with a searchable list of every library indexed by Arduino. You can search by name, keyword, or author, and click **Install** to add it instantly. When installed, it will appear in your **Include Library** menu, ready to use.



**FIGURE 12-3:** The Arduino Library Manager shows a searchable list of libraries.

Each library in the Manager also lists its **version number** and **last update date**. This is a good way to check whether it's actively maintained — libraries that are updated regularly are more likely to work with the latest Arduino boards and compiler versions.

## Installing a library from a ZIP file

Not all libraries appear in the Library Manager. Sometimes developers share them directly from **GitHub** or their own websites. In these cases, the library is usually provided as a ZIP file — for example, `CapSense.zip` or `TimerOne.zip`.

To install a ZIP library in Arduino IDE 2.x:

1. **Download the ZIP file to your computer.**
2. **In the IDE, choose Sketch ⇨ Include Library ⇨ Add .ZIP Library.**
3. **Select the ZIP file you downloaded.**

The IDE automatically extracts and installs the library into your **Documents/Arduino/libraries** folder.

If the library includes example sketches, you'll find them under **File ⇨ Examples ⇨ [Library Name]**. In IDE 2.x, you usually don't need to restart the IDE for new libraries to appear.

Inside each library folder, you'll see at least two files ending in `.h` and `.cpp`. These define the library's **header** and **source code**. You might also find a `keywords.txt` file (for IDE highlighting), a `library.properties` file (for metadata), and an **Examples** folder.

If you ever need to uninstall a library, open **Library Manager**, find it in the list, and click the **Remove** button. You can also delete its folder manually from the `libraries` directory.



TIP

When downloading libraries from GitHub, use the **green “Code” button** → **Download ZIP** option. Don't copy the repository's folder name directly into your libraries folder — GitHub adds extra text (like `-main` or `-master`) that can prevent the IDE from recognizing it. Rename the folder to match the library name before installing, if necessary.

## Further reading

With the Library Manager, discovering and installing new features for your Arduino is now easier than ever. You can browse community-made libraries, try out examples, and even explore the source code on GitHub to see how things work behind the scenes.

Whether you're using a readymade library like FastLED or writing your own, this open-source ecosystem is what keeps Arduino growing and evolving.

Experiment freely — every library you explore adds another skill to your Arduino's toolkit. If you're keen to understand libraries more and maybe even write your own, check out the introduction to writing your own libraries on the Arduino GitHub page at <https://github.com/arduino/library-registry/>.



# 4

## Sussing Out Software

## **IN THIS PART . . .**

Discover how to connect your Arduino projects with software on your computer to create interactive graphics, sound, and data visualizations.

Learn how to use information from your computer or the Internet to control lights, sound, and motion in the physical world.

- » Getting excited about Processing
- » Making shapes of all sizes and colors

## Chapter **13**

# Getting to Know Processing

In the previous chapters, you learn all about using Arduino as a stand-alone device. A program is uploaded to the Arduino and carries out its task ad infinitum, until it is told to stop or powered down. You are affecting the Arduino by simple, clear electrical signals, and as long as no outside influences or coding errors exist and if the components last, the Arduino reliably repeats its function. This simplicity is useful for many applications and allows the Arduino to not only serve as a great prototyping platform but also work as a reliable tool for interactive products and installations for many years, as it already does in many museums.

Although this simplicity is something to admire, many applications are outside the scope of an Arduino's capabilities. Although the Arduino is basically a computer, it's not capable of running comparably large and complex computer programs in the same way as your desktop or laptop. Many of these programs are highly specialized depending on the task you're doing. You could benefit hugely if only you could link this software to the physical world in the same way your Arduino can.

Because the Arduino can connect to your computer and be monitored over the serial port, other programs may also be able to do this, in the same way that your computer talks to printers, scanners, or cameras. So by combining the physical world interaction capabilities of your Arduino with the data-crunching software capabilities of your computer, you can create projects with an enormous variety of inputs, outputs, and processes.

Many specific programs are made for specific tasks, but until you want to specify, it's best to find software that you can experiment with — that is, be a jack-of-all-trades in the same way that your Arduino is for the physical world. Processing is a great place to start.

In this chapter, you learn about Processing, the sister project that was in the first stages of development around the same time as Arduino. Processing is a software environment that you can use to sketch programs quickly, in the same way that you use an Arduino to test circuits quickly. Processing is a great piece of open source software, and its similarities to Arduino make it easy to learn.

As of 2025, *Processing* remains one of the most approachable tools for exploring this type of interaction between software and hardware. The latest release, *Processing 4.x*, runs on Windows, macOS, and Linux and is actively maintained by the Processing Foundation. If you prefer working in the browser, the related *p5.js* project brings Processing's concepts to JavaScript and web platforms, and a Python mode is also available for those who prefer Python syntax.

## Looking Under the Hood

An Arduino can communicate over its serial port as a serial device, which can be read by any program that can talk serial. Many programs are available, but Processing is one of the most popular.

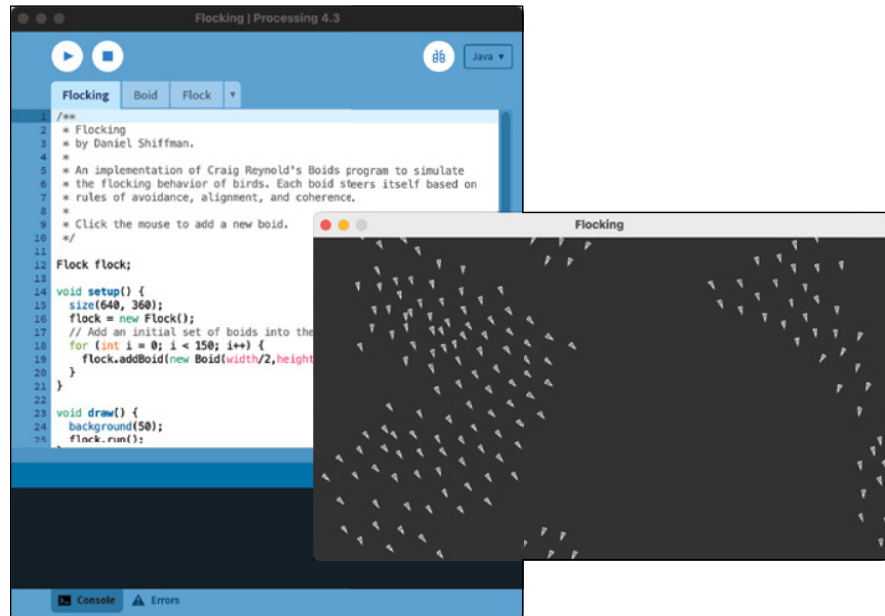
Processing has an enormous breadth of applications ranging from visualizing data to creating generative artwork to performing motion capture using your webcam for digital performances. These are just a few niches; you can find a wealth of examples at [processing.org/examples](https://processing.org/examples).

Processing is written in a Java-based language that looks similar to C (on which Arduino code is based) and C++. It is available for Windows, macOS, and Linux. Ben Fry and Casey Reas developed Processing to allow anyone, not just developers and engineers, to experiment with code. In the same way that ideas are sketched out, Processing is designed to sketch software. Programs can be quickly developed and adapted without a huge investment of time.

Processing's underlying language and interface remain the same in version 4, though its integration with modern operating systems and graphics libraries has been updated for better performance and cross-platform compatibility.

Processing uses a text-based IDE (integrated development environment) similar to that of Arduino. (In fact, it was “borrowed” by the Arduino team when the Arduino IDE was in development.) A window displays the sketch window that the

code creates, as shown in Figure 13-1. As with Arduino, the strength of Processing is the vast community that shares and comments on sketches, allowing the many participants to benefit from a diverse array of creative applications Processing is open source and allows users to modify the software as well as use it.



**FIGURE 13-1:**  
A typical view of  
Processing.

In this chapter, you learn how to get started with Processing. For more information, head over to the Processing site at [processing.org](http://processing.org).



**TIP**

Many other software tools can interface with Arduino. You can find a few of the most popular creative coding options in the sidebar “**Exploring Other Creative Coding Tools**,” or browse the Arduino Playground for a wider list at [arduino.cc/playground/main/interfacing](http://arduino.cc/playground/main/interfacing).

## Exploring Other Creative Coding Tools

Processing is one of the best places to start exploring the link between hardware and software, but it isn’t the only one. A few other tools also make it easy to connect creative code to the physical world.

- » **p5.js:** p5.js is the web-based cousin of Processing. It runs entirely in your browser using JavaScript and shares almost the same syntax, making it a natural next step after Processing. You can use the browser’s **Web Serial API**

to communicate directly with Arduino, or libraries such as **p5.serialport** for more advanced control. It's great for interactive installations, data visualization, or quick online prototypes.

Find out more at <https://p5js.org/tutorials/or github.com/p5-serial/p5.serialport>.

- » **Max/Pure Data (Pd):** Max is a visual programming environment for music, audio, and multimedia that lets you connect functional blocks instead of writing code. Its open-source sibling, **Pure Data (Pd)**, works in the same way. Both can interface with Arduino through serial communication or the **Maxuino** and **Pduino** libraries. They're popular in performance, sound design, and interactive art.

Learn more at [cycling74.com/products/max](http://cycling74.com/products/max), [puredata.info](http://puredata.info), and [maxuino.org](http://maxuino.org).

- » **TouchDesigner:** TouchDesigner is a node-based visual programming environment for creating interactive visuals and installations. It's widely used in live performance, projection mapping, and stage design. TouchDesigner communicates with Arduino via **serial**, **OSC**, or **MIDI**, letting you control graphics, lights, or sound in real time.

For tutorials, see [derivative.ca/community/forum](http://derivative.ca/community/forum) and [derivative.ca/learn](http://derivative.ca/learn).

- » **Unity:** Unity is best known as a game engine, but it's also used for simulations, VR/AR, and interactive experiences. Unity can communicate with Arduino via **serial**, **TCP/IP**, or plugins such as **Ardity** and **Uduino**. It's ideal for projects where you want hardware to interact with 3D environments or immersive interfaces.

Explore more at [unity.com/learn](http://unity.com/learn) and [github.com/dwilches/Ardity](http://github.com/dwilches/Ardity).

Each of these tools builds on the same principle as Processing — using code to turn ideas into interactive experiences that bridge the virtual and physical worlds.

## Installing Processing

Processing is free to download from [processing.org/download/](http://processing.org/download/) and runs on macOS, Windows and Linux. As of 2025, the current release is Processing 4.x, maintained by the Processing Foundation. The same package works across all platforms.

To install Processing:

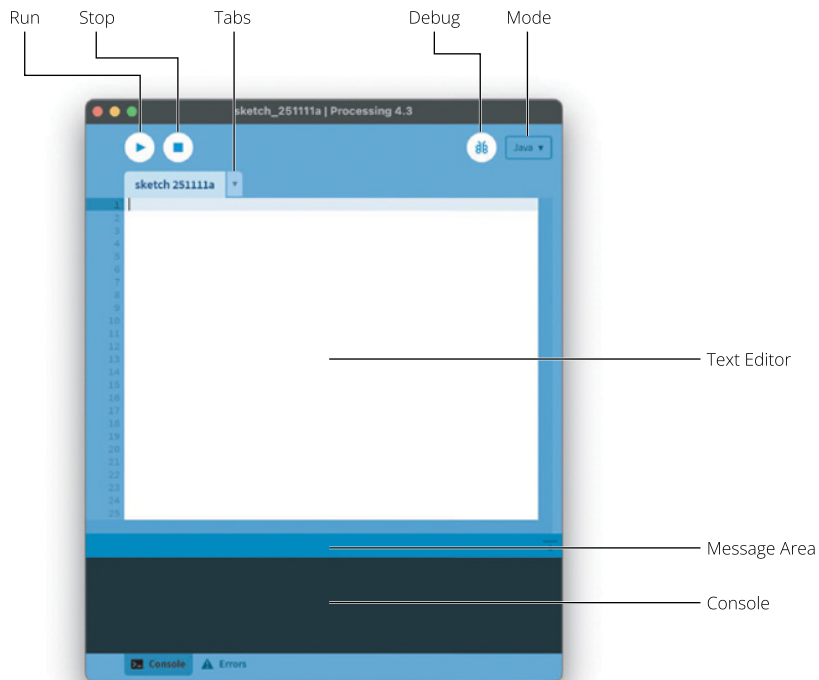
- » **On a Mac:** The ZIP file unzips automatically, revealing the Processing app, which you can then drag to the Applications folder. From there you can drag Processing to the dock for easy access or create a desktop alias.

- » **On Windows:** Unzip the ZIP file and place the Processing folder on your desktop or in a sensible location such as your Program Files folder: C:\Program Files\Processing/. Create a shortcut to Processing.exe and place it somewhere convenient, such as on your desktop or in the Start menu.

## Taking a look at Processing

After you have installed Processing, run the application. Processing opens with a blank sketch (see Figure 13-2) similar to the Arduino window, divided into five main areas:

- » Toolbar with buttons
- » Tabs
- » Text editor
- » Message area
- » Console



**FIGURE 13-2:** The Processing application is similar to but different from the Arduino one.

The blank sketch also contains a menu bar for the main Processing application, which gives you drop-down menus to access the preferences of the processing application, load recent sketches and import libraries, and perform many other functions.

Here's an overview of the Processing toolbar:

- » **Run:** Executes or runs the code in the text editor as a sketch window. The keyboard shortcuts for this command are Ctrl+R for Windows and Cmd+R for macOS.
- » **Stop:** Stops the code from running and closes the sketch window.
- » **Debug:** A tool to help you identify errors and debug your code.
- » **Mode:** Changes the programming mode between Java (default), Python, and p5.js (for web projects). You can explore more modes by clicking Mode ⇨ Manage Modes from the toolbar. For more details, visit [processing.org/reference/modes](http://processing.org/reference/modes).
- » **Tabs:** Organizes multiple files in a Processing sketch. Use tabs in larger programs to separate objects from the main sketch or to incorporate look-up tables of data into a sketch.
- » **Text editor:** Enters code into the sketch. Recognized terms or functions are highlighted in appropriate colors for clarity. The text editor is the same as that in the Arduino IDE.
- » **Message area:** Displays errors, feedback, or information about the current task. You might see a notification that the sketch was saved successfully, but more often than not, the message shows where errors are flagged.
- » **Console:** Displays more details on your sketch. You can use the `println()` function here to display the values in your sketch; additional detail on errors is also shown.

