

EXPLORING ARDUINO®

EXPLORING ARDUINO®

Tools and Techniques for Engineering Wizardry

Second Edition

Jeremy Blum

WILEY

Exploring Arduino®: Tools and Techniques for Engineering Wizardry

Published by

John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2020 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-119-40537-5
ISBN: 978-1-119-40535-1 (ebk)
ISBN: 978-1-119-40530-6 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

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 either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. 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 <http://www.wiley.com/go/permissions>.

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 Web site 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 United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

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.

Library of Congress Control Number: 2019948860

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Arduino is a registered trademark of Arduino AG Corporation. 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.

*To Leah, for helping me to see every
challenge as an opportunity.*

About the Author

Jeremy Blum is currently the director of engineering at Shaper (shapertools.com), where he is using computer vision to reinvent the way people use handheld power tools. Prior to joining Shaper, Jeremy was a lead electrical architect/engineer for confidential products at Google [x], including Google Glass.

Jeremy received his master's and bachelor's degrees in Electrical and Computer Engineering from Cornell University. At Cornell, he co-founded and led Cornell University Sustainable Design, he launched a first-of-its-kind entrepreneurial co-working space for students, and he conducted robotics and machine learning research.

Jeremy has designed prosthetic hands, fiber-optic LED lighting systems, home-automation systems, 3D printers and scanners, self-assembling robots, wearable computing platforms, augmented reality devices, and learning robots. His work has been featured in international conferences, peer-reviewed journals, and popular media outlets such as Discovery Channel, the *Wall Street Journal*, and *Popular Science* magazine. *Forbes* magazine named him to their annual "30 Under 30" list in recognition of his work that has "helped America make things and get stuff done." He is the co-author of several patents in the fields of wearable computing and augmented reality fabrication.

When not building products, Jeremy is teaching. His written and video tutorials have been utilized by millions of people to learn electrical engineering and embedded software design. His book, *Exploring Arduino*, has been translated into multiple languages and is used as an engineering textbook around the world, including at his alma mater, Cornell. Jeremy's passion is using engineering to improve people's lives, and giving people the tools they need to do the same. You can learn more about Jeremy at his website, jeremyblum.com.

About the Technical Editor

Dr. Derek Molloy is an associate professor in the Faculty of Engineering and Computing's School of Electronic Engineering at Dublin City University, Ireland. He lectures at undergraduate and postgraduate levels in object-oriented programming with embedded systems, digital and analog electronics, and connected embedded systems. His research contributions have largely been in the fields of computer and machine vision, embedded systems, 3D graphics/visualization, and e-learning. Derek produces

viii About the Technical Editor

a popular YouTube video series that has introduced millions of people to embedded Linux and digital electronics topics. In 2013, he launched a personal web/blog site that is visited by thousands of people every day and that integrates his YouTube videos with support materials, source code, and user discussion. He has published other books in this Wiley mini-series: *Exploring BeagleBone* in 2015, followed in 2016 by *Exploring Raspberry Pi*. The second edition of *Exploring BeagleBone* was released earlier this year. You can learn more about Derek, his work, and his other publications at his personal website, derekmolloy.ie.

Acknowledgments

In the several years since the first edition of this book was released, I've received so many notes from readers who have told me about the many ways that they've learned from *Exploring Arduino*. I've also received plenty of constructive criticism—little things that I can adjust to improve the book's utility. I've taken all these comments to heart and have tracked them carefully over the past few years. It is my intention to make this second edition even more useful than the first, while still maintaining the approachability that many readers told me that they appreciated. So, THANK YOU to everybody who has given me feedback about the first edition of *Exploring Arduino*!

Second, I must extend my thanks again to Wiley. They've been amazing partners through this journey, and I'm glad to have them to continue to see this book through to a second edition. In particular, I'd like to thank Jim Minatel, Adaobi Obi Tulton, Dr. Derek Molloy, Marylouise Wiack, and Athiyappan Lalith Kumar.

Thanks also to the wonderful folks at Adafruit, who have collaborated with me on ensuring that parts kits are easy to obtain for this book. Adafruit contributes heavily to the open source hardware and software communities, and I certainly would not be the engineer that I am today without their excellent products and guides.

Back when I wrote the first edition of *Exploring Arduino*, I was still getting my master's degree. I've long since graduated, but now I've got my work at Shaper to focus on. I owe a big thanks to all my co-workers both at Shaper and at Google (my previous employer) for always encouraging me, and for building awesome hardware with me!

I want to give a special shout-out to my professors at Cornell, especially Professor François Guimbretière, who taught the course where I was first introduced to Arduino. He has since used the first edition of this book as a textbook for that course, and it makes me so happy to know that I've been able to give back to Cornell in that capacity.

Finally, I want to thank my parents, my brother, my wife, and my friends for putting up with me, and for always encouraging me. I feel so fortunate to have such wonderful people in my life.

Contents at a Glance

Introduction	xxv
PART I Arduino Engineering Basics	1
1 Getting Started and Understanding the Arduino Landscape	3
2 Digital Inputs, Outputs, and Pulse-Width Modulation	23
3 Interfacing with Analog Sensors	47
PART II Interfacing with Your Environment	67
4 Using Transistors and Driving DC Motors	69
5 Driving Stepper and Servo Motors	99
6 Making Sounds and Music	125
7 USB Serial Communication	141
8 Emulating USB Devices	171
9 Shift Registers.....	183
PART III Communication Interfaces	199
10 The I ² C Bus	201
11 The SPI Bus and Third-Party Libraries	223
12 Interfacing with Liquid Crystal Displays.....	247
PART IV Digging Deeper and Combining Functions	273
13 Interrupts and Other Special Functions	275
14 Data Logging with SD Cards.....	295

PART V	Going Wireless.....	337
15	Wireless RF Communications.....	339
16	Bluetooth Connectivity	363
17	Wi-Fi and the Cloud	399
	Appendix A: Deciphering Datasheets and Schematics.....	451
	Index.....	461

Contents

Introduction	XXV
PART I Arduino Engineering Basics	1
1 Getting Started and Understanding the Arduino Landscape	3
Exploring the Arduino Ecosystem	4
Arduino Functionality	5
The Microcontroller	7
Programming Interfaces.....	8
Input/Output: GPIO, ADCs, and Communication Busses	9
Power	9
Arduino Boards	11
Creating Your First Program.....	15
Downloading and Installing the Arduino IDE.....	16
Running the IDE and Connecting to the Arduino.....	17
Breaking Down Your First Program	18
Summary	21
2 Digital Inputs, Outputs, and Pulse-Width Modulation	23
Digital Outputs	24
Wiring Up an LED and Using Breadboards	24
Working with Breadboards.....	24
Wiring LEDs	25
Programming Digital Outputs.....	29
Using For Loops.....	30
Pulse-Width Modulation with <i>analogWrite()</i>	31
Reading Digital Inputs	35
Reading Digital Inputs with Pull-Down Resistors	35
Working with "Bouncy" Buttons	38
Building a Controllable RGB LED Nightlight.....	42
Summary	46

3 Interfacing with Analog Sensors	47
Understanding Analog and Digital Signals	48
Comparing Analog and Digital Signals	48
Converting an Analog Signal to Digital	49
Reading Analog Sensors with the Arduino: <i>analogRead()</i>	51
Reading a Potentiometer	51
Using Analog Sensors.....	56
Using Variable Resistors to Make Your Own Analog Sensors.....	60
Using Resistive Voltage Dividers.....	61
Using Analog Inputs to Control Analog Outputs.....	64
Summary	66
PART II Interfacing with Your Environment	67
4 Using Transistors and Driving DC Motors	69
Driving DC Motors	70
Handling High-Current Inductive Loads	71
Using Transistors as Switches	72
Using Protection Diodes	73
Using a Secondary Power Source	74
Wiring the Motor	74
Controlling Motor Speed with PWM.....	76
Using an H-Bridge to Control DC Motor Direction.....	78
Building an H-Bridge Circuit.....	80
Operating an H-Bridge Circuit	82
Building a Roving Robot	86
Choosing the Robot Parts	87
Selecting a Motor and Gearbox	87
Powering Your Robot.....	87
Constructing the Robot	89
Writing the Robot Software.....	92
Bringing It Together	96
Summary	97

5 Driving Stepper and Servo Motors	99
Driving Servo Motors.....	100
Understanding the Difference between Continuous Rotation and Standard Servos.....	100
Understanding Servo Control.....	101
Controlling a Servo	104
Building a Sweeping Distance Sensor.....	105
Understanding and Driving Stepper Motors.....	109
How Bipolar Stepper Motors Work.....	111
Making Your Stepper Move	113
Building a “One-Minute Chronograph”.....	117
Wiring and Building the Chronograph.....	117
Programming the Chronograph.....	119
Summary	124
6 Making Sounds and Music	125
Understanding How Speakers Work.....	126
The Properties of Sound.....	126
How a Speaker Produces Sound	128
Using <i>tone()</i> to Make Sounds.....	129
Including a Definition File	129
Wiring the Speaker	130
Making Sound Sequences	133
Using Arrays	133
Making Note and Duration Arrays.....	134
Completing the Program.....	134
Understanding the Limitations of the <i>tone()</i> Function	136
Building a Micro Piano.....	136
Summary	139
7 USB Serial Communication	141
Understanding the Arduino’s Serial Communication Capabilities.....	142
Arduino Boards with an Internal or External FTDI or Silicon Labs USB-to-Serial Converter	143

Arduino Boards with a Secondary USB-Capable ATmega MCU	146
Emulating a Serial Converter	146
Arduino Boards with a Single USB-Capable MCU	147
Arduino Boards with USB-Host Capabilities	147
Listening to the Arduino	148
Using <i>print</i> Statements	148
Using Special Characters	150
Changing Data Type Representations	152
Talking to the Arduino	152
Configuring the Arduino IDE's Serial Monitor to Send Command Strings	152
Reading Incoming Data from a Computer or Other Serial Device	153
Telling the Arduino to Echo Incoming Data	153
Understanding the Differences between Chars and Ints	154
Sending Single Characters to Control an LED	156
Sending Lists of Values to Control an RGB LED	158
Talking to a Desktop App	161
Installing Processing	162
Controlling a Processing Sketch from Your Arduino	163
Sending Data from Processing to Your Arduino	166
Summary	169
8 Emulating USB Devices	171
Emulating a Keyboard	173
Typing Data into the Computer	173
Commanding Your Computer to Do Your Bidding	177
Emulating a Mouse	178
Summary	182
9 Shift Registers	183
Understanding Shift Registers	184
Sending Parallel and Serial Data	185
Working with the 74HC595 Shift Register	186
Understanding the Shift Register pin Functions	186
Understanding How the Shift Register Works	187

Shifting Serial Data from the Arduino	189
Converting Between Binary and Decimal Formats	192
Controlling Light Animations with a Shift Register.....	192
Building a “Light Rider”.....	192
Responding to Inputs with an LED Bar Graph	194
Summary	197
PART III Communication Interfaces	199
10 The I²C Bus	201
History of the I ² C Bus.....	202
I ² C Hardware Design	203
Communication Scheme and ID Numbers	203
Hardware Requirements and Pull-Up Resistors.....	206
Communicating with an I ² C Temperature Probe	208
Setting Up the Hardware	208
Referencing the Datasheet	210
Writing the Software.....	212
Combining Shift Registers, Serial Communication, and I ² C Communications.....	214
Building the Hardware for a Temperature Monitoring System.....	214
Modifying the Embedded Program	215
Writing the Processing Sketch.....	218
Summary	221
11 The SPI Bus and Third-Party Libraries	223
Overview of the SPI Bus.....	224
SPI Hardware and Communication Design	225
Hardware Configuration.....	225
Communication Scheme	227
Comparing SPI to I ² C and UART	227
Communicating with an SPI Accelerometer.....	228
What Is an Accelerometer?.....	229
Gathering Information from the Datasheet.....	231
Setting Up the Hardware	233

Writing the Software	235
Installing the Adafruit Sensor Libraries	236
Leveraging the Library	237
Creating an Audiovisual Instrument Using a 3-Axis Accelerometer	241
Setting Up the Hardware	242
Modifying the Software	242
Summary	246
12 Interfacing with Liquid Crystal Displays	247
Setting Up the LCD	248
Using the LiquidCrystal Library to Write to the LCD	251
Adding Text to the Display	252
Creating Special Characters and Animations	254
Building a Personal Thermostat	258
Setting Up the Hardware	258
Displaying Data on the LCD	261
Adjusting the Set Point with a Button	264
Adding an Audible Warning and a Fan	265
Bringing It All Together: The Complete Program	266
Taking This Project to the Next Level	270
Summary	271
PART IV Digging Deeper and Combining Functions	273
13 Interrupts and Other Special Functions	275
Using Hardware Interrupts	276
Knowing the Tradeoffs Between Polling and Interrupting	277
Ease of Implementation (Software)	277
Ease of Implementation (Hardware)	277
Multitasking	278
Acquisition Accuracy	278
Understanding the Arduino Hardware Interrupt Capabilities	278

Building and Testing a Hardware-Debounced Button Interrupt Circuit	279
Creating a Hardware-Debouncing Circuit	280
Assembling the Complete Test Circuit	284
Writing the Software	285
Using Timer Interrupts	288
Understanding Timer Interrupts	288
Getting the Library	289
Executing Two Tasks Simultaneously(ish)	289
Building an Interrupt-Driven Sound Machine	290
Sound Machine Hardware	291
Sound Machine Software	291
Summary	294
14 Data Logging with SD Cards	295
Getting Ready for Data Logging	296
Formatting Data with CSV Files	297
Preparing an SD Card for Data Logging	297
Formatting Your SD Card Using a Windows PC	298
Formatting Your SD Card Using Mac OS	300
Formatting Your SD Card Using Linux	302
Interfacing the Arduino with an SD Card	304
SD Card Shields	304
SD Card SPI Interface	307
Writing to an SD Card	307
Reading from an SD Card	312
Real-Time Clocks	317
Understanding Real-Time Clocks	317
Communicating with a Real-Time Clock	317
Using the RTC Arduino Third-Party Library	318
Using a Real-Time Clock	319
Installing the RTC and SD Card Modules	319
Updating the Software	320

Building an Entrance Logger	327
Logger Hardware.....	328
Logger Software	329
Data Analysis	334
Summary	335
PART V Going Wireless.....	337
15 Wireless RF Communications.....	339
The Electromagnetic Spectrum	340
The Spectrum.....	342
How Your RF Link Will Send and Receive Data	343
Receiving Key Presses with the RF Link.....	346
Connecting Your Receiver	346
Programming Your Receiver	347
Making a Wireless Doorbell	351
Wiring the Receiver.....	351
Programming the Receiver	351
The Start of Your Smart Home—Controlling a Lamp.....	354
Your Home’s AC Power	356
How a Relay Works	356
Programming the Relay Control	358
Hooking up Your Lamp and Relay to the Arduino	360
Summary	361
16 Bluetooth Connectivity	363
Demystifying Bluetooth.....	364
Bluetooth Standards and Versions.....	364
Bluetooth Profiles and BTLE GATT Services	365
Communication between Your Arduino and Your Phone	366
Reading a Sensor over BTLE	366
Adding Support for Third-Party Boards to the Arduino IDE.....	367
Installing the BTLE Module Library.....	369
Programming the Feather Board	369

Connecting Your Smartphone to Your BTLE Transmitter	377
Sending Commands from Your Phone over BTLE	379
Parsing Command Strings	380
Commanding Your BTLE Device with Natural Language.....	384
Controlling an AC Lamp with Bluetooth	389
How Your Phone “Pairs” to BTLE Devices	389
Writing the Proximity Control Software	390
Pairing Your Phone	394
Pairing an Android Phone.....	394
Pairing an iPhone	395
Make Your Lamp React to Your Presence	396
Summary	397
17 Wi-Fi and the Cloud	399
The Web, the Arduino, and You	400
Networking Lingo	401
The Internet vs. the World Wide Web vs. the Cloud.....	401
IP Address	401
Network Address Translation.....	402
MAC Address.....	402
HTML.....	402
HTTP and HTTPS.....	402
GET/POST.....	403
DHCP.....	403
DNS	403
Clients and Servers	403
Your Wi-Fi-Enabled Arduino	404
Controlling Your Arduino from the Web	404
Setting Up the I/O Control Hardware.....	404
Preparing the Arduino IDE for Use with the Feather Board.....	406
Ensuring the Wi-Fi Library Is Matched to the Wi-Fi Module’s Firmware	407
Checking the WINC1500’s Firmware Version.....	408
Updating the WINC1500’s Firmware.....	408

Writing an Arduino Server Sketch.....	408
Connecting to the Network and Retrieving an IP Address via DHCP	409
Writing the Code for a Bare-Minimum Web Server	412
Controlling Your Arduino from Inside and Outside Your Local Network.....	423
Controlling Your Arduino over the Local Network	423
Using Port Forwarding to Control Your Arduino from Anywhere	425
Interfacing with Web APIs.....	427
Using a Weather API.....	428
Creating an Account with the API Service Provider	429
Understanding How APIs Are Structured	430
JSON-Formatted Data and Your Arduino	430
Fetching and Parsing Weather Data	431
Getting the Local Temperature from the Web on Your Arduino	433
Completing the Live Temperature Display	440
Wiring up the LED Readout Display.....	440
Driving the Display with Temperature Data	443
Summary	449
Appendix A: Deciphering Datasheets and Schematics.....	451
Index.....	461

Figure Credits

All images, icons, and marks as displayed in Figure 3-7 and Figure 10-3 are owned by Analog Devices, Inc. (ADI), copyright © 2019. All Rights Reserved. These images, icons, and marks are reproduced with permission by ADI. No unauthorized reproduction, distribution, or usage is permitted without ADI's written consent.

This book contains copyrighted material of Microchip Technology Incorporated replicated with permission. All rights reserved. No further replications may be made without Microchip Technology Inc.'s prior written consent.

Atmel, AVR, ICSP, and In-Circuit Serial Programming are trademarks or registered trademarks of Microchip Technology Inc.

Arm and Cortex are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the United States and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs, and trade secrets.

Introduction

When the first edition of this book came out in 2013, I opened it with the following greeting:

You have excellent timing. As I often like to say, “We’re living in the future.”

I think I backed myself into a corner with that introduction, because if 2013 was “the future,” then I’m not quite sure what to call the present! The *far future*? The *future-future*? My point is, the march of progress has been swift, and the possibilities for what you can do with even a cursory knowledge of embedded electronics and software continue to expand every day.

Since the first edition of this book was released, electronics and software have continued to become increasingly accessible with every passing day. In 2013, I was hesitant to include a chapter about connecting your hardware projects to the internet because the process for doing so was still quite fussy. The “Internet of Things” (IoT) was just an emerging nerdy buzzword in 2013. Now, it’s a key part of the global vernacular. It seems like every product for sale nowadays contains a microcontroller. Everything is “smart” and most of those things also feature phone or web connectivity. I bet you didn’t think you’d be buying a Bluetooth-enabled toothbrush back when “Bluetooth” just referred to people talking to themselves through their wireless cellphone headsets.

Considering all this, I felt it was time to release a new edition of *Exploring Arduino*. This second edition expands upon everything that was covered in the first edition. It updates all the projects with new challenges and details, clarifies questions that people had from the first edition, and adds a plethora of new content, including a lot more details on wireless connectivity, new Arduino hardware, changes to the Arduino ecosystem and software, and more.

Why Arduino?

With the tools available to you today, many of which you’ll learn about in this book, you have the opportunity and the ability to bend the physical world to your whim. Until very recently, it has not been possible for someone to pick up a microcontroller and use it to control their world within minutes. A *microcontroller* is a programmable integrated circuit (IC) that gives you the power to define the operation of complex mechanical, electrical, and software systems using relatively simple commands. The possibilities are endless, and the Arduino microcontroller platform will become your new favorite tool as you explore the world of electronics, programming, human-computer interaction,

art, control systems, and more. Throughout the course of this book, you'll use the Arduino to do everything from detecting motion to creating wireless control systems to communicating over the internet.

Whether you are completely new to any kind of engineering or are a seasoned veteran looking to get started with embedded systems design, the Arduino is a great place to start. Are you looking for a general reference for Arduino development? This book is perfect for you, too. It walks you through a number of separate projects, but you'll also find it easy to return to the book for code snippets, best practices, system schematics, and more. The electrical engineering, systems design, and programming practices that you'll learn while reading this book are widely applicable beyond the Arduino platform and will prepare you to take on an array of engineering projects, whether they use the Arduino or some other platform.

Who This Book Is For

This book is for Arduino enthusiasts of all experience levels. Chapters build upon each other, utilizing concepts and project components from previous chapters to develop more complex ideas. But don't worry. Whenever you face new, complex ideas, a cross-reference reminds you where you first encountered any relevant building-block concepts so that you can easily refresh your memory.

This book assumes that you have little or no previous experience working with programming or electrical engineering. Using feedback from readers of the first edition of this book, I've taken special care to be very detailed in my explanation of the more confusing topics you may encounter. To effectively support readers of various experience levels, the book features several optional sections and *sidebars*, or short excerpts, that explain a particular concept in greater detail. Although these sidebars are not necessary for you to gain a good understanding of how to use the Arduino, they do provide a closer look at technical topics for the more curious reader.

What You'll Learn in This Book

This book is not a recipe book. If you want to follow step-by-step instructions that tell you exactly how to build a particular project without actually explaining why you are doing what you are doing, this book is not for you. You can think of this book as an introduction to electrical engineering, computer science, product design, and high-level thinking using the Arduino as a vehicle to help you experience these concepts in a hands-on manner.

When building hardware components of the Arduino projects demonstrated in this book, you'll learn not just how to wire things together, but also how to read schematics,

why particular parts are used for particular functions, and how to read datasheets that will allow you to choose appropriate parts to build your own projects. When writing software, I provide complete program code, but you will first be stepped through several iterative processes to create the final program. This will help to reinforce specific program functions, good code-formatting practices, and algorithmic understanding.

This book will teach physics concepts, algorithms, digital design principles, and Arduino-specific programming concepts. It is my hope that working through the projects in this book will not just make you a well-versed Arduino developer, but also give you the skills you need to develop more-complex electrical systems, and to pursue engineering endeavors in other fields, and with different platforms.

Features Used in This Book

The following features and icons are used in this book to help draw your attention to some of the most important or useful information in the book:

WARNING Be sure to take heed when you see one of these asides. They appear when particular steps could cause damage to your electronics if performed incorrectly.

TIP These asides contain quick hints about how to perform the task at hand more easily and effectively.

NOTE These asides contain additional information that may be of importance to you, including links to videos and online material that will make it easier to follow along with the development of a particular project.