## Trying Your First Processing Sketch

Unlike with Arduino, you don't need an extra kit to get going with Processing. This feature makes Processing useful for learning about coding because you can enter a line or two of code, click Run, and see the results.

Start your first sketch with these steps:

1. Press **Ctrl+N** (in Windows) or **Cmd+N** (on a Mac) to open a new sketch.

2. Click in the text editor and enter this line of code:

```
ellipse(50,50,10,10);
```

3. Click the **Run** button.

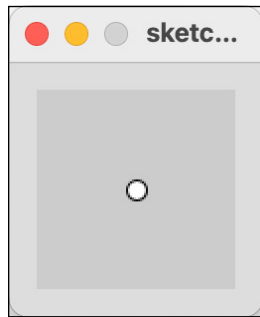
A new sketch window opens, showing a white circle in the middle of a gray box, as in Figure 13-3.



REMEMBER

The Processing 4 text editor may use slightly different colors for highlighting functions, but the example behaves the same way.

**FIGURE 13-3:**  
A Processing sketch that draws an ellipse with equal dimensions, also known as a circle.



Well done! You've just written your first Processing program.

Have you finished admiring your circle? That line of code draws an ellipse. An ellipse normally is not circular, but you gave it the parameters to make a circle.

The word *ellipse* is highlighted in turquoise in the text editor, indicating that it is a recognized function. The first two numbers are the coordinates of the ellipse, which in this case are 50, 50. The unit of the numbers is in pixels. Because the default window is 100 × 100 pixels, coordinates of 50, 50 put the ellipse in the center. The 10, 10 values indicate the width and height of the ellipse, giving you a circle. You could write the function also as

```
ellipse(x,y,width,height)
```

The coordinates for the ellipse (or any shape or point, for that matter) are written as *x* and *y*. These indicate a point in two-dimensional (2D) space, which in this case is a point measured in pixels on your screen. Horizontal positions are referred

to as the x coordinate; vertical positions are the y coordinate. Depth used in 3D space is referred to as z. Add the following line of code, just above the `ellipse()` statement:

```
size(300,200);
```

Click the Run button and you get a rectangular window with the ellipse in the top left, as shown in Figure 13-4. The `size()` function is used to define the size of the sketch window in pixels, which in this case is 300 pixels wide and 200 pixels high. If your screen isn't like Figure 13-4, you may have put the statements in the wrong order. The lines of code are read in order, so if the `ellipse` code is first, the blank window is drawn over the ellipse. And with a rectangular window, you see that the coordinates are measured from the top left.

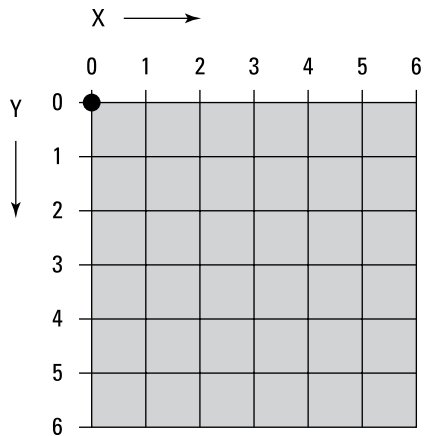


**FIGURE 13-4:**  
A resized  
display window.

Coordinates are measured on an invisible grid with the center point at 0, 0 for 2D (or 0, 0, 0 for 3D), which is referred to as the *origin*. This way of referencing locations is based on the Cartesian coordinate system, which you may have studied in school. Numbers can be positive or negative, depending on which side of the origin they are on. On computer screens, the origin is at the top left because pixels are drawn from top left to bottom right, one row at a time (check out Figure 13-5). Therefore, the statement `size(300,200)` draws a window 300 pixels from left to right on the screen and then 200 pixels from top to bottom.

## Drawing shapes

To gain a better understanding of the possibilities you have in drawing shapes, look at a few basic shapes: point, line, rectangle, and ellipse.

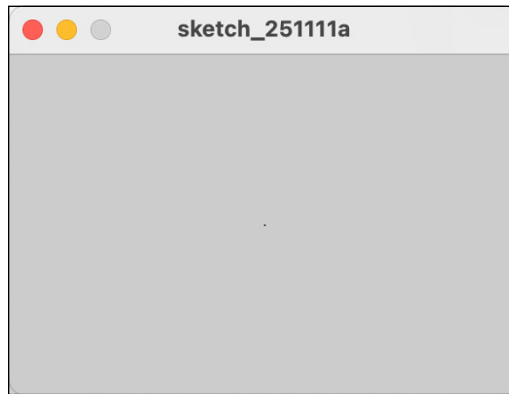


**FIGURE 13-5:**  
How the grid  
looks on  
a computer.

## Point

A single point is the most basic shape and is useful for creating more complex shapes. Write the following code and then click the Run button. Look closely and you'll see a single black pixel in the center of the display window (see Figure 13-6). That is the point that your code drew.

```
size(300,200);
point(150,100);
```



**FIGURE 13-6:**  
If you look  
closely, you can  
see the point.

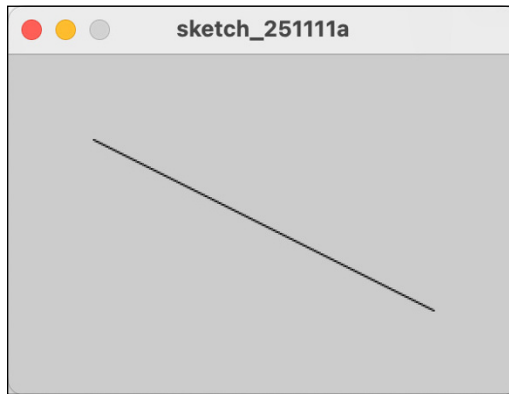
The `point` function can also be written as follows:

```
point(x,y);
```

## Line

A line is made by connecting two points, which is done by defining the start and end points. Write the code to generate a screen like the one in Figure 13-7:

```
size(300,200);  
line(50,50,250,150);
```



**FIGURE 13-7:**  
A line between  
two points.

You can also write a line written as follows:

```
line(x1,y1,x2,y2);
```

## Rectangle

You can draw a rectangle a number of different ways. In this first example, a rectangle is drawn by identifying the starting point and then the width and height of the rectangle. Write the following code to draw a rectangle in the center of your display window:

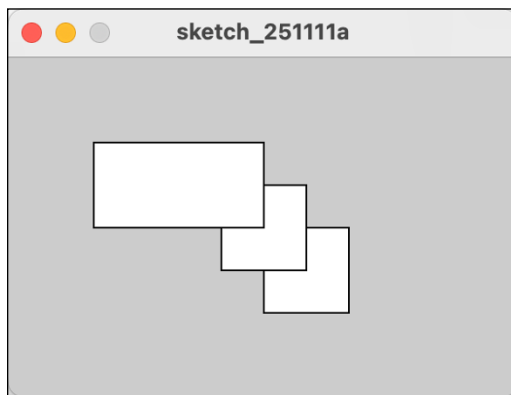
```
size(300,200);  
rect(150,100,50,50);
```

In this case, you have a rectangle that starts in at point 150,100 in the center of the display window. That is the top-left corner of the rectangle, and from there it has a width of 50, which extends the rectangle toward the right of the window, and a height of 50, which extends toward the bottom of the window. This function is particularly useful if you want the size of the rectangle to remain constant but change the position of the rectangle. You could also write the code as follows:

```
rect(x,y,width,height);
```

When drawing rectangles, you can choose among different modes (see Figure 13-8). If the mode is set to center, the rectangle is drawn centered around a point instead of being drawn from that point. Write the following code and you see that the same values display a different position when `rectMode` is changed to `CENTER`:

```
rectMode(CENTER);  
size(300,200);  
rect(150,100,50,50);
```



**FIGURE 13-8:**  
A selection of  
differently drawn  
rectangles.

You can see that the rectangle is now centered in the display window. The shape extends equally from the center point both left to right and top to bottom.

You can also draw a rectangle by declaring two diagonally opposite corners. Write the following code, this time with `rectMode` set to `CORNERS`:

```
rectMode(CORNERS);  
size(300,200);  
rect(150,100,50,50);
```

You see a rectangle that is quite different from the others because it starts at the same point in the center, `150,100`, but ends at point `50,50`, creating a rectangle positioned above and to the left of the starting point. You can also write the code as follows:

```
rect(x1,y1,x2,y2);
```

## Ellipse

The first item covered in this chapter was `ellipse`, which can be used to simply draw an ellipse. Write the following code to draw an ellipse in the center of the display window:

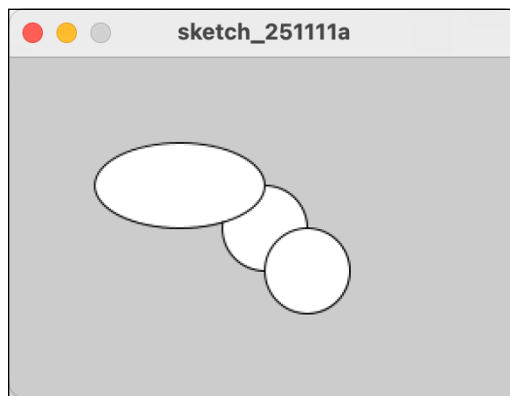
```
ellipse(150,100,50,50);
```

The default mode for `ellipse` is `CENTER`, whereas the default mode for `rect` is `CORNER`. The preceding line of code can also be written as follows:

```
ellipse(x,y,width,height);
```

As with `rectMode()` it's possible to set different modes (see Figure 13-9) for drawing ellipses using `ellipseMode()`. Write the following code to draw an ellipse from its corner instead of its center:

```
ellipseMode(CORNER);  
size(300,200);  
ellipse(150,100,50,50);
```



**FIGURE 13-9:**  
A selection of  
differently  
drawn ellipses.

This draws an ellipse starting from its top-left corner with a width of 50 and a height of 50.

It is also possible to draw an ellipse by specifying multiple corners. Write the following code to change the `ellipseMode` to `CORNERS`:

```
ellipseMode(CORNERS);  
size(300,200);  
ellipse(150,100,50,50);
```

Similarly to `rectMode(CORNERS)`, the `ellipseMode(CORNERS)` creates an ellipse positioned above and to the left of the starting point. The first corner is the center point of the sketch and the second is at point 50,50.

## Changing color and opacity

Now that you have an understanding of shapes, it's time to affect their appearance. The simplest way to change a shape's appearance is to change the color. By using the `background` and `fill` functions, you can change the color of the shapes on your screen to any one of 16,777,216 colors. You can also set the opacity of the objects you draw, allowing you to mix colors by layering semitransparent shapes.

### Background

The `background` function changes the background of your sketch. You can choose grayscale values or color. You'll start by changing grayscale values.

Open a new sketch, and type the following code to change the default gray window to black:

```
background(0);
```

Change 0 to 255 to change the color to white:

```
background(255);
```

Any value between 0 (black) and 255 (white) is a grayscale value. The reason that this range is 0 to 255 is that there are 8 bits of data in a byte, meaning that you need one byte to store a grayscale color value.

## MAKING SENSE OF BINARY, BITS, AND BYTES

The binary number system, also known as base-2, uses only two values: 0 or 1. Decimal numbers use 0 to 9 and are usually referred to as base-10. Hexadecimal numbers use 0 to 9 and A to F and are referred to as base-16.

But how is binary useful when you're trying to talk to lots of things and have only two options? The answer is that you use a lot of binary values.

*(continued)*

(continued)

For example, if you have a base-2 binary number such as 10101101, you can determine its value in base-10 with a simple lookup table. Binary is typically read from right to left. Because binary is base-2, each value is the binary value multiplied by 2 to the power of ( $2^x$ ), where  $x$  is equal to the order of the bit, starting at 0 on the right. For example, as shown in the following, the fourth binary value is equal to  $1 \times (2 \times 2 \times 2) = 8$ .

Binary	1	0	1	0	1	1	0	1	
Calculation	$1 \times 2^7$	$0 \times 2^6$	$1 \times 2^5$	$0 \times 2^4$	$1 \times 2^3$	$1 \times 2^2$	$0 \times 2^1$	$1 \times 2^0$	Total
Decimal	128	0	32	0	8	4	0	1	173

As you can see, extremely large numbers can be formed using only zeros and ones. In this case, you have eight binary values with a total decimal value of 255. When talking about memory, each binary value takes one *bit* of memory, and each group of eight bits is referred to as a *byte*. To give you an idea of scale, a blank Arduino sketch uses 466 bytes; an Uno can store a maximum of 32,256 bytes, and a Mega can store a maximum of 258,048 bytes.

To liven things up a bit, you can add color to your sketch background. Instead of 8-bit grayscale, you use 24-bit color, which is 8-bit red, 8-bit green, and 8-bit blue. The color of the background is defined with three values instead of one:

```
background(200,100,0);
```

This line of code gives you an orange background, which is comprised of a red value of 200, a green value of 100, and a blue value of 0. There are several color modes, but in this case this line of code can be interpreted as follows:

```
background(red,green,blue);
```

## Fill

Want to change the color of the shapes you draw? Use `fill` to both set color and control the shape's opacity.

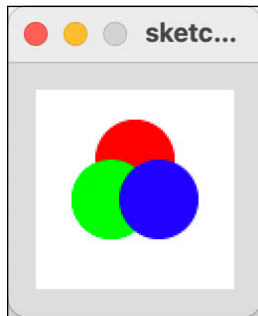
`fill` sets the color for any shape that is drawn after it. By calling `fill` multiple times, you can change the color of several different shapes. Write the following code to draw three ellipses with different colors, as shown in Figure 13-10:

```
background(255);  
noStroke();
```

```
// Bright red
fill(255,0,0);
ellipse(50,35,40,40);

// Bright green
fill(0,255,0);
ellipse(38,55,40,40);

// Bright blue
fill(0,0,255);
ellipse(62,55,40,40);
```



**FIGURE 13-10:**  
The different  
colored circles.

The background is set to white (255), and the `noStroke` function removes borders from the shapes. You can comment out the function by using two forward slashes (`//`), to see the effect.

It's also important to note that all shapes are drawn in the order in which they were programmed. You can see that the first circle to be drawn is red because the other two circles are layered on top of it. The red value is the highest possible (255), as is the second for green and the third for blue. If another shape were drawn at the end of the code, it would be the same strong blue because that is the last `fill` value.