SAMPLE HEADING

These asides go into additional depth about the current topic or a related topic.

Getting the Parts

In preparing the projects outlined in this book, I've taken special care to use components that are readily available through a variety of retailers, both in the United States and internationally. I've also partnered with Adafruit (adafruit.com), a popular retailer

of hobbyist electrical components. You can purchase all the components required for completing the projects in this book from Adafruit. A convenient listing of Adafruit parts for each chapter is available at exploringarduino.com/kits.

At the beginning of each chapter, you'll find a detailed list of parts that you need to complete that chapter—all of these parts are available from many sources. The companion website for this book, www.wiley.com/go/exploringarduino2e, also provides links to multiple sources where you can find the parts for each chapter.

What You'll Need

In addition to the actual parts that you'll use to build your Arduino projects, there are a few other tools and materials that you'll need on your Arduino adventures. Most importantly, you'll need a computer that is compatible with the Arduino integrated development environment (IDE) (Mac OS X 10.7 Lion or newer, Windows XP or later, or a Linux distro). I will provide instructions for all operating systems when warranted.

Arduino now also has an entirely web-based editor, but this book will generally focus on the desktop IDE. All the instructions for the desktop software generally apply to the online IDE as well. The first version of this book was read by people all over the world, representing a wide range of internet speeds and reliability. To ensure that Arduino remains easily accessible to all, I'll mostly provide instructions that use the offline IDE, as constant internet access isn't always an option for everybody.

You may also want some additional tools that will be used throughout the book to debug and assemble hardware. These tools are not only necessary to complete the projects in this book. As you develop your electrical engineering skillset, they will come in handy for other projects, too. I recommend the following:

- A soldering iron and solder (Note: A few shields and microcontroller boards used in the final chapters of this book may be sold with some soldering required—this usually involves easy soldering of thru-hole pins to a circuit board.)
- A multimeter (This will be useful for debugging concepts within this book, but is not required.)
- A set of small screwdrivers
- Tweezers
- Wire cutters and wire strippers
- A hot glue gun
- A magnifying glass (Electronics are small, and sometimes it's necessary to read the tiny, laser-etched markings on integrated circuits in order to look up their datasheets online.)

Source Code and Digital Content

The primary companion site for this book is exploringarduino.com, and it is maintained by the author. You will find code downloads for each chapter on this site (along with videos, links, and other useful materials). Note that both 1st edition and 2nd edition content is available at this URL—ensure that you are visiting the pages for this edition of the book. Digital content for the first edition is located at exploringarduino.com/content1/ . . . and digital content for the second edition of this book is located at exploringarduino.com/content2/ The website clearly differentiates between content for the two editions of the book and is easy to navigate.

Wiley also maintains a repository of digital content that accompanies this book at wiley.com/go/exploringarduino2e. You can also search for the book at wiley.com by ISBN (the ISBN for this book is 9781119405375) to find links to book resources.

The code for this book is hosted on GitHub.com (a popular platform for sharing open source software). Throughout each chapter, you can find references to the names of code files as needed in listing titles and text. Each chapter’s code packages will be linked from exploringarduino.com and wiley.com. You can download the code as a compressed ZIP archive from either source. After you download the code, just decompress it with an appropriate decompression tool—all operating systems have one built in. You can also pull code directly from this book’s GitHub repository (which is linked from exploringarduino.com) if you are comfortable working with Git-based version control.

NOTE Because many books have similar titles, you may find it easiest to search by ISBN; this book’s ISBN is 9781119405375.

NOTE Some URLs (especially the ones that I don’t control) may change or be very long. To make it easier to type in long URLs that I may reference throughout the book, I will list a “shortened URL” using my personal domain name shortener: blum.fyi. For example, blum.fyi/jarvis redirects to a longer URL on my website about a project called “JARVIS.”

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in this book, such as a spelling mistake or faulty piece of code, we would be grateful for your feedback. By

sending in errata, you may save another reader hours of frustration, and at the same time, you can help us provide even higher-quality information.

To find the errata page for this book, go to wiley.com/go/exploringarduino2e and click the Errata link. On this page, you can view all errata that has been submitted for this book and posted by Wiley editors. I also review all errata reports and post errata notes to exploringarduino.com on each chapter page.

Supplementary Material and Support

During your adventures with your Arduino, you'll inevitably have questions and perhaps run into problems. One of the best aspects of using the Arduino is the excellent support community that you can find on the web. This extremely active base of Arduino users will readily help you along your journey. I maintain a list of updated resources for getting help with Arduino, electrical engineering, and embedded software on the Exploring Arduino Resources page:

exploringarduino.com/resources

I used to try to answer people's individual Arduino questions directly, but that's unfortunately no longer possible due to the sheer volume of questions that I receive through my website, Twitter, Facebook, YouTube, and other channels. I highly encourage you to seek help through the forums linked from the Resources page listed here. I can almost guarantee that their response times will be faster than mine.

What Is an Arduino?

The best part about the Arduino prototyping platform is that it's whatever you want it to be. The Arduino could be an automatic plant-watering control system. It could be a web server. It could even be a quadcopter autopilot.

The Arduino is a microcontroller development platform paired with an intuitive programming language that you develop using the Arduino integrated development environment. By equipping the Arduino with sensors, actuators, lights, speakers, add-on modules (called *shields*), and other integrated circuits, you can turn the Arduino into a programmable "brain" for just about any control system.

It's impossible to cover everything that the Arduino is capable of, because the possibilities are limited only by your imagination. Hence, this book serves as a guide to get you acquainted with the Arduino's functionality by executing several projects that will give you the skills you need to develop your own projects.

You'll learn more about the Arduino and the available variations of the board in Chapter 1, "Getting Started and Understanding the Arduino Landscape." If you're eager to know all the inner workings of the Arduino, you're in luck: It is completely open source, and all the schematics and documentation are freely available on the Arduino website. Appendix A, "Deciphering Datasheets and Schematics," covers some of the Arduino's technical specifications.

An Open Source Platform

If you're new to the world of open source, you are in for a treat. This book does not go into detail about the open source hardware movement, but it is worth knowing a bit about the ideologies that make working with the Arduino so wonderful. If you want a full rundown of what open source hardware is, check out the official definitions on the Open Source Hardware Association website (blum.fyi/OSHW-Definition).

Because the Arduino is open source hardware, all the design files, schematics, and source code are freely available to everybody. This means that you can more easily hack the Arduino to serve a very particular function, and also integrate the Arduino platform into your designs, make and sell Arduino clones, and use the Arduino software libraries in other projects. There are hundreds of Arduino derivative boards available (often with specialized functions added on to them).

The Arduino open source license also permits commercial reuse of their designs (so long as you don't utilize the Arduino trademark on your designs). So, if you use an Arduino to prototype an exciting project and you want to turn it into a commercial product, you can do that.

Be sure to respect the licenses of the source code and hardware that you use throughout this book. Some licenses require that you provide attribution to the original author when you publish a design based on their previous work. Others require that you always share improvements that you make under an equivalent license. This sharing helps the community grow, and leads to all the amazing online documentation and support that you'll often refer to during your Arduino adventures. All code examples that I've written for this book (unless otherwise specified) are licensed under the MIT License, enabling you to use them for anything you want.

Beyond This Book

You may already be familiar with my popular series of YouTube Arduino and electronics tutorials (youtube.com/sciguy14). I refer to them throughout this book as a way

to see more-detailed walkthroughs of the topics covered here. If you’re curious about some of the remarkable things that you can do with clever combinations of electronics, microcontrollers, computer science, and creativity, check out my portfolio (jeremyblum.com/portfolio) for a sampling of projects. Like Arduino, most of what I do is released via open source licenses that allow you to easily duplicate my work for your own needs.

I’m anxious to hear about what you do with the skills you acquire from this book. I encourage you to share them with me and with the rest of the world (use the tag *#ExploringArduino* on social media). Good luck on your Arduino adventures!



Arduino Engineering Basics

- Chapter 1:** Getting Started and Understanding the Arduino Landscape
- Chapter 2:** Digital Inputs, Outputs, and Pulse-Width Modulation
- Chapter 3:** Interfacing with Analog Sensors

1

Getting Started and Understanding the Arduino Landscape

What You'll Need for This Chapter:

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at: exploringarduino.com/content2/ch1

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

Whether you are a weekend warrior looking to learn something new, an aspiring electrical engineer, or a software developer looking to better understand the hardware that runs your code, getting your first Arduino project up and running is sure to be an energizing experience. The preface of this book should have already given you some perspective on the Arduino platform and its capabilities; now it's time to explore your options in the world of Arduino. In this chapter, you will examine the available hardware, learn about the programming environment and language, and get your first program up and running. Once you understand the functionality that the Arduino can provide, you'll write your first program and get the Arduino to blink!

NOTE To follow along with a video that introduces the Arduino platform, visit the Chapter 1 content page for the second edition of this book at exploringarduino.com/content2/ch1.

Exploring the Arduino Ecosystem

In your adventures with the Arduino, you'll depend on three main components for your projects:

- First-party or third-party Arduino boards
- External hardware (including both shields and manually created circuits, which you'll explore throughout this book)
- The Arduino integrated development environment, or Arduino IDE

All these system components work in tandem to enable you to accomplish just about anything with your Arduino.

You have a lot of options when it comes to Arduino development boards. Most of this book uses Arduino boards designed by Arduino. Some of the final chapters leverage Arduino-compatible hardware that is designed by third parties to add features like Bluetooth and Wi-Fi to the standard Arduino offerings. Many third-party Arduino boards are directly compatible with Arduino software, libraries, hardware, etc. Some of these boards are designed to be exact clones of official Arduino boards, while others add their own features or capabilities. All the boards used in this book are programmable via the same IDE. When relevant, this book will list caveats about using different boards for various projects. Most of the projects in this book use the Arduino Uno because it has become the de facto introductory board for learning Arduino. You can freely substitute the Adafruit METRO 328 board in places where the Uno is called for—it is functionally identical. You'll see it used in place of the Uno in some of the photos and videos that accompany this book. Most introductory tutorials that you'll find on the web use the Uno or a variant of it. If you do use the Adafruit METRO 328, you may need to install the drivers for it to be detected as an Arduino Uno on Windows. Download and run the installer from blum.fyi/adafruit-windows-drivers.

WARNING Beware of Counterfeits. Only buy Arduino boards and Arduino-compatible boards from reputable sources (such as those listed throughout this book). There are many companies that manufacture clones of popular Arduino boards with inferior components.

You will start by exploring the basic functionality that is found in every Arduino board. Then you will examine the differences between each modern board so that you can make an informed decision when choosing a board to use for your next project.

THE GREAT ARDUINO SCHISM AND REFORMATION

Before you jump into understanding the available options in the Arduino ecosystem, I need to talk about the elephant in the room: the Great Arduino Schism and Reformation (not an official name). Between the release of the first edition of this book and the release of the second edition, the people behind the Arduino hardware and software had a falling out. I won't go into the details here, or pick a side. Basically, Arduino split into two entities represented by two websites: Arduino.cc and Arduino.org. Each group started producing slightly different hardware offerings, forked the codebase, and made conflicting claims about which hardware was genuine. Thankfully, the two sides of this battle have since reconciled their differences and we're back to one Arduino again. Throughout this book, I'll generally talk about the hardware offerings from Arduino.cc, though by the time you get this book, the two Arduinos should be one again. If you'd like to learn more about this nerdy drama, Hackaday.com did a series of reports on it. You can read about the resolution at blum.fyi/arduino-vs-arduino.

Arduino Functionality

All Arduino boards have a few key capabilities and functions. Take a moment to examine the Arduino Uno shown in Figure 1-1; it will be your base configuration. These are some functional groups that you'll be concerning yourself with:

- **Microcontroller:** At the heart of every Arduino is a microcontroller. This is the brain of your Arduino.
- **Programming:** Programming interfaces enable you to load software onto your Arduino.
- **I/O:** Input/Output (I/O) circuitry is what enables your Arduino interface with sensors, actuators, etc.
- **Power:** There are a variety of ways to supply power to an Arduino. Most Arduino boards can automatically switch between power from multiple sources (such as USB and a battery).

6 Exploring Arduino

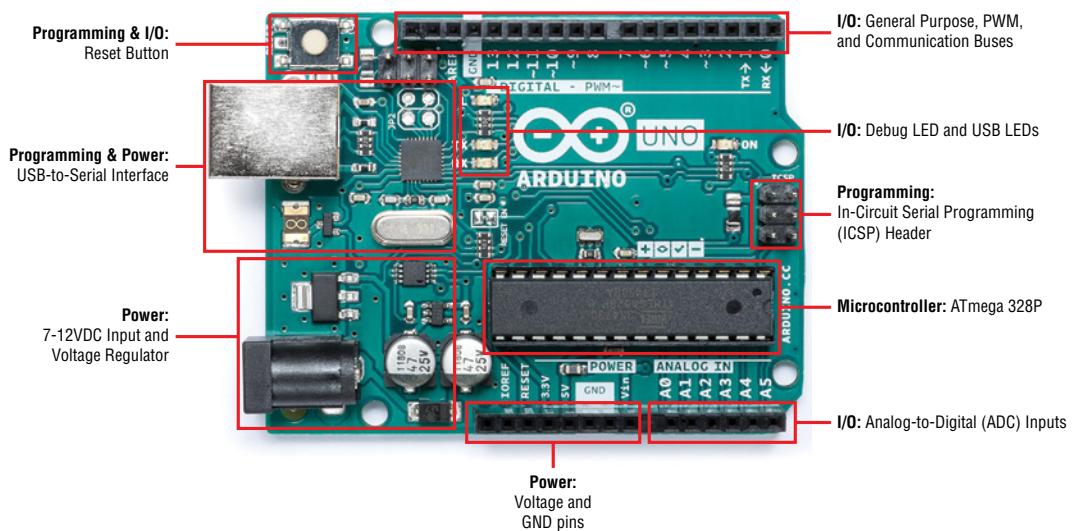


Figure 1-1: Arduino Uno components

Credit: Arduino, arduino.cc; annotations by author

The Microcontroller

At the heart of every Arduino is a microcontroller unit (MCU). All the original Arduino boards, including the Arduino Uno, used an 8-bit Atmel® ATmega microcontroller based on the AVR® architecture. The Arduino Uno in Figure 1-1, for example, uses an ATmega 328P. For most projects that you'll want to build, a simple 8-bit MCU like this one will be more than enough for your needs, so that's what you'll use throughout most of the exercises in this book.

NOTE MICROCHIP AND ATMEL Microchip, a chip manufacturer famous for making the PIC series of microcontrollers, recently acquired Atmel. ATmega chip production has continued under this new brand. Therefore, you may see Microchip and Atmel used interchangeably in reference to the manufacturer of ATmega microcontroller chips. The chips are functionally identical if they have the same part number.

NON-AVR MICROCONTROLLER ARCHITECTURES

But what about when you want to start doing crazy things like synthesizing music, running a web server, and driving massive LED displays? Though possible with clever and efficient programming on an 8-bit MCU, some of these needs are better served by faster and more capable processors.

As an answer to this, in recent years, Arduino has been expanding the range of available Arduino boards to include some that run on Intel (x86 and ARC—Argonaut RISC Core) architectures, and some that use Arm® (Advanced RISC Machine) architectures. The Arduino Due, for example, uses a 32-bit Arm Cortex®-M3 microprocessor. This Cortex processor is faster and contains more peripherals than the 8-bit AVR MCU, thus enabling the Due to do things like play music. Other new Arduino boards add functionality like built-in Wi-Fi and Bluetooth, which is also facilitated by faster and more capable processors. I'll touch on some of these boards later in this chapter, and you'll also get the opportunity to build projects with them later in this book.

You don't need to understand the intricacies of processor architectures to program or use an Arduino—it's all abstracted away for you. However, some people like to know what underlies their hardware projects. The following list will help clarify the buzzwords you just read:

- 8-bit architecture—An MCU architecture type where all addresses, integers, and other key data types are represented as 8-bit numbers.
- 32-bit architecture—An MCU architecture type where all the addresses, integers, and other key data types are represented as 32-bit numbers.

(Continued)

(Continued)

- Microchip (previously Atmel)—A company that makes microcontrollers. Microchip/Atmel makes both AVR MCUs and Arm processors. Most Arduinos use processors that are made by Microchip/Atmel.
- AVR—A microcontroller architecture developed by Atmel for their ATmega MCUs.
- Arm—A collection of 32/64-bit processor architectures developed by a company of the same name. Arm licenses its embedded architecture designs to be used by companies like Microchip and others.
- Cortex-M Series—Cortex M0, M3, and so on represent microprocessor Arm architectures.

The Arduino's microcontroller is responsible for holding all your compiled code and executing the commands you specify. The Arduino programming language gives you access to microcontroller peripherals, including analog-to-digital converters (ADCs), general-purpose input/output (GPIO or just I/O) pins, communication buses (including I²C, SPI, UART, and others), and serial/USB interfaces. Utilizing copper wires etched into the Arduino's printed circuit board, all of this useful functionality is routed from the tiny pins on the microcontroller to accessible headers on the Arduino that you can plug wires or shields into. In the case of the Uno, a 16 MHz ceramic resonator or oscillating crystal is wired to the ATmega's clock pins, which serves as the reference by which all program commands execute. You can use the Reset button to restart the execution of your program. Most Arduino boards come with a debug LED already connected to pin 13, which enables you to run your first program (blinking an LED) without connecting any additional circuitry.

Programming Interfaces

Ordinarily, microcontroller programs are written in C or assembly, and programmed via the In-Circuit Serial Programming™ (ICSP™) interface using a dedicated programmer (see Figure 1-2). Perhaps the most important characteristic of an Arduino is that you can program it directly using only an ordinary USB cable. This functionality is made possible by the Arduino bootloader. The bootloader is loaded onto the microcontroller at the factory (using the ICSP header), which allows a serial USART (Universal Synchronous/Asynchronous Receiver/Transmitter) to load your program on the Arduino without using a separate programmer. (You can learn more about how the bootloader functions in the sidebar, “The Arduino Bootloader and Firmware Setup.”)

In the case of the Arduino Uno and Mega 2560, a secondary microcontroller (an ATmega16U2 or ATmega8U2, depending on your revision) serves as an interface

between a USB cable and the serial USART pins on the main microcontroller. In the Adafruit METRO 328, a Silicon Labs bridge chip is used in place of the ATmega 16U2, but its function is equivalent. The Arduino Leonardo, which uses an ATmega32U4 as the main microcontroller, has USB incorporated, so a secondary microcontroller is not needed. The Arduino M0 uses a Cortex M0 that also includes USB functionality, so it doesn't need a secondary USB chip. In older Arduino boards, an FTDI brand USB-to-serial chip was used as the interface between the ATmega's serial USART port and a USB connection. It's still a popular solution when creating your own Arduino-compatible product.



Figure 1-2: AVRISP mkII programmer

*Credit: © Microchip Technology Incorporated.
Used with permission.*

Input/Output: GPIO, ADCs, and Communication Busses

The part of the Arduino that you'll care most about during your projects is the general-purpose Input/Output (GPIO) and ADC pins. All of these pins can be individually addressed via the programs you'll write. These pins can serve as digital inputs and outputs. The ADC pins can also act as analog inputs that can measure voltages between 0V and 5V (usually from sensors). Many of these pins are also multiplexed to serve special functions, which you will explore later in this book. These special functions include various communication interfaces, serial interfaces, pulse-width-modulated outputs, and external interrupts.

Power

For most of your projects, you will simply use the 5V power that is provided over your USB cable. However, when you're ready to untether your project from a computer, you have other power options. Most Arduinos can accept between 6V and 20V (7V to 12V is the recommended voltage supply range) via the direct current (DC) barrel jack connector,

or into the VIN pin. Some Arduinos operate at 5V logic levels, and others operate at 3.3V logic levels. For 5V Arduinos, like the Uno, the power is configured as follows:

- 5V is used for all the logic on the Uno board. In other words, when you toggle a digital I/O pin, you are toggling it between 5V and 0V.
- 3.3V is broken out to a pin to accommodate 3.3V shields and external circuitry.

For most projects in this book, you can generally assume the use of a 5V Arduino, unless I explicitly specify otherwise.

THE ARDUINO BOOTLOADER AND FIRMWARE SETUP

A *bootloader* is a chunk of code that lives in a reserved space in the program memory of the Arduino’s main MCU. In general, AVR microcontrollers are programmed with an ICSP, which talks to the microcontroller via a Serial Peripheral Interface (SPI). Programming via this method is straightforward, but necessitates the user having a hardware programmer such as an STK500 or an AVRISP mkII (see Figure 1-2).

When you first boot the Arduino board, it enters the bootloader, which runs for a few seconds. If it receives a programming command from the IDE over the MCU’s UART (serial interface) in that time period, it loads the program that you are sending it into the rest of the MCU’s program memory. If it does not receive a programming command, it starts running your most recently uploaded sketch, which resides in the rest of the program memory.

When you send an “upload” command from the Arduino IDE, it instructs the USB-to-serial chip (an ATmega 16U2 or 8U2 in the case of the Arduino Uno) to reset the main MCU, thus forcing it into bootloader mode. Then, your computer immediately begins to send the program contents, which the MCU is ready to receive over its UART connection (facilitated by the USB-to-serial converter).

Bootloaders are great because they enable simple programming via USB with no external hardware. However, they do have two downsides:

- They take up valuable program space. If you have written a complicated sketch, the approximately 2 KB of space taken up by the bootloader might be really valuable.
- Using a bootloader means that your program will always be delayed by a few seconds at bootup as the bootloader checks for a programming request.

If you have a programmer (or another Arduino that can be programmed to act as a programmer), you can remove the bootloader from your ATmega and program it directly by connecting your programmer to the ICSP header and using the *File > Upload Using Programmer* command from within the IDE.

Arduino Boards

This book cannot possibly cover all the available Arduino boards; there are many, and manufacturers are constantly releasing new ones with various features. I will focus on a subset of the most commonly used Arduino boards. The following section highlights some of the features in these boards.

The Uno (see Figure 1-3) is the flagship introductory-level Arduino and will be used heavily in this book. It uses an ATmega328P as the main MCU.