You can also affect the opacity of the colors, creating semitransparent shapes. By adding a fourth value to the `fill` function, you can set the opacity from 0 (fully transparent) to 255 (solid color). Update the preceding code with the following values to give the circles transparency:

```
background(255);
noStroke();
```

```
// Bright red
fill(255,0,0,100);
ellipse(50,35,40,40);

// Bright green
fill(0,255,0,100);
ellipse(38,55,40,40);

// Bright blue
fill(0,0,255,100);
ellipse(62,55,40,40);
```

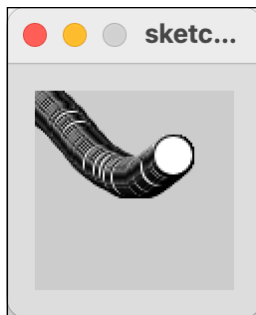
## Playing with interaction

All of this is fun but static. In the next example, you inject some life into your sketches by using your mouse as an input. To do this, you must constantly update the sketch by looping through it over and over again, sending new values for each loop. Write the following code to create an interactive sketch:

```
void setup() {
}

void draw() {
  ellipse(mouseX,mouseY,20,20);
}
```

This code draws an ellipse centered on your mouse pointer coordinates, so when you move your mouse you leave a trail of ellipses behind, as shown in Figure 13-11. The functions `mouseX` and `mouseY` are shown in pink in the text editor and take the coordinates of your mouse pointer in the display window. The values are the number of pixels horizontally and vertically, respectively.

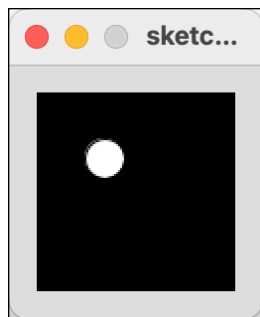


**FIGURE 13-11:**  
Drawing lots of  
ellipses wherever  
your mouse  
pointer goes.

This code might look familiar. Instead of Arduino's `void setup` and `void loop`, Processing uses `void setup` and `void draw`. These work in almost the same way: `setup` runs once at the start of the sketch; `loop` and `draw` run forever or until they are told to stop. You can stop the sketch at any time by clicking the Stop button.

Change the sketch slightly, and you can cover up all those previous ellipses to display only the most recent (see Figure 13-12).

```
void setup() {  
}  
  
void draw() {  
  background(0);  
  ellipse(mouseX, mouseY, 20, 20);  
}
```



**FIGURE 13-12:** Just one ellipse at the location of your mouse pointer.

If you're curious to go a little further, Processing can also render in 2D or 3D using the `P2D` and `P3D` renderers. Try adding a third parameter to the `size()` function, such as `size(300, 200, P2D);`, to see how lighting and drawing performance change.

There is much more to Processing that I can't cover in this book. However, these few points should be enough to gain a basic comprehension of how code relates to the onscreen visuals. You can find a wealth of examples, both on the Processing site and in the Processing software. The best approach is to run the examples and then tweak the values to see what happens. You'll learn what's going on much more quickly by experimenting — and with no electronics, you won't break anything.



#### IN THIS CHAPTER

- » Turning on a real light with a virtual switch
- » Graphing data from the physical world
- » Sending signals between the Arduino and Processing

## Chapter **14**

# Processing the Physical World

In the preceding chapter, you learn the basics of Processing and its similarities to and differences from Arduino. This chapter is all about combining both tools to integrate the virtual and physical worlds. These few exercises teach you the basics about sending and receiving data in both Processing and Arduino. You can build on this knowledge to create your own projects, maybe to generate some awesome onscreen visuals from your sensors or to turn on a light every time someone mentions you on Twitter.

## Making a Virtual Button

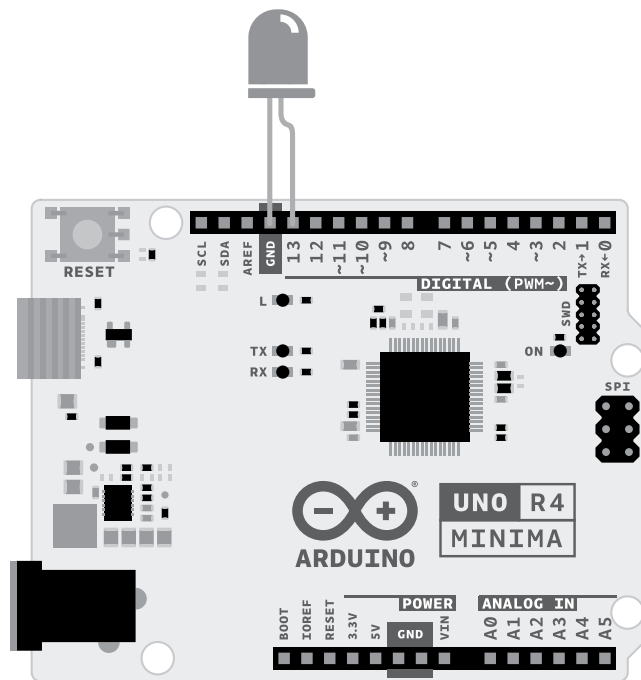
In this example, you learn how to make an onscreen button in Processing that affects a physical LED on your Arduino. This is a great sketch to get started with interactions between computers and the real world, and between an Arduino and Processing.

You need the following:

- » An Arduino Uno
- » An LED

The setup is simple for this introduction to Arduino and Processing, requiring only a single LED.

As shown in Figures 14-1 and 14-2, insert the long leg of the LED into pin 13 and the short leg into GND. If you don't have an LED, you can simply monitor the onboard LED marked L.

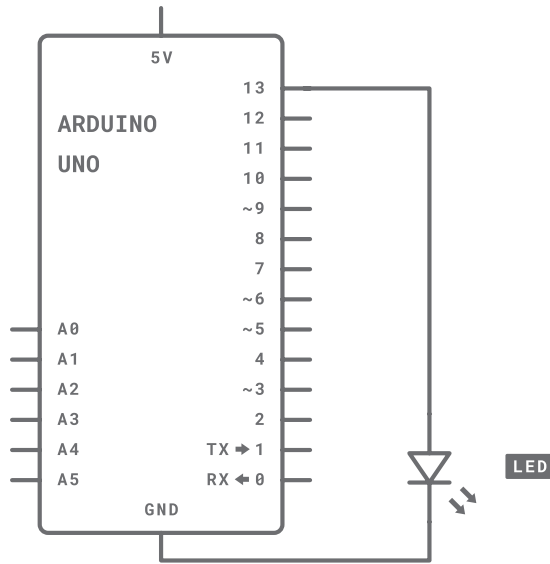


**FIGURE 14-1:**  
A circuit diagram  
of an Arduino  
Uno with an LED  
connected  
to pin 13.

## Setting up the Arduino code

After your circuit is assembled, you need the appropriate software to use it. From the Arduino menu, choose File ⇨ Examples ⇨ 04.Communication ⇨ PhysicalPixel to open the sketch. In Arduino IDE 2.x the path is the same, but the Examples panel appears in the left sidebar.

**FIGURE 14-2:**  
A schematic of an  
Arduino Uno with  
an LED connected  
to pin 13.



This sketch contains both Arduino code and the relevant Processing code for the sketch to work. (It also has a variation in Max 5.) The code below the Arduino code is commented out to avoid interfering with the Arduino code.



In early versions of the Arduino software, sketches used the .pde suffix (like Processing). Today, all Arduino sketches use .ino files, so if you open an old .pde file, the IDE automatically converts it.

```

/*
Physical Pixel

An example of using the Arduino board to receive data from the computer. In
this case, the Arduino boards turns on an LED when it receives the character
'H', and turns off the LED when it receives the character 'L'.

The data can be sent from the Arduino Serial Monitor, or another program like
Processing (see code below), Flash (via a serial-net proxy), PD, or Max/MSP.

The circuit:
- LED connected from digital pin 13 to ground through 220 ohm resistor

created 2006
by David A. Mellis
modified 30 Aug 2011
by Tom Igoe and Scott Fitzgerald

```

This example code is in the public domain.

```
https://docs.arduino.cc/built-in-examples/communication/PhysicalPixel/  
*/  
  
const int ledPin = 13; // the pin that the LED is attached to  
int incomingByte;      // a variable to read incoming serial data into  
  
void setup() {  
  // initialize serial communication:  
  Serial.begin(9600);  
  // initialize the LED pin as an output:  
  pinMode(ledPin, OUTPUT);  
}  
  
void loop() {  
  // see if there's incoming serial data:  
  if (Serial.available() > 0) {  
    // read the oldest byte in the serial buffer:  
    incomingByte = Serial.read();  
    // if it's a capital H (ASCII 72), turn on the LED:  
    if (incomingByte == 'H') {  
      digitalWrite(ledPin, HIGH);  
    }  
    // if it's an L (ASCII 76) turn off the LED:  
    if (incomingByte == 'L') {  
      digitalWrite(ledPin, LOW);  
    }  
  }  
}
```

Now go through the steps to upload your sketch. On the Uno R4 Minima, select the correct board and port under **Tools** → **Board** → **Arduino UNO R4 Minima** and **Tools** → **Port** before uploading.

With the Arduino set up to receive a message from Processing, you need to set up the Processing sketch to send a signal message over the same serial port to your Arduino.

## Setting up the Processing code

The Processing code is available within multiline comment markers (`/* */`) at the bottom of the Arduino PhysicalPixel sketch. Copy the code within the comment

markers, paste it into a new Processing sketch, and save it with an appropriate name, such as PhysicalPixel:

```
// Mouse over serial

// Demonstrates how to send data to the Arduino I/O board, in order to
// turn ON a light if the mouse is over a square and turn it off
// if the mouse is not.

// created 2003-4
// based on examples by Casey Reas and Hernando Barragan
// modified 30 Aug 2011
// by Tom Igoe
// This example code is in the public domain.

import processing.serial.*;

float boxX;
float boxY;
int boxSize = 20;
boolean mouseOverBox = false;

Serial port;

void setup() {
  size(200, 200);
  boxX = width/2.0;
  boxY = height/2.0;
  rectMode(RADIUS);

  // List all the available serial ports in the output pane.
  // You will need to choose the port that the Arduino board is
  // connected to from this list. The first port in the list is
  // port #0 and the third port in the list is port #2.
  // if using Processing 2.1 or later, use Serial.printArray()
  println(Serial.list());

  // Open the port that the Arduino board is connected to (in this case #0)
  // Make sure to open the port at the same speed Arduino is using (9600bps)
  port = new Serial(this, Serial.list()[0], 9600);
}

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

```

// Test if the cursor is over the box
if (mouseX > boxX-boxSize && mouseX < boxX+boxSize &&
mouseY > boxY-boxSize && mouseY < boxY+boxSize) {
  mouseOverBox = true;
  // draw a line around the box and change its color:
  stroke(255);
  fill(153);
  // send an 'H' to indicate mouse is over square:
  port.write('H');
}
else {
  // return the box to its inactive state:
  stroke(153);
  fill(153);
  // send an 'L' to turn the LED off:
  port.write('L');
  mouseOverBox = false;
}

// Draw the box
rect(boxX, boxY, boxSize, boxSize);
}

```



TECHNICAL  
STUFF

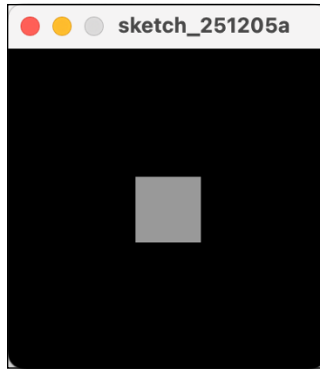
In Processing 4 or later, you can use `Serial.printArray()` instead of `println(Serial.list())`; to see available serial ports listed more clearly. Both methods work the same way.

When you run this sketch for the first time on macOS, Processing may ask for permission to access the serial port; grant access so it can communicate with your Arduino.

Click the Run button to run the Processing sketch. A window appears with a black background and a gray square in the middle, representing your virtual button (shown in Figure 14-3). If you move your mouse cursor over the gray square (or pixel), you can see that its edges turn white. If you then look at your Arduino, you see that whenever your mouse cursor hovers over the gray square, the LED on your board illuminates, giving you a physical representation of your pixel.

If your LED doesn't light, double-check your wiring:

- » Make sure that you're using the correct pin number.
- » Make sure that the LED legs are inserted correctly.
- » Check that you're using the correct serial port.



**FIGURE 14-3:**  
Your Processing sketch window displaying the virtual pixel.

- » Check that your Arduino code uploaded correctly and that your Processing code has no errors. You cannot upload while the Processing sketch is communicating with your Arduino, so stop the Processing sketch first.

## Understanding the Processing PhysicalPixel sketch

To understand how this works, it helps to break the project into its two parts — the Processing input and the Arduino output. We'll start with Processing.

The structure of a Processing sketch is similar to Arduino. You include libraries and declare variables at the start of the sketch and set fixed values or initializations in `setup`. The `draw` function then repeats its process until told otherwise.

Processing uses libraries to add functionality in the same way as Arduino does. In this case, a serial communication library is needed to talk to Arduino. In Arduino, this library is included by using `#include <libraryName.h>`. However, in Processing, you use the `import` keyword, followed by the name and the syntax `*` to load all the related parts of that library:

```
import processing.serial.*;
```

A *float* is a floating-point number, one with a decimal place, such as 0.5 or 10.9. In this case, two floating-point numbers are declared, `boxX` and `boxY`. These are the coordinates for the location of the box:

```
float boxX;  
float boxY;
```

Next, `boxSize` defines the size of the box as an integer, or whole number. Because it is square, only one value is needed:

```
int boxSize = 20;
```

A Boolean (which can be only `true` or `false`) is used to communicate that the mouse cursor is over the box. `boolean` is set to start as `false`:

```
boolean mouseOverBox = false;
```

The last thing to do is create a new `Serial` port object. Many serial connections could be in use by your computer, so it's important that each one be named so that it can be used as needed. In this case, you are using only one port. The word *serial* is specific to the serial library to indicate that you want to create a new serial object (connection), and the word *port* is the name of the object (connection) used to refer to the port from this point on. You can think of this as labelling a specific serial connection so you can refer to it later.