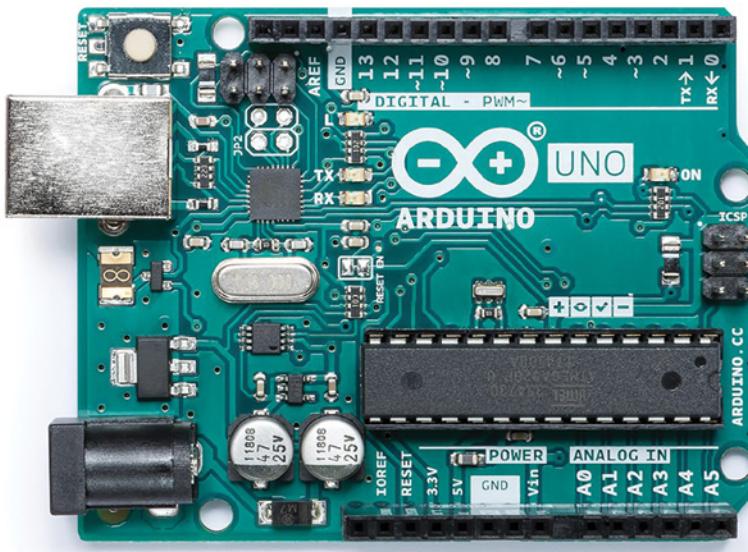


Figure 1-3: The Arduino Uno

Credit: Arduino, arduino.cc

The Mega 2560 (see Figure 1-4) employs an Microchip/Atmel ATmega2560 as the main MCU, which has 54 general I/Os to enable you to interface with many more devices. Think of the Mega as a supercharged version of the Uno—it's faster, has more memory, exposes more ADC channels, and has four hardware serial interfaces (unlike the one serial interface found on the Uno). It costs approximately 50 percent more than the Uno.

The Arduino Leonardo and Arduino Micro (see Figure 1-5 and Figure 1-6) both use the ATmega32U4 as the main microcontroller, which has a USB interface built in. Therefore, they don't need a secondary MCU to perform the serial-to-USB conversion. This cuts down on the cost and enables you to do unique things like emulate a joystick or a keyboard instead of a simple serial device. You will learn how to use these features in Chapter 8, “Emulating USB Devices”. The Micro is functionally identical to the Leonardo, but is a smaller form factor that is designed to be plugged into a solderless or soldered breadboard.

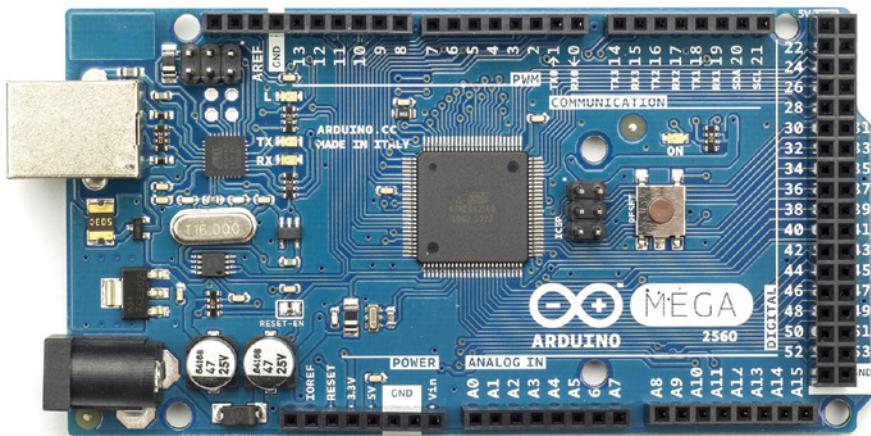


Figure 1-4: The Arduino Mega 2560

Credit: Arduino, arduino.cc



Figure 1-5: The Arduino Leonardo

Credit: Pololu Robotics & Electronics, pololu.com

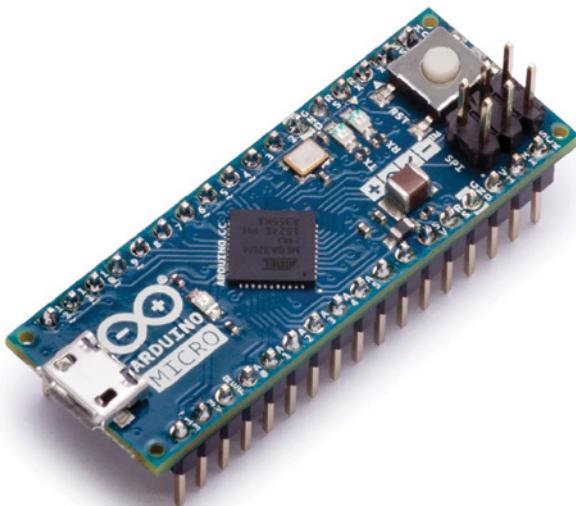


Figure 1-6: The Arduino Micro

Credit: Arduino, arduino.cc

The Due (see Figure 1-7) was Arduino's first foray into using the Arm microarchitecture. It uses a 32-bit Arm Cortex-M3 SAM3X. The Due offers higher-precision ADCs, selectable-resolution pulse-width modulation (PWM), digital-to-analog converters (DACs), a USB host connector, and an 84 MHz clock speed.



Figure 1-7: The Arduino Due

Credit: Pololu Robotics & Electronics, pololu.com

There are a variety of other Arduino boards as well. As you go through the chapters of this book, you may want to consider using some of those boards for more sophisticated projects that you dream up. As your needs get more specific, you may consider using some of the third-party Arduino-compatible boards that are available from companies like SparkFun, Adafruit, Pololu, and others. Because Arduino is an open-source platform, literally hundreds of clones and derivatives are available. The products and companies that I specifically call out in this book are ones that I have tested personally and can confirm work well. Use caution when buying generic Arduino clones online; read the reviews to find out if they work the way they are intended to. When in doubt, buy official Arduino products, or products from well-trusted companies like the ones I've mentioned.

When it comes to things like Bluetooth and Wi-Fi interoperability, the official Arduino offerings are a bit lacking at the time of this writing, so my recommended route is to check out the extremely well-documented Arduino-compatible Feather boards from Adafruit.com. You'll learn how to use these boards for building wireless Bluetooth and Wi-Fi projects in the final chapters of this book. Figure 1-8 shows a Bluetooth-enabled Arduino board from Adafruit.

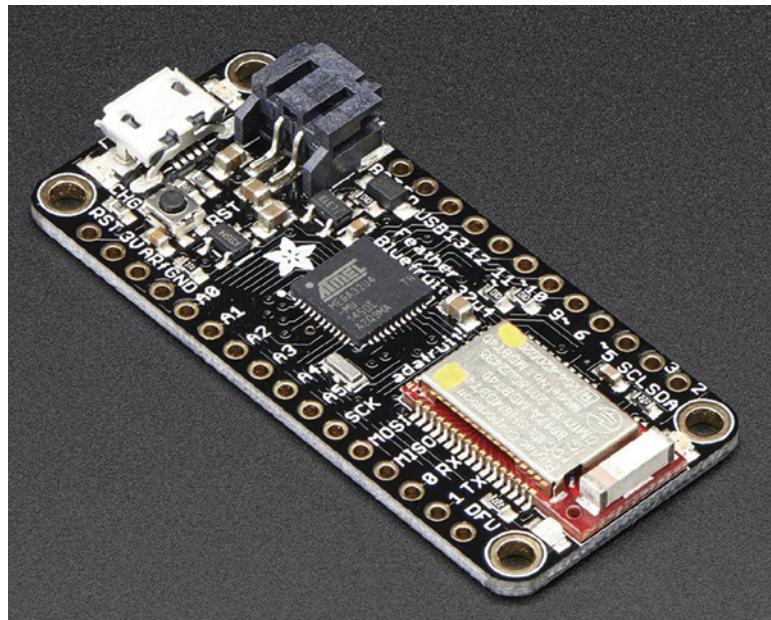


Figure 1-8: The Adafruit Feather 32u4 Bluefruit LE

Credit: Adafruit, adafruit.com

The skills you learn from this book will also easily transfer to a variety of Arduino-inspired platforms that use an Arduino-like programming interface coupled with their own hardware. The Photon (see Figure 1-9) from Particle is a great example of a Wi-Fi enabled microcontroller that uses a programming language inspired by the Arduino language. I use Particle Photons in my apartment to control my reading lamps and window shades from my phone.

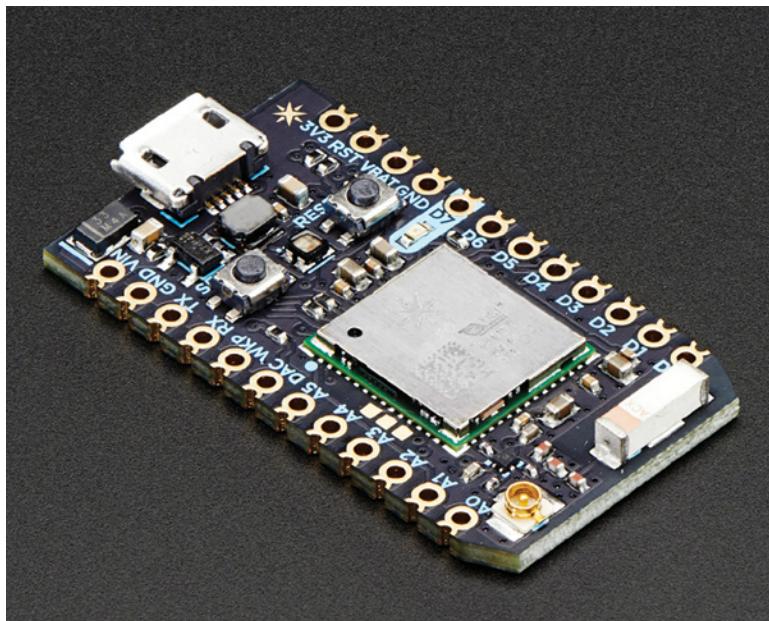


Figure 1-9: The Particle Photon

Credit: Adafruit, adafruit.com

Creating Your First Program

Now that you understand the hardware you'll be using throughout this book, you can install the software or access the Arduino web IDE and run your first program. Throughout this book, you'll generally use the downloaded desktop IDE. Start by downloading the Arduino software to your computer.

THE ARDUINO CLOUD IDE

The Arduino Cloud IDE is not explicitly used in this book's tutorials, but you can use it instead of the desktop IDE if you prefer. Simply set up an account at arduino.cc, and navigate to the editor, at create.arduino.cc/editor. Follow the instructions to install the plug-in and to start uploading code.

Downloading and Installing the Arduino IDE

Go to the Arduino website at arduino.cc and click the Software tab to display the Software page (see Figure 1-10). From there, you can download the newest version of the IDE that corresponds to your operating system.



Figure 1-10: The Arduino.cc page where you can download the Arduino IDE

If you're on Windows, download the installer instead of the Zip file. The installer will handle loading the necessary drivers for you. Run the installer and follow the onscreen directions. All the default options should be fine. For macOS or Linux, download the

compressed folder and extract it. On Mac OS X, simply drag the application into your Applications folder.

Running the IDE and Connecting to the Arduino

Now that you have the IDE downloaded and ready to run, you can connect the Arduino to your computer via USB, as shown in Figure 1-11. Linux and macOS machines usually install the drivers automatically.



Figure 1-11: Arduino Uno connected to a computer via USB

NOTE Having trouble getting the IDE installed, or connecting to your board? Arduino.cc provides great troubleshooting instructions for all operating systems and Arduino hardware. Check out blum.fyi/install-arduino.

Now, launch the Arduino IDE. You’re ready to load your first program onto your Arduino. To ensure that everything is working as expected, you’ll load the Blink example program, which will blink the onboard LED. Most Arduinos have an onboard LED

(connected to pin 13 in the case of the Arduino Uno). Navigate to *File > Examples > Basic*, and click the Blink program. This opens a new IDE window with the Blink program already written for you. First, you'll program the Arduino with this example sketch, and then you'll analyze the program to understand the important components so that you can start to write your own programs in the next chapter.

Before you send the program to your Arduino board, you need to tell the IDE what kind of Arduino you have connected and what port it is connected to. Go to *Tools > Board* and ensure that the right board is selected. This example uses the Uno, but if you are using a different board, select that one (assuming that it also has an onboard LED—most do).

The last step before programming is to tell the IDE what port your board is connected to. Navigate to *Tools > Serial Port* and select the appropriate port. On Windows machines, this will be COM*, where * is some number representing the serial port number.

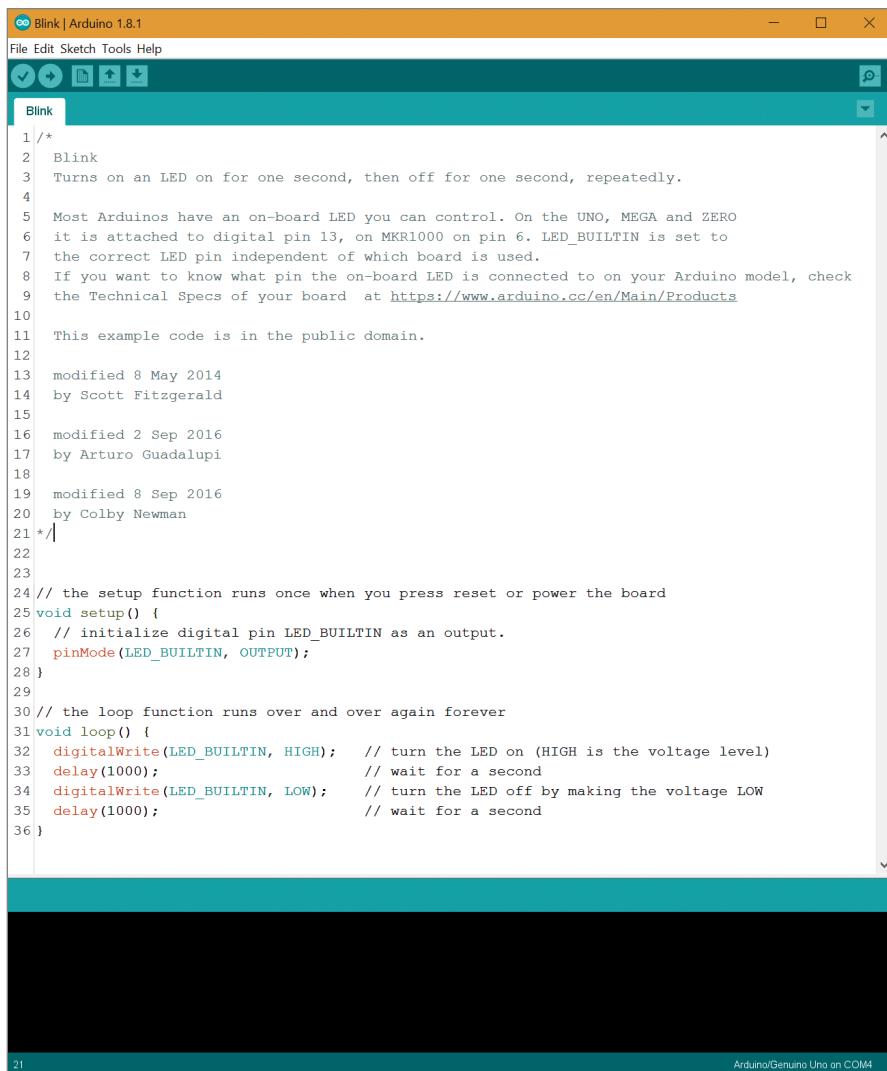
TIP If you have multiple serial devices attached to your computer, try unplugging your board to see which COM port disappears from the menu; then plug it back in and select that COM port.

On Linux and macOS computers, the serial port looks something like /dev/tty.usbmodem* or /dev/tty.usbserial*, where * is a string of alphanumeric characters.

You're finally ready to load your first program. Click the Upload button () in the top-left corner of the IDE. The status bar at the bottom of the IDE shows a progress bar as it compiles and uploads your program. The TX/RX LEDs on your Arduino will flash as it is programming. These LEDs show that data is being transferred to the board from your computer. When the upload completes, the onboard LED on your Arduino should be blinking once per second. Congratulations! You've just uploaded your first Arduino program.

Breaking Down Your First Program

Take a moment to deconstruct the Blink program so that you understand the basic structure of programs written for the Arduino. Consider Figure 1-12.



The screenshot shows the Arduino IDE interface with the title bar "Blink | Arduino 1.8.1". The code editor displays the "Blink" sketch. Line numbers are visible on the left side of the code. The code itself is a standard Blink example, which turns an LED on and off repeatedly. It includes comments explaining the setup and loop functions, and copyright information for various contributors.

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

Figure 1-12: The Blink program (with line numbers)

Here's how the code works, piece by piece:

- 1. Lines 1–21:** This is a multiline comment. Comments are important for documenting your code. Whatever you write between these symbols will not be compiled or even seen by your Arduino. Multiline comments start with `/*` and end with `*/`. Multiline comments are generally used when you have to say a lot (like the description of this program).

2. **Line 24:** This is a single-line comment. When you put `//` on any line, the compiler ignores all text after that symbol on the same line. This is great for annotating specific lines of code or for “commenting out” a particular line of code that you believe might be causing problems.
3. **Line 25:** `void setup()` is one of two functions that must be included in every Arduino program. A *function* is a piece of code that does a specific task. Code within the curly braces of the `setup()` function is executed once at the start of the program. This is useful for one-time settings, such as setting the direction of pins, initializing communication interfaces, and so on. In this program, it will configure the pin that connects to the LED as an output, because you will be telling the pin to do something, instead of querying the pin to determine its state.
4. **Line 27:** The Arduino’s digital pins can all function as inputs or outputs. To configure their direction, use the command `pinMode()`. All pins default to inputs unless you explicitly tell the Arduino to treat them as outputs. Defining a pin as an output during the `setup()` will mean that the pin stays configured as an output for the duration of the program execution (unless you explicitly change it again in the main loop). Set a pin as an output to assign a value to it (5V or 0V in the case of a digital pin on a 5V board like the Uno). Set a pin as an input if you want to “read” the value being applied to it. You’ll explore these concepts more in the next chapter.

The `pinMode()` command takes two arguments. An *argument* gives commands information on how they should operate. Arguments are placed inside the parentheses following a command. The first argument to `pinMode()` determines which pin is having its direction set. For instance, you could simply specify 13 as the first argument, because the onboard LED is connected to pin 13 on the Uno. However, the Arduino language has a number of built-in defined words. These words enable one Arduino program to be abstracted to a variety of different hardware based on what board you’ve told the IDE you are using. The Arduino compiler converts these special words to specific instructions depending on your hardware. For instance, `LED_BUILTIN` is a special word that the compiler converts to the pin number of the built-in LED on your board. On the Uno, this gets converted to “13.” On the MKR1000, this gets converted to “6” because the LED is connected to those pin numbers on those boards. By using this special word instead of just writing the pin number, you ensure that your program is *portable*, meaning it can be executed on various types of Arduino hardware. In the next chapter, you’ll learn about variables, which are special words that you define yourself to assign a meaningful name to numbers, text, and other data.

The second argument to `pinMode()` sets the direction of the pin: INPUT or OUTPUT. These are additional special predefined words that the compiler uses to

configure the MCU onboard your Arduino. Because you want to light an LED, you have set the LED pin to an output (when configured as an output, a pin can “source” or “sink” current by toggling internal switches called transistors).

5. **Line 31:** The second required function in all Arduino programs is `void loop()`. The contents of the `loop` function repeat forever as long as the Arduino is on. If you want your Arduino to do something once at boot only, you still need to include the `loop` function, but you can leave it empty.
6. **Line 32:** `digitalWrite()` is a command that is used to set the state of an output pin. It can set the pin to either 5V or 0V. When an LED is connected to a pin (through a current-limiting resistor), setting it to 5V will enable you to light up the LED. (You will learn more about this in the next chapter.) The first argument to `digitalWrite()` is the pin you want to control. The second argument is the value you want to set it to, either `HIGH` (5V) or `LOW` (0V). The pin remains in this state until it is changed later in the code.
7. **Line 33:** The `delay()` function accepts one argument: a delay time in milliseconds. When calling `delay()`, the Arduino stops doing anything for the amount of time specified. In this case, you are delaying the program for 1000 ms, or 1 second. This results in the LED staying on for 1 second before you execute the next command.
8. **Line 34:** Here, `digitalWrite()` is used to turn the LED off, by setting the pin state to `LOW`.
9. **Line 35:** Again, you delay for 1 second to keep the LED in the off state before the loop repeats and switches to the on state again.

That's all there is to it. Don't be intimidated if you don't fully understand all the code yet. As you put together more examples in the following chapters, you'll become more proficient at understanding program flow, and writing your own code.

Summary

In this chapter, you learned about the following:

- All of the components that comprise an Arduino board
- How the Arduino bootloader allows you to program Arduino firmware over a USB connection
- The differences between the various Arduino boards
- How to connect and install the Arduino with your system
- How to load and run your first program

2

Digital Inputs, Outputs, and Pulse-Width Modulation

What You'll Need for This Chapter:

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

Half-size or full-size breadboard

Assorted jumper wires

Pushbutton

220 Ω resistors ($\times 3$)

10k Ω resistor

5mm red LED

5mm common-anode RGB LED

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch2

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

Blinking an LED is great, as you learned in the preceding chapter, but what makes the Arduino microcontroller platform so useful is that the system is equipped with both inputs and outputs. By combining both, your opportunities are nearly limitless. For example, you can use a magnetic reed switch to play music when your door opens, create an electronic lockbox, or build a light-up musical instrument!

In this chapter, you will start to learn the skills you need to build projects like these. You'll explore the Arduino's digital input capabilities, learn about pull-up and pull-down

resistors, and learn how to control digital outputs. Most Arduinos do not have an analog output, but it is possible to use digital pulse-width modulation (PWM) to emulate it in many scenarios. You will learn about generating pulse-width modulated signals in this chapter. You will also learn how to debounce digital switches, a key skill when reading human input. By the end of the chapter, you will be able to build and program a controllable RGB (Red, Green, Blue) LED nightlight.

NOTE If you want to follow along with a video as I teach you about digital inputs and outputs, debouncing, pulse width modulation, and basic electrical engineering concepts, check out the video content for this chapter: exploringarduino.com/content2/ch2.

Digital Outputs

In Chapter 1, “Getting Started and Understanding the Arduino Landscape,” you learned how to blink an LED. In this chapter, you will further explore Arduino digital output capabilities, including the following topics:

- Setting pins as outputs
- Wiring up external components
- New programming concepts, including for loops, variables, and constants
- Digital versus analog outputs and PWM

Wiring Up an LED and Using Breadboards

In Chapter 1, you learned how to blink the onboard LED, but what fun is that? Now it is time to use the breadboard to wire up an external LED to pin 9 of your Arduino. Adding this external LED will be a stepping-stone towards helping you to understand how to wire up more complex external circuits in the coming chapters. What’s more, pin 9 of the Uno is PWM-enabled (denoted by a ~ on the circuit board next to the header pin), which will allow you to complete the analog output examples later in this chapter.