```
Serial port;
```

In `setup`, the first item to define is the size of the display window, which is set to 200 pixels square:

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

The variables for `boxX` and `boxY` are set to be proportional to the width and height, respectively, of the display window. They are always equal to half the width and height. Next, `rectMode` is set to `RADIUS`, which is similar to `CENTER`, but instead of specifying the overall width and height of the rectangle, `RADIUS` specifies half the height and width. (`CENTER` could be interpreted as diameter in that respect.) Because the coordinates of the box are centered and are aligned to the center point of the display window, the box is also perfectly centered:

```
boxX = width/2.0;  
boxY = height/2.0;  
rectMode(RADIUS);
```

Your computer may have a lot of serial connections, so it's best to print a list of them to locate your Arduino:

```
println(Serial.list());
```

The most recent port usually appears at the top of this list in position 0, so if you've just plugged in your Arduino, the first item is likely the one you want. If you're not using the `Serial.list` function, you could replace `Serial.list()[0]` with another number in the list, which will be printed on the console. You can also replace `Serial.list()[0]` with the name of the port, such as `/dev/tty.usbmodem26221` or `COM5`. Specifying the name is also useful if you have multiple Arduinos connected to the same computer. The number 9600 refers to the baud rate, which is the rate at which you're communicating with the Arduino.

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



REMEMBER

If the baud rate number is not the same on both the sending and receiving ends, the data will not be received.

In draw, the first task is to draw a black background:

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

Processing uses the same (or similar) conditionals as Arduino. This `if` statement tests the mouse value to see whether the cursor is over the box area. If `mouseX` is greater than the box coordinate (center) minus the size of the box (half the box width), and less than the box coordinate (center) plus the size of the box (half the box width), the horizontal position is over the box. This statement is used again with the vertical position, using `AND` statements (`&&`) to add to the conditions of the `if` statement. Only if all these are `true` can the Boolean `mouseOverBox` be declared `true`:

```
// Test if the cursor is over the box
if (mouseX > boxX-boxSize && mouseX < boxX+boxSize &&
    mouseY > boxY-boxSize && mouseY < boxY+boxSize) {
    mouseOverBox = true;
}
```

To indicate that `mouseOverBox` is `true`, the code draws a white line around the box. Rather than requiring that another box be drawn, the white line appears

simply by changing the stroke, or outline, value (*stroke* is a term common in most graphics software). The stroke is set to 255, which outlines the box in white:

```
// draw a line around the box and change its color:  
stroke(255);
```

Fill is set to 153, a mid-gray, which colors the next object drawn:

```
fill(153);
```

Then the all-important communication is sent. The `port.write` statement is similar to `Serial.print` but is used for writing to a serial port in Processing. The character sent is H, for high:

```
// send an 'H' to indicate mouse is over square:  
port.write('H');  
}
```

The `else` statement tells Processing what to do if the mouse cursor is not over the box:

```
else {
```

The `stroke` value is set to the same mid-gray as the box. The box fill color remains the same whether active or inactive:

```
// return the box to its inactive state:  
stroke(153);  
fill(153);
```

The character L is sent to the serial port to signify that the LED should be set low:

```
// send an 'L' to turn the LED off:  
port.write('L');
```

The Boolean `mouseOverBox` is set to `false`:

```
mouseOverBox = false;  
}
```

Finally, the box (technically a rectangle) is drawn. Its coordinates are always centered, and its size remains the same; the only difference is the color applied by the `if` statement. If the mouse cursor is over the box, the `stroke` value is changed to white (active), and if not, the `stroke` value is set to the same gray as the box and appears to not be there (inactive):

```
// Draw the box
rect(boxX, boxY, boxSize, boxSize);
}
```

## Understanding the Arduino PhysicalPixel sketch

In the preceding section, you find out how the Processing side provides a signal. The signal is sent over the serial connection to your Arduino. On the Arduino side, the code listens for those characters from Processing and switches the LED accordingly. In this section, I explain what the Arduino code does with the signal. It's a short, clear example that forms a good test whenever you're setting up Arduino-to-Processing communication.

First, the constant and variable values are declared. The LED pin — pin 13 — is the LED output and does not change, so it is declared as a constant. The `incomingByte` value does change and is declared as an integer (`int`), not a character (`char`). I explain why a bit later.

```
const int ledPin = 13; // the pin that the LED is attached to
int incomingByte;     // a variable to read incoming serial data into
```

In `setup`, the serial communication is initialized and set to a matching baud rate of 9600.



REMEMBER

When communicating between Processing and Arduino, the baud rate — for example, 9600 bps — must match on both sides. If they don't match, you'll see garbled data or nothing at all.

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

Pin 13, or `ledPin` as it is named, is set to be an output.

```
// initialize the LED pin as an output:
pinMode(ledPin, OUTPUT);
}
```

The first action in the loop is to determine whether any data is available. `Serial.available` reads the serial buffer, which stores any data sent to the Arduino before it is read. Nothing happens until data is sent to the buffer.

By checking that the value is greater than 0, you reduce the number of readings considerably. Reading lots of 0 or null values can considerably slow down the operation of your Arduino and any programs or hardware reading from it:

```
void loop() {  
  // see if there's incoming serial data:  
  if (Serial.available() > 0) {
```

If a value is greater than 0, it is stored in the int variable `incomingByte`:

```
    // read the oldest byte in the serial buffer:  
    incomingByte = Serial.read();
```

Now you need to know if the data received is what your program is expecting. Processing sent H as a character, but that is just a byte of data that can be understood as a number or a character. In this case, you're treating it as an integer. This `if` statement checks to see whether the integer value is equal to 72, which is equal to the character H in ASCII. The quotation marks ('H') indicate that the H is a character and not a variable. The statement `if (incomingByte == 72) {` would return the same result:

```
    // if it's a capital H (ASCII 72), turn on the LED:  
    if (incomingByte == 'H') {
```

If the values are equal, pin 13 is set HIGH:

```
        digitalWrite(ledPin, HIGH);  
    }
```

If the value is the character L, or the integer value 76, the same pin is set LOW:

```
        // if it's an L (ASCII 76) turn off the LED:  
        if (incomingByte == 'L') {  
            digitalWrite(ledPin, LOW);  
        }  
    }  
}
```

This basic Processing-to-Arduino interaction works great as the basis for larger projects. In this example, the onscreen interaction is the input and could be swapped for more useful or elaborate inputs. One such input is face tracking: When your face is at the center of the screen, the signal is sent. On the Arduino side of the code, as is true of Processing, a vast array of outputs could be triggered besides lighting an LED (lovely though it is). For instance, you could link

optocouplers to a remote and begin playback whenever a high signal is sent, and you could pause playback whenever a low signal is sent. See the bonus chapter, “Hacking Other Hardware,” at [www.dummies.com/go/arduino4d](http://www.dummies.com/go/arduino4d) for more about working with optocouplers.

## Drawing a Graph

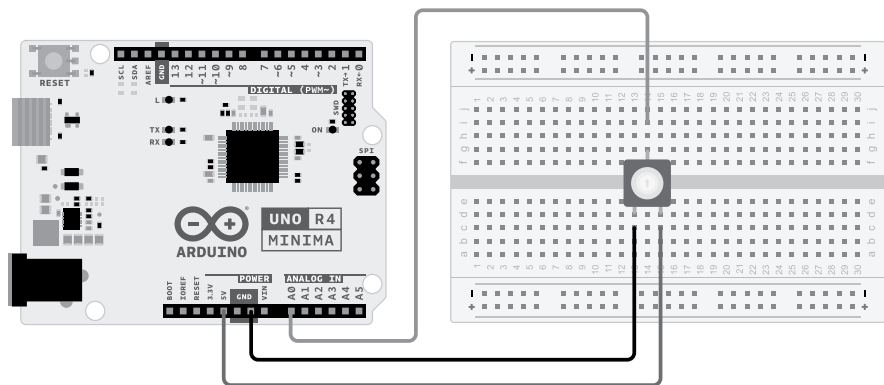
In the preceding section, you see how to send a signal in one direction. Now you’ll do the reverse — sending data from Arduino to Processing — to create a live visual graph. In this example, you find out how to read the value of a potentiometer using your Arduino and display it in a Processing sketch window.

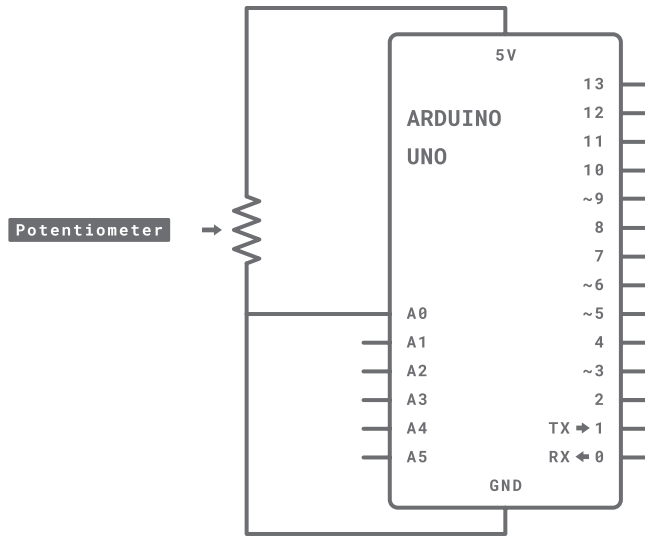
You need the following:

- » An Arduino Uno
- » A breadboard
- » A potentiometer
- » Jump wires

This basic circuit uses a potentiometer to send an analog value to Processing that can be interpreted and displayed on an onscreen graph. Assemble the circuit by connecting the center pin of the potentiometer to analog pin 0, following Figures 14-4 and 14-5. The potentiometer is wired with the central pin connected to analog pin 0. Of the other two pins, one is connected to 5V and the other to GND. You can reverse the outer pins if you want the direction of increase to be reversed when turning the knob.

**FIGURE 14-4:**  
A circuit diagram for a potentiometer input.





**FIGURE 14-5:**  
A schematic for a potentiometer input.

## Setting up the Arduino code

After you assemble your circuit, you need the appropriate software to use it. From the Arduino menu, choose File ⇨ Examples ⇨ 04.Communication ⇨ Graph to open the sketch. This sketch contains both Arduino code and the relevant Processing code for the sketch to work (and has a variation in Max 5 as well). The Processing code below the Arduino code is commented out to avoid interfering with the Arduino code.

```

/*
  Graph

  A simple example of communication from the Arduino board to the computer:
  the value of analog input 0 is sent out the serial port. We call this "serial"
  communication because the connection appears to both the Arduino and the
  computer as a serial port, even though it may actually use
  a USB cable. Bytes are sent one after another (serially) from the Arduino
  to the computer.

  You can use the Arduino Serial Monitor to view the sent data, or it can
  be read by Processing, PD, Max/MSP, or any other program capable of reading
  data from a serial port. The Processing code below graphs the data received
  so you can see the value of the analog input changing over time.

  The circuit:
  - any analog input sensor attached to analog in pin 0.

```

```
created 2006
by David A. Mellis
modified 9 Apr 2012
by Tom Igoe and Scott Fitzgerald
```

This example code is in the public domain.

```
https://docs.arduino.cc/built-in-examples/communication/Graph/
*/

void setup() {
  // initialize the serial communication:
  Serial.begin(9600);
}

void loop() {
  // send the value of analog input 0:
  Serial.println(analogRead(A0));
  // wait a bit for the analog-to-digital converter
  // to stabilize after the last reading:
  delay(2);
}
```

Now go through the steps to upload your sketch.

With the Arduino now set up to send a message to Processing, you need to set up the Processing sketch to receive that message over the serial port.

## Setting up the Processing code

The Processing code is found within multiline comments markers (`/* */`) at the bottom of the Arduino Graph sketch. Copy the code within the comment markers and then paste it into a new Processing sketch, saved with an appropriate name, such as Graph.

```
/* Graphing sketch

// This program takes ASCII-encoded strings from the serial port at 9600 baud
// and graphs them. It expects values in the range 0 to 1023, followed by a
// newline, or newline and carriage return

// Created 20 Apr 2005
// updated 24 Nov 2015
```

```

// by Tom Igoe
// This example code is in the public domain.

import processing.serial.*;

Serial myPort;          // The serial port
int xPos = 1;          // horizontal position of the graph
float inByte = 0;

void setup () {
  // set the window size:
  size(400, 300);

  // List all the available serial ports
  // if using Processing 2.1 or later, use Serial.printArray()
  println(Serial.list());

  // I know that the first port in the serial list on
  // my mac is always my Arduino, so I open Serial.list()[0].
  // Open whatever port is the one you're using.
  myPort = new Serial(this, Serial.list()[0], 9600);

  // don't generate a serialEvent() unless you get a newline character:
  myPort.bufferUntil('\n');
  // set initial background:

  background(0);
}

void draw () {
  // draw the line:
  stroke(127, 34, 255);
  line(xPos, height, xPos, height - inByte);

  // at the edge of the screen, go back to the beginning:
  if (xPos >= width) {
    xPos = 0;
    background(0);
  } else {
    // increment the horizontal position:
    xPos++;
  }
}
}

```

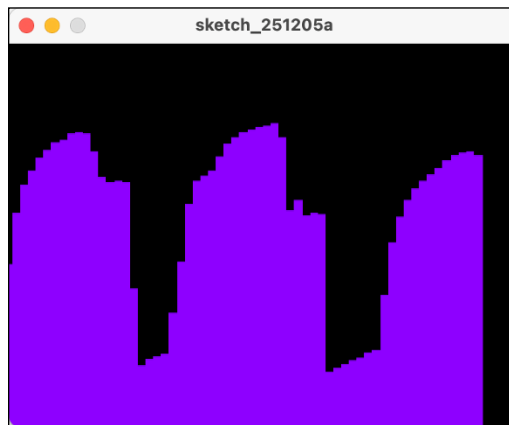
```

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

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

```

Click the Run button to execute the Processing sketch, and a sketch window appears with a black background and a purple graph representing the analog input, as shown in Figure 14-6. As you turn the potentiometer, the purple graph changes to match it. The graph updates over time, and when it reaches the right edge, it resets to the left and continues scrolling.



**FIGURE 14-6:**  
A Processing window showing the live sensor graph.

If you don't see a graph, double-check your wiring:

- » Make sure you're using the correct pin number.
- » Make sure that the potentiometer is wired the correct way around.
- » Check that you're using the correct serial port.
- » Check that your Arduino code uploaded correctly and that your Processing code has no errors. You can't upload to the Arduino while Processing is running — stop the Processing sketch before uploading again.

## Understanding the Arduino Graph sketch

The Arduino sketch reads analog data and sends it to the serial port so that Processing can visualize it. In `setup`, the code just needs to initialize the serial port. The serial port is set to a baud rate of 9600, which must match the baud rate in the Processing sketch.

The analog input pins are set to input by default, so you don't need to set their `pinMode`:

```
void setup() {  
  // initialize the serial communication:  
  Serial.begin(9600);  
}
```

Analog pins are inputs by default, so you don't need to call `pinMode()`. In `loop()`, the analog reading from A0 is printed to the serial port, followed by a newline character:

```
void loop() {  
  // send the value of analog input 0:  
  Serial.println(analogRead(A0));  
}
```

Analog readings are made extremely quickly, usually more quickly than they can be converted to digital format. Sometimes this speed causes errors, so a short delay of 2 milliseconds between readings can help stabilize the results. Think of it like a tap that limits the flow of water:

```
// wait a bit for the analog-to-digital converter  
// to stabilize after the last reading:  
delay(2);  
}
```

## Understanding the Processing Graph sketch

When Arduino sends serial data, Processing reads and visualizes it as a moving line. The same approach can be used for any type of analog sensor — for example, a light, temperature, or sound sensor.

First you import the serial library into the sketch and create a new instance of it. This library lets Processing communicate with your Arduino through the same USB connection used for uploading sketches. In this case, the new serial port object is called `myPort`:

```
import processing.serial.*;

Serial myPort;      // The serial port
```

One integer, defined as `xPos`, keeps track of where the latest bar in the bar graph is drawn (the x position):

```
int xPos = 1;      // horizontal position of the graph
```

```
float inByte = 0;
```

In `setup`, the display window is defined as 400 pixels wide and 300 pixels tall:

```
void setup () {
  // set the window size:
  size(400,300);
}
```

To find the correct serial port, `Serial.list` is called and printed to the console with `println`. In Processing 4 or later, you can use `Serial.printArray()` instead — both commands list available ports, but `printArray()` displays them more clearly. The function `println` is similar to `Serial.println` in Arduino but is used in Processing for monitoring values. These values print to the console rather than the serial port and are used for debugging rather than communication:

```
// List all the available serial ports
// if using Processing 2.1 or later, use Serial.printArray()
.println(Serial.list());
```

Your Arduino is likely to appear at the top of the list, so `myPort` uses position 0 as the Arduino serial port. If you have more than one serial device connected, check the console output to confirm which one corresponds to your Arduino. If you are not using the `Serial.list` function, you can replace `Serial.list()[0]` with another number in the list, which prints to the console. You can also replace `Serial.list()[0]` with the name of the port, such as `/dev/tty.usbmodem26221` or `COM5`. The number 9600 refers to the baud rate — the communication speed. Make sure the baud rate matches the value in your Arduino sketch; otherwise, data will appear garbled or not at all.

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

In this example, you have another way to sort out the good data from the bad. The `serialEvent()` function (part of the serial library) triggers every time new data arrives in the serial buffer. In this sketch, Processing waits until it receives a

newline character (`\n`), which marks the end of each value sent by Arduino. You could also use other delimiters such as a tab (`\t`) if you prefer.

```
// don't generate a serialEvent() unless you get a newline character:
myPort.bufferUntil('\n');
```

To start, the background is colored black:

```
// set initial background:
background(0);
}
```

This bar graph is detailed, with one bar represented by a column of pixels. The `line()` function draws each column of the graph, using RGB values to set the color. Here, purple is created by the combination `stroke(127, 34, 255);`.

Each frame, Processing draws a single vertical line whose height represents the latest sensor reading. The bottom of the line is fixed at the base of the window, and the top varies according to the sensor value. When the line reaches the right edge of the window, the program clears the background and starts drawing again from the left, producing a continuous scrolling graph.

```
void draw () {
  // draw the line:
  stroke(127, 34, 255);
  line(xPos, height, xPos, height - inByte);
}
```

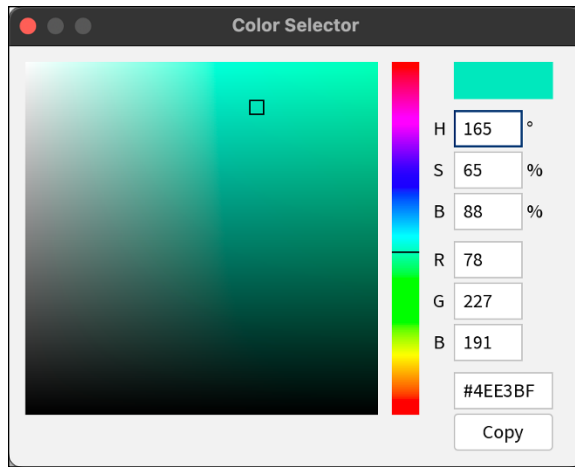


TIP

If you're having trouble selecting a color, choose Tools ⇄ Color Selector on the Processing menu. The Color Selector displays a color picker with the RGB, HSB and hexadecimal values of any color you select, as you can see in Figure 14-7.

This next bit of code handles the movement of the `xPos`, or horizontal, position of the graph over time. Each new reading advances one pixel to the right; when the end of the window is reached, `xPos` resets to 0 and the background clears so the graph can restart.

```
// at the edge of the screen, go back to the beginning:
if (xPos >= width) {
  xPos = 0;
  background(0);
} else {
  // increment the horizontal position:
  xPos++;
}
}
```



**FIGURE 14-7:**  
The built-in color selector can be very useful.

The `serialEvent` function is part of the serial library and triggers when data arrives in the serial buffer. Because you used `bufferUntil('\n')`, the function runs only when a full line of data — ending with a newline character — has been received.

```
void serialEvent (Serial myPort) {
```

A temporary string is declared to store the data read from `myPort`. The data is read as a line of text until the newline marker (`\n`) is reached.

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

An `if` statement checks that the string contains data and isn't null:

```
if (inString != null) {
```

To make sure that no anomalies exist, the `trim` function removes spaces, tabs, and carriage returns from the string. This ensures that only numeric characters remain before converting the string to a number.

```
// trim off any whitespace:  
inString = trim(inString);
```

Now the clean string of numbers is converted into a float, called `inByte`. Converting it to a float allows smoother scaling when mapping the data to screen height.

```
// convert to an int and map to the screen height:  
inByte = float(inString);  
println(inByte);
```

The newly converted `inByte` is then mapped, or scaled, to a more useful range. Because the Arduino's analog-to-digital converter produces values from 0 to 1023, the `map()` function scales this range to `0 - height` so the full sensor range fits perfectly within the Processing window.

```
inByte = map(inByte, 0, 1023, 0, height);
```

After mapping, the next frame of `draw()` displays the new line height, creating a live visualization of your analog data. This example is a great exercise for getting familiar with communication between Arduino and Processing. Try swapping the potentiometer for another analog sensor or changing the line into a circle whose size matches the sensor value — a simple way to explore how visuals can represent physical data.

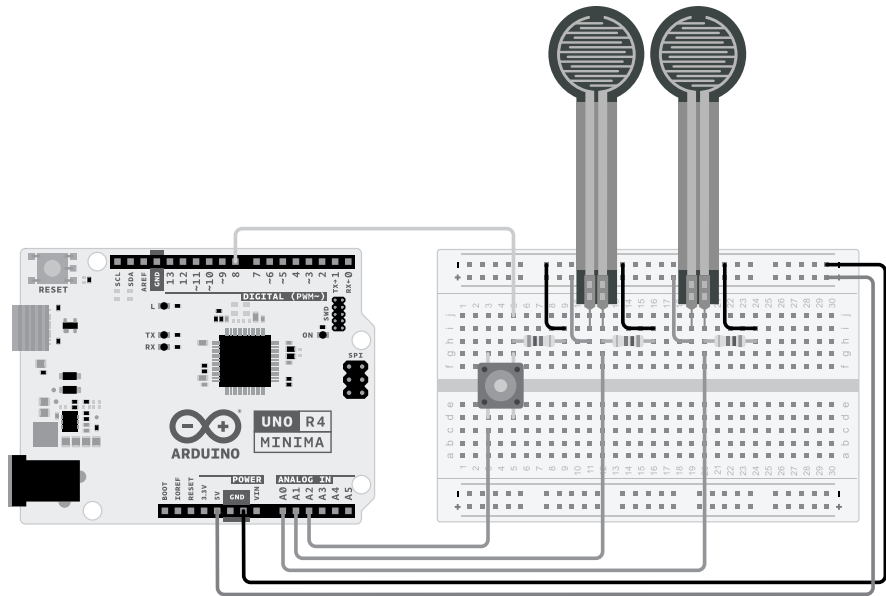
## Sending Multiple Signals

The only thing better than sending a single signal to Processing is sending several at once! This final example shows how to send data from two potentiometers and a button to Processing at the same time. Sending multiple values is straightforward, but you must make sure Processing reads them in the correct order.

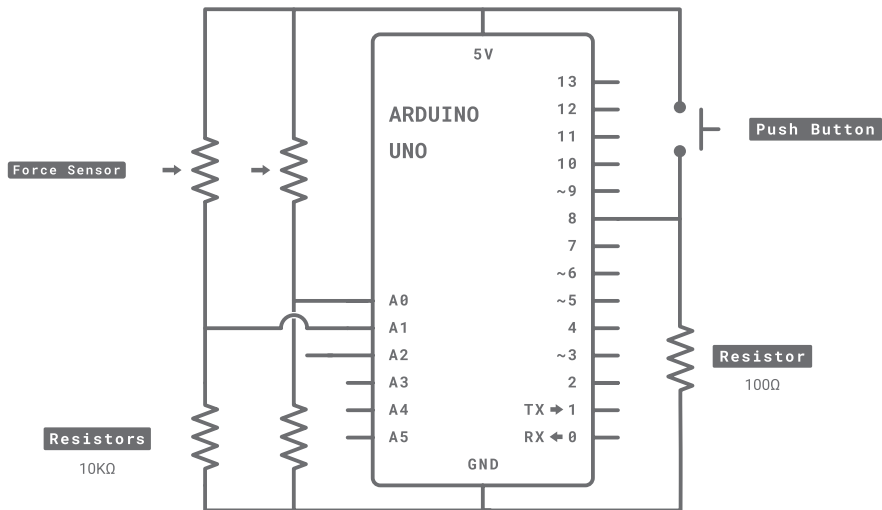
You need the following:

- » An Arduino Uno
- » A breadboard
- » Two 10k ohm potentiometers
- » A pushbutton
- » A 10k ohm resistor
- » Jump wires

The circuit is a combination of three separate inputs. Two potentiometers provide analog readings on A0 and A1, and a pushbutton provides a digital input on pin 2. Wire each potentiometer between 5V and GND with its middle pin going to the analog input. Wire the button between 5V and digital pin 2 with a 10 kΩ resistor to GND as a pull-down. Figures 14-8 and 14-9 show the complete circuit.



**FIGURE 14-8:**  
A circuit diagram for two analog inputs and one digital input.



**FIGURE 14-9:**  
A schematic for two analog inputs and one digital input.

## Setting up the Arduino code

After you assemble your circuit, you need the appropriate software to use it. From the Arduino menu, choose File ⇨ Examples ⇨ 04.Communication ⇨ SerialCall-Response. This sketch contains both Arduino code and the relevant Processing

code for the sketch to work (along with a variation in Max 5 as well). The Processing code below the Arduino code is commented out to avoid interference with the Arduino sketch:

```
/*
  Serial Call and Response
  Language: Wiring/Arduino

  This program sends an ASCII A (byte of value 65) on startup
  and repeats that until it gets some data in.
  Then it waits for a byte in the serial port, and
  sends three sensor values whenever it gets a byte in.

  Thanks to Greg Shakar and Scott Fitzgerald for the improvements

  The circuit:
  * potentiometers attached to analog inputs 0 and 1
  * pushbutton attached to digital I/O 2

  Created 26 Sept. 2005
  by Tom Igoe
  modified 24 April 2012
  by Tom Igoe and Scott Fitzgerald

  This example code is in the public domain.

  https://docs.arduino.cc/built-in-examples/communication/SerialCallResponse/
  */

int firstSensor = 0;    // first analog sensor
int secondSensor = 0;  // second analog sensor
int thirdSensor = 0;   // digital sensor
int inByte = 0;        // incoming serial byte

void setup()
{
  // start serial port at 9600 bps:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }
  pinMode(2, INPUT);    // digital sensor is on digital pin 2
  establishContact();   // send a byte to establish contact until receiver
                        // responds
}

```

```

void loop()
{
  // if we get a valid byte, read analog ins:
  if (Serial.available() > 0) {
    // get incoming byte:
    inByte = Serial.read();
    // read first analog input, divide by 4 to make the range 0-255:
    firstSensor = analogRead(A0)/4;
    // delay 10ms to let the ADC recover:
    delay(10);
    // read second analog input, divide by 4 to make the range 0-255:
    secondSensor = analogRead(1)/4;
    // read switch, map it to 0 or 255L
    thirdSensor = map(digitalRead(2), 0, 1, 0, 255);
    // send sensor values:
    Serial.write(firstSensor);
    Serial.write(secondSensor);
    Serial.write(thirdSensor);
  }
}

void establishContact() {
  while (Serial.available() <= 0) {
    Serial.print('A'); // send a capital A
    delay(300);
  }
}

```

Upload this code to your Arduino. Now that the Arduino is set up to send data to Processing, you need to set up the Processing sketch to receive and interpret the data over the serial port.

## Setting up the Processing code

You find the Processing code within multiline comment markers (`/* */`) at the bottom of the Arduino `SerialCallResponse` sketch. Copy the code within the comment markers into a new Processing sketch and save with an appropriate name, such as `SerialCallResponse`:

```

// This example code is in the public domain.

import processing.serial.*;

```

```

int bgcolor;                // Background color
int fgcolor;                // Fill color
Serial myPort;              // The serial port
int[] serialInArray = new int[3]; // Where we'll put what we receive
int serialCount = 0;        // A count of how many bytes we receive
int xpos, ypos;            // Starting position of the ball
boolean firstContact = false; // Whether we've heard from the
                                // microcontroller

void setup() {
    size(256, 256); // Stage size
    noStroke();     // No border on the next thing drawn

    // Set the starting position of the ball (middle of the stage)
    xpos = width/2;
    ypos = height/2;

    // Print a list of the serial ports, for debugging purposes:
    println(Serial.list());

    // I know that the first port in the serial list on my mac
    // is always my FTDI adaptor, so I open Serial.list()[0].
    // On Windows machines, this generally opens COM1.
    // Open whatever port is the one you're using.
    String portName = Serial.list()[0];
    myPort = new Serial(this, portName, 9600);
}

void draw() {
    background(bgcolor);
    fill(fgcolor);
    // Draw the shape
    ellipse(xpos, ypos, 20, 20);
}

void serialEvent(Serial myPort) {
    // read a byte from the serial port:
    int inByte = myPort.read();
    // if this is the first byte received, and it's an A,
    // clear the serial buffer and note that you've
    // had first contact from the microcontroller.
    // Otherwise, add the incoming byte to the array:
    if (firstContact == false) {
        if (inByte == 'A') {
            myPort.clear(); // clear the serial port buffer
        }
    }
}

```

```

        firstContact = true;    // you've had first contact from the
        microcontroller
        myPort.write('A');    // ask for more
    }
}
else {
    // Add the latest byte from the serial port to array:
    serialInArray[serialCount] = inByte;
    serialCount++;

    // If we have 3 bytes:
    if (serialCount > 2 ) {
        xpos = serialInArray[0];
        ypos = serialInArray[1];
        fgcolor = serialInArray[2];

        // print the values (for debugging purposes only):
        println(xpos + "\t" + ypos + "\t" + fgcolor);

        // Send a capital A to request new sensor readings:
        myPort.write('A');
        // Reset serialCount:
        serialCount = 0;
    }
}
}
}

```



TIP

In Processing 4, you may be asked to allow serial access the first time you run a sketch (especially on macOS). Grant permission so the app can communicate with your Arduino.

Click Run to execute the Processing sketch. A black window appears. Turning the potentiometers moves a white dot horizontally and vertically; pressing the push-button turns the dot white (active). Each new reading prints to the console as tab-separated values, which is useful for debugging.

If you don't see the correct behavior, double-check your wiring:

- » Check pin numbers and wiring.
- » Make sure the potentiometers are wired correctly.
- » Check that you're using the correct serial port.
- » Stop the Processing sketch before uploading new Arduino code.

# Understanding the Arduino SerialCallResponse sketch

At the start of the sketch, four integer variables are declared. Three are for the sensor values and one stores an incoming byte from the Processing sketch:

```
int firstSensor = 0;    // first analog sensor
int secondSensor = 0;  // second analog sensor
int thirdSensor = 0;   // digital sensor
int inByte = 0;        // incoming serial byte
```

In `setup()`, the serial port is established with a baud rate of 9600.

The `while(!Serial)` line waits for a serial connection on boards that use native USB (Leonardo or R4 Minima, for example).

```
void setup()
{
  // start serial port at 9600 bps:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for USB port only
  }
}
```

Pin 2 is the pushbutton pin and is set as an input by using `pinMode()`:

```
pinMode(2, INPUT); // digital sensor is on digital pin 2
```

The custom function `establishContact()` is then called from lower down in the sketch. This sends the letter 'A' every 300 ms until Processing responds, confirming that both sides are ready.

```
establishContact(); // send a byte to establish contact until
                    // receiver responds
}
```

Inside `loop()`, the Arduino waits for a byte from Processing. When one arrives, it reads the three sensors and sends their values as bytes (0–255) over serial.

```
if (Serial.available() > 0) {
  // get incoming byte:
  inByte = Serial.read();
  // read first analog input, divide by 4 to make the range 0–255:
  firstSensor = analogRead(A0)/4;
}
```

```

// delay 10ms to let the ADC recover:
delay(10);
// read second analog input, divide by 4 to make the range 0-255:
secondSensor = analogRead(1)/4;
// read switch, map it to 0 or 255L
thirdSensor = map(digitalRead(2), 0, 1, 0, 255);
// send sensor values:
Serial.write(firstSensor);
Serial.write(secondSensor);
Serial.write(thirdSensor);
}

```

## Understanding the Processing SerialCallResponse sketch

The Processing sketch receives three values from your Arduino and uses them to update the position and color of a dot onscreen. Unlike the previous examples, this sketch also sends a message back to the Arduino after each update to request the next set of readings. This two-way exchange is called *handshaking*, and it helps keep both sides synchronized.

At the top of the sketch, the serial library is imported, and several variables are declared:

```

import processing.serial.*;
int bgcolor; // Background color
int fgcolor; // Fill color of the dot
Serial myPort; // Serial connection to Arduino
int[] serialInArray = new int[3]; // Incoming bytes from Arduino
int serialCount = 0; // How many bytes have been received
int xpos, ypos; // Position of the dot
boolean firstContact = false; // Whether handshaking is complete

```

The `serialInArray` stores the three incoming bytes (two potentiometers and one button). The `serialCount` variable keeps track of how many bytes have arrived so far.

The `setup()` function prepares the window, sets the starting position of the dot, and opens the serial connection:

```

void setup() {
  size(256, 256); // Size of the display window
}

```

```

noStroke();           // Draw shapes without outlines

xpos = width/2;      // Start the dot in the center
ypos = height/2;

println(Serial.list()); // List available serial ports
String portName = Serial.list()[0];
myPort = new Serial(this, portName, 9600);
}

```

The call to `Serial.list()` prints all available serial ports in the console. Your Arduino usually appears at position 0, but confirm this if multiple devices are connected. The serial port is opened at 9600 baud to match the speed set in the Arduino sketch.

The `draw()` loop continually refreshes the display:

```

void draw() {
  background(bgcolor); // Clear the window with the background color
  fill(fgcolor);       // Use the most recent sensor value as the fill

  ellipse(xpos, ypos, 20, 20); // Draw the dot at the current x/y position
}

```

At this point, the dot will only move and change color after values start arriving from the Arduino.

Most of the work happens inside `serialEvent()`, which triggers automatically whenever a new byte is available on the serial port:

```

void serialEvent(Serial myPort) {
  int inByte = myPort.read(); // Read the newest incoming byte
}

```

The first time any data arrives, the sketch checks whether the byte is the character A. This is the signal the Arduino sends repeatedly until Processing is ready. Once this handshake byte is received, Processing clears the serial buffer, records that contact has been made, and sends an A back to request the first real set of values:

```

if (firstContact == false) {
  if (inByte == 'A') {
    myPort.clear(); // Clear anything already in the buffer
    firstContact = true; // Handshake complete
    myPort.write('A'); // Ask for the first sensor readings
  }
}
}

```

After handshaking is complete, the sketch stores each incoming byte in order:

```
else {
  serialInArray[serialCount] = inByte;
  serialCount++;
}
```

When all three bytes have arrived, the sketch assigns them to the dot's position and color:

```
if (serialCount > 2) {
  xpos = serialInArray[0];    // Horizontal position
  ypos = serialInArray[1];    // Vertical position
  fgcolor = serialInArray[2]; // Button controls color

  println(xpos + "\t" + ypos + "\t" + fgcolor); // Debug output
}
```

Finally, Processing requests the next three values by sending another A back to Arduino, then resets the counter:

```
myPort.write('A'); // Request new data
  serialCount = 0;
}
}
}
```

This method keeps the dot responsive as it moves around the window based on the potentiometers, and it changes color when the button on the Arduino is pressed. More importantly, this example demonstrates how to structure reliable two-way communication between Processing and Arduino — an approach you can extend to more complex interactive projects.

This example is a great way to get started reading multiple sensors into a computer program. Why not build a large, pressure-sensitive control surface for a game, music app, or interactive installation? Or gather data from a variety of sensors to monitor the weather around your house and visualise it live on your computer or phone.

The output for this sketch is intentionally simple, but it offers huge potential for creative ways to map and visualise data. A few web searches for interactive data visualisation, creative coding, or physical computing projects will give you a strong sense of what's possible.



# 5

## The Part of Tens

**IN THIS PART . . .**

Discover websites that will inspire you to learn more about Arduino.

Browse online stores for all sorts of Arduino-compatible components.

- » Looking for inspiration online
- » Searching for tutorials and further reading
- » Finding communities in the real world

## Chapter **15**

# Ten Places to Learn More about Arduino

If this is your first step into the world of Arduino, you'll be glad to know there's an abundance of resources out there to help you along the way. From official tutorials to community-driven projects and real-world meetups, there are countless ways to learn and get inspired. Here are ten of the best places to start exploring.

## Arduino Blog

`blog.arduino.cc`

The Arduino Blog is the official source for all Arduino-related news. You'll find updates on the latest hardware and software, as well as highlights of projects and community events from around the world.

# Arduino Project Hub

---

`projecthub.arduino.cc`

The Project Hub is packed with tutorials and community projects. Each one includes code, diagrams, and step-by-step instructions so you can follow along and build something yourself.

# Arduino Forum

---

`forum.arduino.cc`

The Arduino Forum is a great place to get answers to specific questions. Chances are someone else has faced the same problem you're working on, so with a little searching you'll often find a solution — and if not, you can ask the community directly.

# Adafruit

---

`learn.adafruit.com` | `blog.adafruit.com`

Adafruit is an online shop, repository, and forum for all kinds of kits to help you make your projects work. Its blog announces the ever-growing selection of available Adafruit products as well as other interesting tech news.

# SparkFun

---

`learn.sparkfun.com`

SparkFun sells a wide range of components, many of which are perfect for Arduino projects. Just as useful are their Hookup Guides and tutorials, which show you exactly how to get sensors, displays, and other parts working with your board.

# Hackaday

---

[hackaday.com](http://hackaday.com)

Hackaday is an excellent resource for all sorts of technological magic. In addition to presenting a lot of Arduino-related projects and posts, the site offers equal amounts of just about any other category of technology that you can think of. This site contains an excellent collection of posts and information to fuel your imagination.

# YouTube

---

[youtube.com](http://youtube.com)

YouTube is a great place to kill time, but rather than watching cats do funny things, why not search for *Arduino* to discover new project, tutorials, and demos. YouTube videos won't always be the most reliable source for well-documented projects, but they provide a broad look at Arduino projects in action.

# Instructables

---

[instructables.com](http://instructables.com)

Instructables is a huge online library of DIY projects where makers share step-by-step instructions for their creations. It isn't limited to Arduino or even technology, so you'll find a whole world of creative projects to explore.

# Reddit and Discord Communities

---

[reddit.com/r/arduino](http://reddit.com/r/arduino)

Want real-time advice or just to share your latest creation? Online communities such as Reddit's *r/arduino* and various Discord servers are active places to learn, swap tips, and troubleshoot. They're especially useful if you want quick feedback from fellow makers.

# Maker Spaces, FabLabs, and Workshops

Maker Spaces, FabLabs, and workshops are physical spaces where artists, designers, makers, hackers, coders, engineers, and anyone else can meet to learn, socialize, and collaborate on projects. These spaces are popping up in cities, libraries, and universities worldwide. A quick online search or a visit to [Meetup.com](https://www.meetup.com) will usually reveal one near you.

- » Discovering Arduino suppliers around the world
- » Introducing international electronics distributors
- » Reusing old parts

## Chapter **16**

# Ten Great Shops to Know

**W**hen it comes to buying parts for your project, you'll find a huge and growing number of shops that cater to your needs. Although these stores deal in other hobby electronics as well as Arduino, they stock a variety of boards and components that you can use in your Arduino project. This chapter provides just a small sample of the stores out there, so shop around.

## Arduino Store

[store.arduino.cc](http://store.arduino.cc)

The official Arduino Store is the most direct place to buy Arduino boards and accessories. Here you'll find every official board, starter kit, and shield, as well as select third-party products chosen by the Arduino team. Buying here ensures that what you get is genuine and fully supported, and you'll often find bundles and kits that are designed with beginners in mind.

# Adafruit

---

[adafruit.com](http://adafruit.com)

Founded in 2005 by engineer Limor “Ladyada” Fried, Adafruit has become one of the most trusted names in hobby electronics. The company not only sells its own high-quality products but also curates tools, sensors, and modules from around the world. Adafruit’s real strength is education — their tutorials, videos, and forums make even the trickiest concepts more approachable. Based in New York, they ship worldwide and have distributors in many countries.

# SparkFun

---

[sparkfun.com](http://sparkfun.com)

SparkFun has been supporting hobbyists since 2003 with its own line of boards and kits alongside a broad range of Arduino-compatible hardware. Their online store doubles as a classroom, with guides, tutorials, and lively comment sections that help you learn from other users’ experiences. Based in Boulder, Colorado, SparkFun ships globally and remains one of the most hobbyist-friendly suppliers.

# Seeed Studio

---

[seeedstudio.com](http://seeedstudio.com)

Seeed Studio, based in Shenzhen, China, describes itself as an “open hardware facilitation company.” They are known for affordable modules and systems like Grove, which make it easy to snap components together without breadboards. Seeed also offers PCB manufacturing and small-batch production, so you can prototype and scale up with the same partner. Their community features allow makers to propose and vote on projects, giving them a unique, collaborative spirit.

# Pimoroni

---

[shop.pimoroni.com](http://shop.pimoroni.com)

Pimoroni is a UK-based maker and retailer known for its colourful, creative approach to electronics. They design and manufacture their own breakout boards,

kits, and accessories, many of which are Arduino and Raspberry Pi compatible. Pimoroni also stocks official Arduino products, tools, and books, making it a friendly one-stop shop for hobbyists. Their focus on well-designed, playful hardware makes electronics feel approachable and fun, while still offering plenty for more advanced makers.

## Digi-Key

[digikey.com](http://digikey.com)

Digi-Key started out supplying hobbyists in amateur radio but has since grown into one of the largest electronics distributors in the world. Their website makes it easy to search through a massive range of components, from the simplest resistor to complete development kits. Beginners may find the breadth overwhelming at first, but Digi-Key is a trustworthy place to get exactly what you need.

## RS Components

[rs-online.com](http://rs-online.com)

RS Components is one of the world's oldest and most established electronics distributors, serving both professionals and hobbyists. Based in the UK with operations worldwide, RS offers a huge catalogue of components, tools, and test equipment. Although it feels more like an engineer's supplier than a hobby shop, RS is reliable, ships quickly, and stocks a wide range of Arduino boards and accessories. For international readers, especially in Europe, RS is a dependable place to find genuine parts at scale.

## Tindie

[tindie.com](http://tindie.com)

Tindie is a global marketplace for indie hardware makers. Here you'll find unique boards, kits, and tools made by other hobbyists and small creators. It's a great way to support the community while discovering inventive products you won't see from big distributors. Quality can vary, but the site's review system and active community help guide you to trusted sellers.

# AliExpress

---

[aliexpress.com](http://aliexpress.com)

AliExpress offers an enormous selection of modules, sensors, and accessories direct from manufacturers, often at very low prices. It's a good place to experiment with inexpensive parts, though shipping times can be long and quality varies. Reading reviews and ordering from sellers with high ratings is key. For those willing to wait, it can be one of the cheapest ways to build up your component collection.

# Treasure Hunting

---

eBay, local surplus stores, and even salvaging old electronics can be a treasure trove for parts. Printers, scanners, and other discarded gadgets often contain useful motors, gears, and sensors that would be costly to buy new. Although it takes some research and tinkering, recycling components is not only cost-effective but also a great way to reduce waste while learning more about how everyday electronics work.

# Index

## Numerics

3.3V pin, 20, 21

5V pin, 20, 21

## A

Adafruit, 25, 265, 328, 332

2.8" TFT Touch Shield, 260–261

Arduino Starter Pack, 29

FONA 800 shield, 264

Motor/Stepper/Servo shield  
v2.3, 262–263

Music Maker shield, 259–260

RGB 16x2 LCD shield, 260

Ultimate GPS Logger shield, 263

Adafruit Blog, 265

Adafruit Feather, 25

Adafruit Proto-Screwshield, 259

adhesive putty for soldering,  
172, 182, 187

AI. *See* artificial intelligence

Albion Café, 155

AliExpress, 29, 334

Amazon, 27, 29

AnalogInOutSerial sketch,  
109–114, 186

circuit, 109–110

code, 110–111

hardware, 109

understanding, 112–114

analog input pins, 19, 104, 106,  
134–135, 242, 310

AnalogInput sketch, 102–108

circuit, 103–104

code, 104–105

hardware, 102

tweaking, 107–108

understanding, 106–107

analogRead() function, Arduino, 107, 217

analog sensors, 106, 212–213, 310

analogWrite() function, Arduino, 87,  
91, 93, 96, 113, 126, 224

Arduino, 1, 7–8, 275

community, 255, 265

history of, 8–10

inputs and outputs for, 14

open source philosophy of, 15, 271

Playground, 15, 277

projects, using AI in, 34–38

Arduino and Processing,  
communication between

drawing graph, 305–314

making virtual button, 293–305

sending multiple signals, 314–323

Arduino Blog, 265, 327

- Arduino boards, 10–11, 17, 21–22.
  - See also* sketches
  - components and starter kits, 27–30
  - official, 22–24
  - power supply to, 189
  - securing, 191
  - using circuit diagrams with, 78–80
- Arduino\_CapacitiveTouch
  - library, 244–245
- Arduino Cloud AI Assistant, 38
- Arduino Cloud Editor, 32, 33
- Arduino-compatible boards, 22, 24–25
- Arduino Forum, 15, 265, 328
- Arduino GIGA, 23
- Arduino IDE, 24, 31, 50, 121, 129, 133, 266. *See also* libraries
  - installation of, 31
  - key areas of, 32–34
  - Library Manager, 244–245, 266, 269–270
  - ZIP library installation in, 270
- Arduino MKR, 23
- Arduino Motor shield Rev3, 263
- Arduino Nano, 23
- Arduino Nicla, 23–24
- Arduino Portenta, 23
- Arduino Project Hub, 162, 328
- Arduino projects, 161–162
  - Baker Tweet, 155–156
  - Chorus, 151–153
  - Compass Lounge, The*, 156–158
  - Good Night Lamp, The*, 158–160
  - Little Printer, 160–161
  - Push Snowboarding, 153–155
  - workspace for working on, 30–31
- Arduino Pro Mini, 153
- Arduino shield, 178–182
  - assembly, 181
  - circuit building, 186–188
  - header pins, 181–182
  - laying out pieces of circuit, 179–181
  - proto kit, 178–181
  - testing, 188
- Arduino Shield List, 266
- Arduino Starter Kit, 29
- Arduino Starter Pack, Adafruit, 29
- Arduino Store, 25, 265, 331
- Arduino Uno, 10, 14, 17–23, 28
  - built-in LEDs, 21
  - header sockets, 19–20
  - R4 Minima, 18–21, 41, 42
  - R4 WiFi, 18
  - Renesas RA4M1 microcontroller, 19
  - reset button, 21, 140
  - special pin features, 20
  - USB and power, 20–21
- arrays, 141–142, 216–218
- artificial intelligence (AI), 34–38
  - limitations of, 36–37
  - role in debugging, 35
  - role in prototyping and creative exploration, 36

role in research, 35–36  
tutoring role of, 35  
audio shields, 259–260

## B

background() function,  
Processing, 287–288  
Baker Tweet project, 155–156  
Banzi, Massimo, 10  
Barragán, Hernando, 9  
base (transistor), 120  
Basic Stamp, 10  
baud rate, 301, 303, 310, 311, 320  
Berg, 160–161  
binary number system (base-2),  
287–288  
bit, 288  
blinking an LED, 7, 39  
connecting external LED to Arduino  
board, 52–54  
identification of board, 41  
software configuration, 41–43  
Blink sketch, 45–52  
code, 193–194  
comments, 47–48  
finding, 40–41  
functions, 48–50  
loop() function, 51–52  
setup, 49–50  
tweaking, 54  
uploading, 43–44  
BlinkWithoutDelay sketch, 195–200

circuit, 195–196  
code, 196–198  
hardware, 195  
understanding, 198–201  
block comments. *See*  
multiline comments  
blogs, 265, 327  
Booleans, 236, 300  
breadboards, 20, 58–59, 77–78, 81, 164,  
186. *See also* circuit, building  
brushed DC motor. *See* DC motor  
budget starter kits, 29  
Burton Snowboards, 153  
butane soldering irons,  
167–168  
Button sketch, 97–102  
circuit, 97–98  
code, 98–99  
hardware for, 97  
tweaking, 101–102  
understanding, 100–101  
byte, 288

## C

Calibration sketch, 219–224  
circuit, 219  
code, 220–221  
hardware, 219–220  
understanding, 222–224  
Camel Case naming convention, 49  
CapacitiveSensor library, 243  
capacitive sensors, 243–248

CapacitiveTouch sketch, 245–248  
code, 246–247  
hardware, 245  
Love Button, 246  
understanding, 247–248

Chorus project, 151–153

circuit, building  
cleaning, 188  
knowing your circuit, 186  
laying out your circuit, 187  
soldering, 187–188  
wires, 187

circuit bending, 12

circuit diagrams, 76–78. *See also specific sketches*  
light switch, 77  
symbols, 77  
using with Arduino board, 78–80

collector (transistor), 120

color  
changing, Processing, 287–290  
color-coding, 80–82, 89–90, 94  
resistor color charts, 82–84

Color Selector tool, Processing, 312, 313

comments (code), 47–48

*Compass Cards*, 157–158

*Compass Lounge, The*, 156–158

constants, 100

constrain() function, Arduino, 223

continuity tester (multimeter), 62, 67, 97

copper braid, 175–176

core functions, 49

C (programming language), 45, 92

crocodile clips, 62, 171, 182

Cuartielles, David, 10

current, 70  
Joule’s Law, 75–76  
measurement of, 65  
Ohm’s Law, 71–73  
reverse, 117, 120

## D

datasheets, 81–82

DC motor, 29  
MotorControl sketch, 123–126  
and PWM, 87  
spinning, 117–122

Debounce sketch, 200–205  
circuit, 201  
code, 202–203  
hardware, 200–201  
understanding, 204–205

declarations, 91–92

delay() function, Arduino, 52, 114, 194

Deschamps-Sonsino, Alexandra, 158

de-soldering wire, 175–176

DFRobot Gamepad shield v2.0, 261

DFRobot SIM7600G LTE Shield, 264  
Digi-Key, 27, 333  
digital display (multimeter), 62  
DigitalInputPullup sketch, 227–231  
  circuit, 227–228  
  code, 228–229  
  hardware, 227  
  understanding, 230–231  
digital pins, 19, 20, 119  
digitalRead() function, Arduino, 101  
digitalWrite() function, Arduino, 51,  
  52, 101, 108, 200, 236  
dimmer circuit, 109–110, 188, 218  
diodes, 29, 73, 117, 120–121  
Discord servers, 329  
display shields, 259–261  
distance measurement, sensors  
  for, 249–254  
draw() function, Processing,  
  299, 314, 322  
duty cycle, 87

## E

eBay, 27, 29, 334  
editor pane, Arduino IDE, 33  
EEPROM. *See* erasable programmable  
  read-only memory  
electrical polarity  
  of LEDs, 78, 79  
  of piezos, 137  
  of UNO R4, 21

electricity, 69–71  
electric motors, 115–116  
  DC motor, 29, 87, 117–126  
  servomotors, 28, 117, 126–135  
  shields, 262–263  
electromagnetism, 115–116  
electronics, 12, 36  
  color-coding, 80–81, 89–90, 94  
  datasheets, 81–82  
  distributors, 27  
  old, salvaging, 334  
element14, 27  
ellipse, drawing (Processing),  
  281–282, 286–287  
ellipseMode() function,  
  Processing, 286–287  
else statement, 302  
emitter (transistor), 120  
enclosures, 189–190, 265  
equipment wire, 60–61, 176–177  
erasable programmable read-only  
  memory (EEPROM), 267  
ESP32, 24  
Espressif, 24  
Ethernet library, 267  
eye protection, and soldering,  
  177

## F

FabLabs, 330  