Working with Breadboards

It is important to understand how breadboards work so that you can use them effectively for the projects in this book. A *breadboard* is a simple prototyping tool that allows you to easily wire up simple circuits without having to solder together parts to a custom-printed circuit board. Consider the blue and red lines that run the length of the board. The pins adjacent to these color-coded lines are designed to be used as power and ground buses. All the red pins are electrically connected, and are generally used for providing power. In the case of most Arduinos and the projects in this book, this will generally be at 5V. All the blue pins are electrically connected and are used for the

ground bus. All the vertically aligned pins are also connected in rows, with a division in the middle to make it easy to mount integrated circuits (ICs) on the breadboard. ICs in the Dual-Inline Package (DIP) form-factor fit neatly across across the center device. Figure 2-1 highlights how the pins are electrically connected, with all the thick lines representing electrically connected holes.

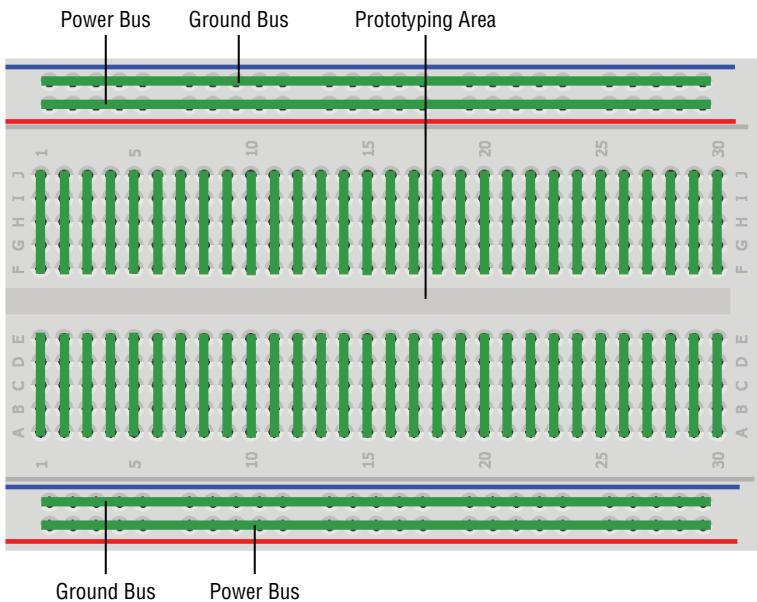


Figure 2-1: Breadboard electrical connections

THE TWO POWER/GROUND BUSES

The red/blue buses on the top of the breadboard are not internally connected to the corresponding buses on the bottom of the breadboard. In other words, if you connect the Arduino's 5V supply to one of the top red pins, it will not be internally connected to the bottom red pins. If you want to use both buses to connect to 5V, you'll need to connect them with a wire. The same is true for the two ground buses. When using a 3.3V Arduino that is interfacing with 5V devices, it's common to use one of the red buses for 5V, and the other for 3.3V.

Wiring LEDs

LEDs will probably be one of the most-used parts in your projects throughout this book. LEDs are polarized; in other words, it matters in what direction you hook them up. The positive lead is called the *anode*, and the negative lead is called the *cathode*.

If you look at the clear top of the LED, there will usually be a flat side on the lip of the casing. That side is the cathode. Another way to determine which side is the anode and which is the cathode is by examining the leads. The shorter lead is the cathode.

LED stands for light-emitting diode. Like all diodes, LEDs allow current to flow in only one direction—from their anode to their cathode. Because current flows from positive to negative, the anode of the LED should be connected to the current source (a 5V digital signal in this case), and the cathode should be connected to ground. LEDs are specified to draw a certain maximum amount of current. The exact amount depends on the LED. Because you'll be driving the LED directly from an Arduino's output pins in this example, it's important to limit the maximum current through the LED. Failure to do so could draw more current from the Arduino's pins than they are specified to supply, or it could result in the LED burning out. Current limiting is easily accomplished by installing a resistor in series with the LED. The resistor can be inserted in series on either side of the LED. Resistors are not polarized, and so you do not have to worry about their orientation.

You'll wire the LED into pin 9 in series with a resistor. LEDs must always be wired in series with a resistor to serve as a current limiter. The larger the resistor value, the more it restricts the flow of current and the dimmer the LED glows. In this scenario, you use a 220Ω resistor. Wire it up as shown in Figure 2-2.

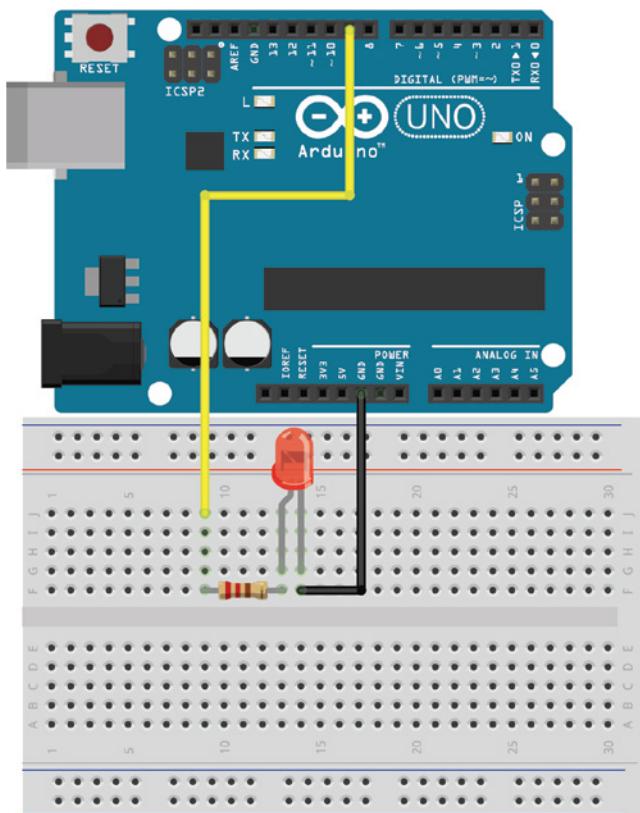


Figure 2-2: Arduino Uno
wired to an LED
Created with Fritzing

OHM'S LAW AND THE POWER EQUATION

The most important equation for any electrical engineer to know is Ohm's law. Ohm's law dictates the relationship between voltage (measured in volts), current (measured in amps), and resistance (measured in ohms or Ω) in a circuit. A circuit is a closed loop with a source of electrical energy (like a 9V battery) and a load (something to use up the energy, like an LED). Before delving into the law, it is important to understand what each term means, at least at a basic level:

- *Voltage* represents the potential electrical difference between two points.
- *Current* flows from a point of higher potential energy to a point of lower potential energy. You can think of current as a flow of water, and voltage as elevation. Water (or current) always flows from a higher elevation (higher voltage) to a lower elevation (ground, or a lower voltage). Current, like water in a river, will always follow the path of least resistance in a circuit.
- *Resistance*, in this analogy, is representative of how easy it is for current to flow. When the water (the current) is flowing through a narrow pipe, less can pass through in the same amount of time as through a larger pipe. The narrow pipe is equivalent to a high resistance value because the water will have a harder time flowing through. The wider pipe is equivalent to a low resistance value (like a wire) because current can flow freely through it.

Ohm's law is defined as follows:

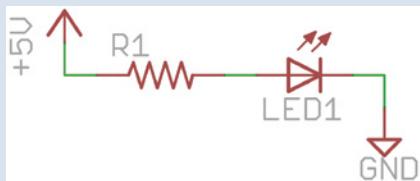
$$V = IR$$

where V is voltage difference in volts, I is current in amps, and R is the resistance in ohms.

In a circuit, all voltage gets used up, and each component offers up some resistance that lowers the voltage. Knowing this, the equation comes in handy for things like figuring out what resistor value to match up with an LED. LEDs have a pre-defined voltage drop across them and are designed to operate at a particular current value. The larger the current through the LED, the brighter the LED glows, up to a limit. For the most common “low-power” LEDs, the maximum current designed to go through an LED is 20 milliamps (a milliamp is 1/1000 of an amp and is typically abbreviated as mA). The voltage drop across an LED is defined in its datasheet. A common value for a red LED is around 2V. Consider the LED circuit shown in Figure 2-3.

(continued)

(continued)

**Figure 2-3:** Simple LED circuit

Created with EAGLE

You can use Ohm's law to decide on a resistor value for this circuit. Assume that this is a standard LED with 20mA forward current and a 2V drop across it. Because the source voltage is 5V and it ends at ground, a total of 5V must drop across this circuit. Because the LED has a 2V drop, the other 3V must drop across the resistor. Knowing that you want approximately 20mA to flow through these components (these components are in series with each other, so the amount of current that flows through the resistor must be the same amount as flows through the LED—there is nowhere else for that current to go), you can find the resistor value by solving for R:

$$R = V / I$$

where $V = 3V$ and $I = 20mA$.

Solving for R, $R = 3V / 0.02A = 150\Omega$. So, with a resistor value of 150Ω , 20mA flows through both the resistor and LED. As you increase the resistance value, less current is allowed to flow through. 220Ω is a bit more than 150Ω , but still allows the LED to glow sufficiently bright, and is a very commonly available resistor value.

Another useful equation to keep in mind is the power equation. The power equation tells you how much power, in watts, is dissipated across a given resistive component. Because increased power is associated with increased heat dissipation, components generally have a maximum power rating. You want to ensure that you do not exceed the maximum power rating for resistors because otherwise they might overheat and fail. A common power rating for through-hole resistors is 1/8 watt (abbreviated as W, milliwatts as mW). The power equation is as follows:

$$P = IV$$

where P is power in watts, and I and V are still defined as the current and voltage, respectively.

For the resistor defined earlier with a voltage drop of 3V and a current of 20mA, $P = 3V \times 0.02A = 60mW$, well under the resistor's rating of 1/8W, or 125mW. So, you do not have to worry about the resistor overheating; it is well within its operating limits.

Programming Digital Outputs

By default, all Arduino pins are set to inputs. If you want to make a pin an output, you need to first tell the Arduino how the pin should be configured. In the Arduino programming language, the program requires two parts: the `setup()` and the `loop()`.

As you learned in Chapter 1, the `setup()` function runs one time at the start of the program, and the `loop()` function runs over and over again. Because you'll generally dedicate each pin to serve as either an input or an output, it is common practice to define all your pins as inputs or outputs in the `setup` function. You start by writing a simple program that sets pin 9 as an output and turns it on when the program starts.

To write this program, use the `pinMode()` command to set the direction of pin 9, and use `digitalWrite()` to make the output high (5V), as shown in Listing 2-1.

Listing 2-1

Turning on an LED—`led.ino`

```
const int LED=9;           // Define LED for pin 9
void setup()
{
    pinMode (LED, OUTPUT); // Set the LED pin as an output
    digitalWrite(LED, HIGH); // Set the LED pin high
}

void loop()
{
    // We are not doing anything in the loop!
}
```

Load this program onto your Arduino, wired as shown in Figure 2-2. Most of the code in this program should look familiar based on the `blink` example that you executed in the first chapter. Notice that at the top of this program, there is a line that defines a variable, `LED`, as equal to 9. Because you set `LED` equal to 9 at the top of the program, `LED` is now interpreted as a variable equivalent to 9 when it is referenced in other parts of the program. Think of it like a placeholder. Anywhere else you see `LED` in the program, the Arduino is replacing that with 9. So, when `pinMode()` and `digitalWrite()` are called, their first argument is pin 9.