face tracking, 304

Fade sketch, LED, 87–96  
  calculation of resistance, 88–89  
  circuit, 88, 89  
  code, 90–91  
  declarations, 92  
  hardware for, 87  
  for loop, 96  
  tweaking, 94–96  
  variables, 92–94

Farnell, 27

`fill()` function, Processing, 288–290

Firmata library, 269

fixed-temperature soldering  
  irons, 165–166

float, 248, 299, 313

flux-cored solder, 171

FONA 800 shield, Adafruit, 264

force sensors, 237–239. *See also*  
  *toneKeyboard* sketch

for loop, 96, 131, 142, 217, 242–243

forward voltage, 73

Fried, Limor “Ladyada,” 25, 332

Fry, Benjamin, 9, 276

functions, 48–49. *See also*  
  *specific functions*

## G

Geiger–Müller tube, 264–265

GitHub, 265, 267, 270, 271

global variables, 199

GND (ground) pins, 20

*Good Night Lamp, The*, 158–160

graph, drawing, 305  
  circuit, 305–306  
  setting up Arduino code, 306–307  
  setting up Processing code,  
    307–309  
  understanding Arduino Graph  
    sketch, 310  
  understanding Processing Graph  
    sketch, 310–314

graphic user interface (GUI),  
  10, 32

ground rail (breadboard), 58

GUI. *See* graphic user interface

## H

Hackaday, 265, 329

hacking, 12–13

Hackster.io, 162

handshaking, 321, 322

hardware hacking, 12–13

HC-SR04 sketch, 249–254  
  circuit, 250–251  
  code, 251–252  
  hardware, 250  
  understanding, 252–254

header pins, 181–182

header sockets, 19–20, 181

helping hand. *See* third hand  
  for soldering

hobby motor. *See* DC motor

HX711 amplifier board, 239

## I

ICSP (in-circuit serial programming) connector, 181

IDE. *See* integrated development environment

IDII. *See* Interaction Design Institute Ivrea

`if` statement, 93–94, 101, 108, 126, 194, 205, 210, 218, 230, 301, 304, 313

infrared proximity sensors, 249

inputs, 14, 29

- access, and enclosures, 189
- analog input pins, 19, 104, 106, 134–135, 242, 310
- calibration of, 218–224
- obstruction by shields, 256
- shields, 261–262

Instructables, 162, 329

insulated equipment wire, 60–61

integrated development environment (IDE)

- Arduino, 24, 31–34, 50, 121, 129, 133, 244–245, 266, 269–270
- Processing, 276–277

interaction design, 8–10

Interaction Design Institute Ivrea (IDII), 8

Internet of Things (IoT), 23, 155, 158, 160

`int` variable, 92, 141

IoT. *See* Internet of Things

`isTouched()` method, 248

## J

Joule, James Prescott, 75

Joule’s Law, 75–76

jump wires, 60–61

## K

keyboards, 12

Kin Design, 156

Knob sketch, 132–135

- circuit, 132–133
- code, 133–134
- hardware, 132
- understanding, 134–135

Knock sketch, 232–237

- circuit, 232–233
- code, 233–235
- hardware, 232
- understanding, 235–237

## L

LDR. *See* light-dependent resistor

lead-free solder, 170–171

lead poisoning, 170

lead solder, 170

LEDs. *See* light-emitting diodes

legacy shields, 263–265

Libelium/Cooking-Hacks Geiger-counter shield, 264–265

libraries, 266–267. *See also specific libraries*  
additional, installation of, 269–270  
community-made, 269, 271  
example sketches in, 266, 270  
installation from ZIP file, 270  
serial communication, 299  
for shields, 257  
standard, 267–269

Library Manager, Arduino IDE, 244–245, 266, 269–270

light-dependent resistor (LDR), 28, 213

light-emitting diodes (LEDs), 28, 189  
blinking, 7, 39–54  
BlinkWithoutDelay sketch, 195–200  
built-in, Arduino Uno R4 Minima, 21  
connecting external LED to Arduino board, 52–54  
Fade sketch, 87–96  
light switch circuit, 70–71  
polarity of, 78, 79

light sensors, 28  
Calibration sketch, 219–224  
Smoothing sketch, 213–218

light switch circuit, 70–71  
diagram, 77–78  
running through Arduino board, 78–80

line() function, Processing, 284, 312

LiquidCrystal library, 268

Little Printer, 160–161

load sensors, 237–239

local variables, 199

long, 198, 204

loop() function, Arduino, 51–52, 96, 101, 107, 108, 113, 122, 131, 205, 253, 303, 310, 320

## M

Make: Makezine, 265

Maker Spaces, 330

manual wire strippers, 173–174

map() function  
Arduino, 113, 135, 147, 223  
Processing, 314

Max (software), 278

Maxuino library, 278

mechanical wire strippers, 173–174

menu bar, Arduino IDE, 32

message area, Arduino IDE, 34

microcontroller boards, 7, 10, 19

microcontrollers, 7, 10, 14

MIDI protocol, 262

millis() function, Arduino, 199, 222

mode-selection dial (multimeter), 62

Moggridge, Bill, 8

Moog synthesizer, 11

MotorControl sketch, 123–126  
circuit, 123–124  
code, 124  
hardware, 123  
tweaking, 125–126  
understanding, 125

Motor shield Rev3, Arduino, 263

motor shields, 262–263

Motor sketch, 118–122  
  circuit, 118–121  
  code, 122  
  hardware, 118  
  understanding, 122

Motor/Stepper/Servo shield v2.3,  
  Adafruit, 262–263

Mouser, 27

multicore wires, 61, 176, 187, 191

multiline comments, 47–48

multimeter, 62–63, 174  
  continuity of circuit, checking, 67, 174  
  current measurement using, 65  
  resistance measurement using, 65–66  
  voltage measurement using, 64

multiple signals, sending  
  circuit, 314–315  
  hardware, 314  
  setting up Arduino code, 315–317  
  setting up Processing code, 317–319  
  understanding Arduino  
    SerialCallResponse sketch, 320–321  
  understanding Processing  
    SerialCallResponse sketch, 321–323

musical instrument, 143–147

Music Maker shield, Adafruit, 259–260

## N

needle-nosed pliers, 61–62, 174

Newark, 27

NewPing library, 250, 251, 253, 254

Nintendo GameBoy, 12

Nokia, 153

non-light-emitting diodes, 73

noStroke() function, Processing, 289

noTone() function, Arduino, 143

NPN transistor, 120

## O

Ohm, Georg Simon, 71

Ohm's Law, 71–73

opacity, changing (Processing),  
  288–290

open source, 15, 271

outputs, 14, 29. *See also* inputs  
  access, and enclosures, 189  
  obstruction by shields, 256

## P

p5.js, 277–278

p5.serialport library, 278

patching, 10

Pduino library, 278

perfboards, 179–180

phone switchboards, 11

physical computing, 9

piezos, 29, 136, 137, 231–232  
  Knock sketch, 232–237  
  PitchFollower sketch, 144–147  
  toneMelody sketch, 136–143

Pimoroni, 26, 265, 332–333

ping\_median() method, 254

pinMode() function, Arduino, 50, 92–93,  
  100, 106, 122, 230, 310, 320

pitches.h, 139–140, 242  
 PitchFollower sketch, 144–147  
     circuit, 144–145  
     code, 145–146  
     hardware, 144  
     understanding, 146–147  
 PNP transistor, 120  
 point() function, Processing, 283  
 potentiometer, 28, 103, 105, 109, 111, 123, 125–126, 131–133, 186, 214, 305–306, 309, 314, 319  
 power, 70, 71  
     calculation of, 74–75  
     Joule’s Law, 75–76  
     socket, Arduino Uno, 20–21  
     and transistors, 119  
 power rail (breadboard), 58  
 power shields, 262–263  
 power supply, 51  
     and enclosures, 189  
     power pins, 19  
     and shields, 256–257  
 pre-defined variables, 49, 53  
 pressure pads, 237–238, 249  
 println() function,  
     Processing, 280, 311  
 probes (multimeter), 62  
 Processing, 9, 10, 15, 32, 276–277.  
     *See also* Arduino and Processing,  
     communication between  
     IDE, 276–277  
     installation of, 278–279  
     toolbar, 280  
     window, main areas of, 279–280  
 Processing sketches, 280–282  
     changing color and opacity,  
         287–290  
     drawing shapes, 282–287  
     interactions, 290–291  
 project box, 189–190  
 projected capacitive sensors,  
     243–244  
 Proto shield kit Rev3, 258  
 prototyping, 163, 186  
     shields, 258–259  
     tools for, 54–67  
     use of AI in, 36  
 pull-down resistors, 54, 227, 314  
 pull-up resistors, 227, 230  
 pulseIn() function, Arduino, 253  
 pulse-width modulation (PWM),  
     20, 87, 112, 113, 130, 222.  
     *See also* Fade sketch, LED  
 Pure Data (Pd), 278  
 pushbuttons, 28, 77–78, 181,  
     226–227  
     bouncing, 200  
     Button sketch, 97–102  
     Debounce sketch, 200–205  
     DigitalInputPullup sketch, 227–231  
     StateChangeDetection sketch,  
         206–212  
 Push Snowboarding project, 153–155  
 PWM. *See* pulse-width modulation

## R

Rapid, 27  
Raspberry Pi Pico, 24  
Reas, Casey, 9, 276  
rectangle, drawing (Processing), 284–287  
rectMode() function, Processing, 285, 300  
Reddit, 265, 329  
relays, 29  
Renesas RA4M1 microcontroller, 19  
reset button, Arduino Uno, 21, 140  
resistance, 71  
    calculation, LED Fade sketch, 88–89  
    Joule’s Law, 75–76  
    measurement of, 65–66  
    Ohm’s Law, 71–73  
resistors, 78  
    color charts, 82–84  
    force-sensitive, 237–243  
    pull-down, 54, 227, 314  
    pull-up, 227, 230  
    resistance, measurement of, 65–66  
    variable, 66, 103, 109  
Restriction of Hazardous Substances Directive (RoHS), 170  
reverse current, 117, 120  
reverse polarity, 21  
reverse voltage, 73  
RGB 16x2 LCD shield, Adafruit, 260  
RobotShop, 26  
RoHS. *See* Restriction of Hazardous Substances Directive  
RS Components, 27, 333

## S

SdFat library, 268  
SD library, 268  
Seeed Studio, 26, 265, 332  
    Blog, 265  
    Seeeduino v4.3, 22  
    Wio LTE Cat.1 Shield, 264  
sensors, 225–226  
    analog, 106, 212–213, 310  
    capacitive, 243–248  
    for distance measurement, 249–254  
    piezo, 231–237  
    pressure, force, and load sensors, 237–243  
    pushbuttons, 226–231  
serial communication, 108, 230  
    AnalogInOutSerial sketch, 109–114  
    library, 299  
serialEvent() function, Processing, 311, 313, 322  
serial library, Processing, 310, 321  
Serial.list() function, Processing, 301, 311, 322  
serial monitor, 108, 109, 111, 112, 125, 147, 211, 218, 235, 247, 253  
Serial Peripheral Interface (SPI), 268  
serial port, 41–43, 86, 113, 147, 210, 216, 236, 298, 300, 310, 311, 320, 322  
Serial.print() function, Arduino, 113–114  
Serial.println() function, Arduino, 113–114

- Servo library, 129, 133, 134, 268
- servomotors, 28, 117, 126–127
  - Knob sketch, 132–135
  - Sweep sketch, 127–131
- setup() function, Arduino, 49–50, 92, 100, 113, 122, 130, 135, 142, 147, 199, 204, 210, 216, 222, 236, 242, 253, 303
- shapes, drawing (Processing), 281–287
- shields, 20, 256. *See also* Arduino shield
  - audio and display, 259–261
  - combinations, considerations for, 256–257
  - input, 261–262
  - latest products, 265–266
  - legacy, 263–265
  - motor and power, 262–263
  - prototyping, 258–259
- short circuit, 71, 121
- sidebar, Arduino IDE, 33
- signal wires, 90
- sine waves, 136, 137
- single-core wires, 60, 61, 176
- single-line comments, 48
- size() function, Processing, 282, 291
- sketches, 10, 32, 85. *See also* libraries
  - AnalogInOutSerial, 109–114, 186
  - AnalogInput, 102–108
  - Blink, 40–41, 43–44, 193–194
  - BlinkWithoutDelay, 195–200
  - Button, 97–102
  - Calibration, 219–224
  - CapacitiveTouch, 245–248
  - Debounce, 200–205
  - DigitalInputPullup, 227–231
  - example sketches in libraries, 266, 270
  - HC-SR04, 250–254
  - Knob, 132–135
  - Knock, 232–237
  - making virtual button, 293–305
  - Motor, 118–122
  - MotorControl, 123–126
  - PitchFollower, 144–147
  - Processing, 280–291
  - resetting, 21
  - SerialCallResponse, 314–323
  - Smoothing, 213–218
  - StateChangeDetection, 206–212
  - Sweep, 127–131
  - toneKeyboard, 239–243
  - toneMelody, 136–143
  - uploading, 86
- Smoothing sketch, 213–218
  - circuit, 213–214
  - code, 214–215
  - hardware, 213
- sockets (multimeter), 63
- SoftwareSerial library, 268
- soldering, 59, 163–164, 183–184, 187–188
  - adhesive putty, 172, 182, 187
  - assembling a Arduino shield, 178–182

- equipment wire, 176–177
- multimeter, 174
- needle-nosed pliers, 174
- safety precautions, 177–178
- solder, 170–171, 178
- solder fumes, 165, 171, 178
- solder stations, 169–170, 183
- solder sucker, 174–176
- solder wick, 175–176
- technique, 182–185
- thinning the tip, 184, 185
- third hand (helping hand), 171–172, 182
- wire cutters, 173
- wire strippers, 173–174
- workspace for, 164–165
- soldering irons
  - cleaning, 178
  - handling, 177
  - tips, 170
  - types of, 165–169
- source code, 15
- SparkFun, 26, 265, 328, 332
- SparkFun Inventor's Kit (v5.0), 29
- SparkFun MIDI shield, 261–262
- Speak & Spell, 12
- Special Projects, 153
- SPI. *See* Serial Peripheral Interface
- square waves, 136, 137
- starter kits, 27–30
- StateChangeDetection sketch, 206–212

- circuit, 206–207
- code, 207–209
- hardware, 206
- understanding, 209–212
- Stepper library, 268
- stripboards, 179–180, 191
- Sweep sketch, 127–131
  - circuit, 128
  - code, 129
  - hardware, 127
  - understanding, 130–131
- synthesizers, 11, 12

## T

- tabs, Arduino IDE, 34, 140
- temperature-controlled soldering irons, 168–169
- temperature sensor, 28
- terminal blocks, 190–191
- test leads (multimeter), 62
- Theremin, 143–147
- third hand for soldering, 171–172, 182
- timers, 194–195, 204
- Tindie, 333
- tone() function, Arduino, 142
- toneKeyboard sketch, 239–243
  - circuit, 239–240
  - code, 240–241
  - hardware, 239
  - understanding, 242–243

- toneMelody sketch, 136–143
  - circuit, 137–138
  - code, 138–140
  - hardware, 137
  - understanding, 140–143
- toolbar, Arduino IDE, 32–33
- TouchDesigner, 278
- touchRead(pin) function, Arduino, 248
- transistors, 29, 81, 82, 119–120
- triangle waves, 136, 137
- trim() function, Processing, 313

## U

- Ultimate GPS Logger shield, Adafruit, 263
- ultrasonic range finders, 249–254
- United Visual Artists (UVA), 151
- Unity, 278
- Universal Serial Bus (USB), 108
  - access, in enclosures, 189
  - Arduino Uno, 20–21
  - USB-C cable, 28, 86
  - USB-C soldering irons, 166–167
- unsigned long, 198
- USB. *See* Universal Serial Bus
- UVA. *See* United Visual Artists

## V

- variable resistors, 66, 103, 109
- variables
  - global, 199

- local, 199
  - pre-defined, 49, 53
- ventilated environment, for soldering, 165, 171, 178
- Verplank, Bill, 8
- Vin (voltage-in) pin, 20, 21
- virtual button, making, 293–294
  - circuit, 294, 295
  - setting up Arduino code, 294–296
  - setting up Processing code, 296–299
  - understanding Arduino PhysicalPixel sketch, 303–305
  - understanding Processing PhysicalPixel sketch, 299–303
- Vitamins Design, 153
- void draw, 291
- void keyword, 49
- void loop, 49
- void setup, 49
- voltage, 70
  - measurement of, 64
  - Ohm’s Law, 71–73

## W

- Waste Electrical and Electronic Equipment Directive (WEEE), 170
- Watterott Electronic, 26
- Watterott, Stephan, 26
- watts, 70, 74–75
- WEEE. *See* Waste Electrical and Electronic Equipment Directive
- Weller solder station, 169, 183
- Wheatstone bridge, 238

- while loop, 222, 320
- WiFi library, 267
- WiFiNINA library, 267–268
- WiFiS3 library, 267–268
- wire cutters, 173
- Wire (I<sup>2</sup>C/TWI) library, 268
- wire strippers, 173–174
- wiring, 190–191
  - terminal blocks, 190–191
  - twisting and braiding, 190
- Wiring project, 9, 10

- workshops, 330
- workspace
  - for soldering, 164–165
  - for working on Arduino projects, 30–31

## **Y**

- YouTube, 329

## **Z**

- zero indexing, 134, 141
- ZIP file, library installation from, 270



## About the Author

---

John Nussey is a designer, technologist, and educator based in Folkestone, United Kingdom.

He specializes in transforming ideas into real electronic products, drawing on a career that spans physical computing, prototyping, and human-centred design. His work includes commercial hardware development, interactive installations, and exploratory research into the future of technology.

Since 2007, John has worked with organizations ranging from early-stage start-ups to global companies, including the BBC, Logitech, and Google. He is co-founder of ONN Studio, a design consultancy that helps teams develop thoughtful, well-engineered products from early concepts through to manufacture.

Teaching has been a central thread in John's career. He has introduced thousands of people to Arduino and physical computing through workshops, talks, and university courses. He has taught at institutions including Goldsmiths, the Bartlett School of Architecture, the Royal College of Art, and the Copenhagen Institute of Interaction Design, where he continues to champion playful, accessible approaches to learning electronics.

## Dedication

---

To Avril O'Neil, the love of my life (and the only person I trust with a soldering iron), for sharing so many adventures and always supporting me when it matters most; and to Alexandra Deschamps-Sonsino and Massimo Banzi, whose early encouragement and generosity introduced me to Arduino and helped set the course of my career.

# Author's Acknowledgments

---

I would like to thank the team at Wiley — especially Jennifer Yee, Chris Morris, Ajith Kumar, and everyone involved in guiding this new edition into print. My thanks also go to Guy Hart-Davis for his excellent technical editing and careful attention to detail.

A heartfelt thank you to my friends and family for their support, and to the Arduino community and *Arduino For Dummies* readers I've met over the years. Your enthusiasm, curiosity, and willingness to experiment are a constant source of motivation. I hope this book encourages you to keep exploring, keep learning, and keep making.

## **Publisher's Acknowledgments**

**Acquisitions Editor:** Jennifer Yee

**Project Editor:** Christopher Morris

**Copy Editor:** Christopher Morris

**Technical Editor:** Guy Hart-Davis

**Managing Editor:** Ajith Kumar

**Production Editor:** Athiyappan Lalith Kumar

**Cover Image:** Arduino® UNO R4 Minima,  
courtesy of Arduino SRL

# **WILEY END USER LICENSE AGREEMENT**

Go to [www.wiley.com/go/eula](http://www.wiley.com/go/eula) to access Wiley's ebook EULA.