Variables can have different types. In this case, the variable is type `int`. `int` is short for “integer,” meaning that the `LED` variable is sized in memory to hold an integer (as opposed to a text string, or a decimal number, for example). In this program, also notice

that I used the `const` operator before defining the pin integer variable. Ordinarily, you'll use variables to hold values that may change during program execution. By putting `const` before your variable declaration, you are telling the compiler that the variable is "read only" and will not change during program execution. When you are defining values that will not change, using the `const` qualifier is recommended because it will prevent you from accidentally altering the value of that variable later in your code. In some of the examples later in this chapter, you will define non-constant variables that may change during program execution.

You must specify the type for any variable that you declare. In the preceding case, it is an integer (pins will always be integers), so you should set it as such. You can now easily modify this sketch to match the one you made in Chapter 1 by moving the `digitalWrite()` command to the loop and adding some delays. Experiment with the delay values and create different blink rates.

Using For Loops

It's frequently necessary to use loops with changing variable values to adjust the parameters of a program. In the case of the program you just wrote, you can implement a `for` loop to see how different blink rates impact your system's operation. You can visualize different blink rates by using a `for` loop to cycle through various rates. The code in Listing 2-2 accomplishes that.

Listing 2-2

LED with changing blink rate—`blink.ino`

```
const int LED=9;           // Define LED for Pin 9
void setup()
{
    pinMode (LED, OUTPUT); // Set the LED pin as an output
}

void loop()
{
    for (int i=100; i<=1000; i=i+100)
    {
        digitalWrite(LED, HIGH);
        delay(i);
        digitalWrite(LED, LOW);
        delay(i);
    }
}
```

Compile the preceding code and load it onto your Arduino. What happens? Take a moment to break down the for loop to understand how it works. The for loop declaration always contains three semicolon-separated entries:

- The first entry sets the index variable for the loop. In this case, the index variable is `i` and is set to start at a value of 100.
- The second entry specifies when the loop should stop. The contents of the loop will execute over and over again while that condition is true. The `<=` operator indicates “less than or equal to.” So, for this loop, the contents will continue to execute as long as the present value of the variable `i` is still less than or equal to 1000.
- The final entry specifies what should happen to the index variable at the end of each loop execution. In this case, `i` will be set to its current value plus 100.

To better understand these concepts, consider what happens in two passes through the for loop:

1. The present value of `i` equals 100.
2. The value of 100 is less than or equal to 1000, so the loop contents execute.
3. The LED is set high, and stays high for 100ms, the present value of `i`.
4. The LED is set low, and stays low for 100ms, the present value of `i`.
5. At the end of the loop, `i` is incremented by 100, so it is now 200.
6. The value of 200 is less than or equal to 1000, so the loop repeats again.
7. The LED is set high, and stays high for 200ms, the present value of `i`.
8. The LED is set low, and stays low for 200ms, the present value of `i`.
9. At the end of the loop, `i` is incremented by 100, so it is now 300.
10. This process repeats until `i` surpasses 1000 and the outer loop function repeats, setting the `i` value back to 100 and starting the process again.

Now that you’ve generated digital outputs from your Arduino, you’ll learn about using PWM to create analog outputs from the I/O pins on your Arduino.

Pulse-Width Modulation with *analogWrite()*

So, you have mastered digital control of your pins. This is great for blinking LEDs, controlling relays, and spinning motors at a constant speed. But what if you want to output a voltage other than 0V or 5V? Well, you can’t—unless you are using a digital-to-analog converter (DAC) integrated circuit, or an Arduino with a built-in DAC (like the Due).

However, you can get pretty close to generating analog output values by using a trick called *pulse-width modulation* (PWM). Select pins on each Arduino can use the `analogWrite()` command to generate PWM signals that can emulate a pure analog signal when used with certain peripherals. These pins are marked with a ~ on the board. On the Arduino Uno, pins 3, 5, 6, 9, 10, and 11 are PWM pins. If you're using an Uno, you can continue to use the circuit from Figure 2-2 to test out the `analogWrite()` command with your LED. Presumably, if you can decrease the voltage being dropped across the resistor, the LED should glow more dimly because less current will flow. That is what you will try to accomplish using PWM via the `analogWrite()` command. The `analogWrite()` command accepts two arguments: the pin to control and the 8-bit value to write to it.

The PWM output is an 8-bit value. In other words, you can write values from 0 to $2^8 - 1$, or 0 to 255. In the case of your LED circuit, setting the output to 255 will result in full brightness, and 0 will result in the LED turning off, with the brightness varying between these two values. Try using a similar for loop structure to the one you used previously to cycle through varying brightness values (see Listing 2-3).

Listing 2-3

LED fade sketch-fade.ino

```
const int LED=9;    // Define LED for Pin 9
void setup()
{
    pinMode (LED, OUTPUT);    // Set the LED pin as an output
}

void loop()
{
    for (int i=0; i<256; i++)
    {
        analogWrite(LED, i);
        delay(10);
    }
    for (int i=255; i>=0; i--)
    {
        analogWrite(LED, i);
        delay(10);
    }
}
```

What does the LED do when you run this code? You should observe the LED fading from off to on, then from on to off. Of course, because this is all in the main loop, this pattern repeats ad infinitum. Be sure to note a few differences in these for loops. In the

first loop, `i++` is just shorthand code to represent `i=i+1`. Similarly, `i--` is functionally equivalent to `i=i-1`. The first for loop fades the LED up, and the second for loop fades it down.

PWM control can be used in a lot of circumstances to emulate pure analog control, but it cannot always be used when you actually need an analog signal. For instance, PWM is great for driving direct current (DC) motors at variable speeds (you'll experiment with this in Chapter 4, "Using Transistors and Driving DC Motors"), but it does not work well for driving speakers unless you supplement it with some external circuitry. Take a moment to examine how PWM actually works. Consider the graphs shown in Figure 2-4.

PWM works by modulating the duty cycle of a square wave (a signal that switches on and off). *Duty cycle* refers to the percentage of time that a square wave is high versus low. You are probably most familiar with square waves that have a duty cycle of 50 percent—they are high half of the time, and low half of the time (this would be accomplished by running `analogWrite(9,127)`).

The `analogWrite()` command sets the duty cycle of a square wave depending on the value you pass to it:

- Writing a value of 0 with `analogWrite()` indicates a square wave with a duty cycle of 0 percent (always low).
- Writing a 255 value indicates a square wave with a duty cycle of 100 percent (always high).
- Writing a 127 value indicates a square wave with a duty cycle of 50 percent (high half of the time, low half of the time).

The graphs in Figure 2-4 show that for a signal with a duty cycle of 25 percent, it is high 25 percent of the time, and low 75 percent of the time. The frequency of this square wave, in the case of the Arduino Uno, is about 490 Hz. In other words, the signal varies between high (5V) and low (0V) about 490 times every second.

FREQUENCY VS. PERIOD

"Period" is often also used to describe an alternating signal, in place of frequency. The "period" of this signal is the time to complete each cycle. The period can easily be computed by dividing 1 second by the frequency. $1/490 \text{ Hz} = .002 \text{ seconds} = 2 \text{ milliseconds per cycle}$.

If you are not actually changing the voltage being delivered to an LED, why do you see it get dimmer as you lower the duty cycle? It is really a result of your eyes playing a trick on you! If the LED is switching on and off every 1 ms (which is the case with a duty cycle of 50 percent), it appears to be operating at approximately half brightness because

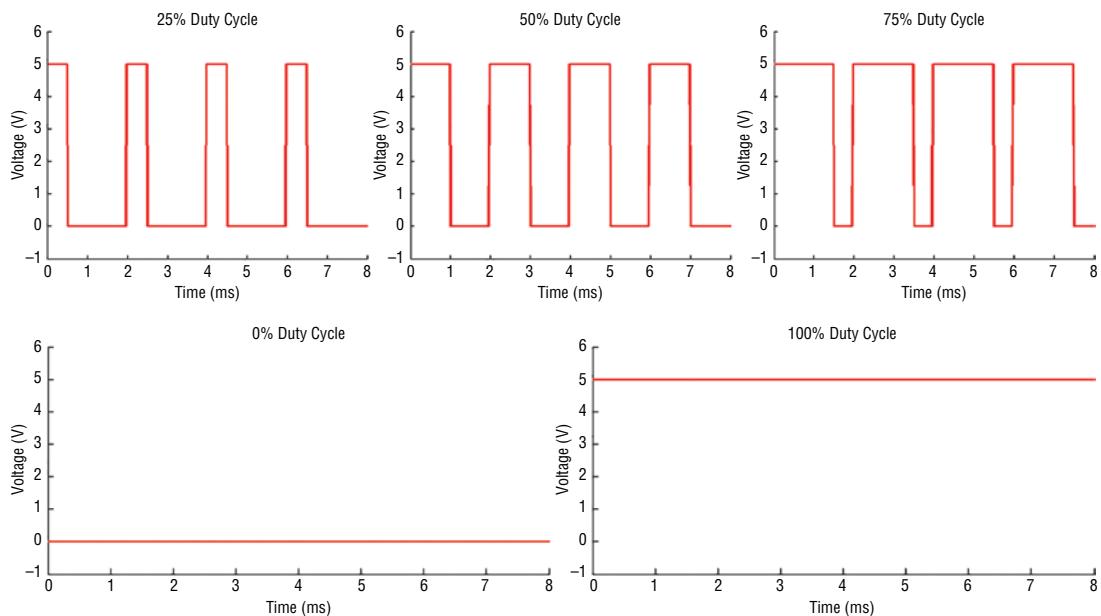


Figure 2-4: PWM signals with varying duty cycles

Created with MATLAB

it is blinking faster than your eyes can perceive. Therefore, your brain actually averages out the signal and tricks you into believing that the LED is operating at half brightness. A similar effect is accomplished with DC motors, which you'll experiment with in Chapter 4. Because motors can't change speed instantaneously, duty cycling their power at 50 percent results in them running at about 50 percent of their maximum speed.

Reading Digital Inputs

Now it is time for the other side of the equation. You've managed to successfully *generate* both digital and analog(ish) outputs. The next step is to *read* digital inputs, such as switches and buttons, so that you can interact with your project in real time. In this section, you learn to read inputs, implement pull-up and pull-down resistors, and debounce a button in software.

Reading Digital Inputs with Pull-Down Resistors

You should start by modifying the circuit that you first built from Figure 2-2. Following Figure 2-5 (schematic of button + pull-down circuit) and Figure 2-6 (breadboard layout of LED and button + pull-down circuit), you'll add a pushbutton and a pull-down resistor connected to a digital input pin.

TIP Be sure to also connect the power and ground buses of the breadboard to the Arduino. Now that you're using multiple devices on the breadboard, that will come in handy.

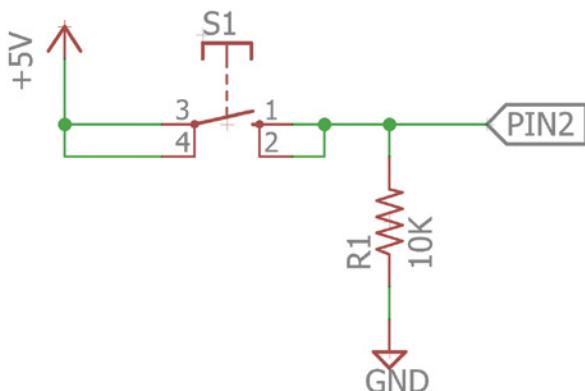


Figure 2-5: Pushbutton input with pull-down resistor schematic

Created with EAGLE

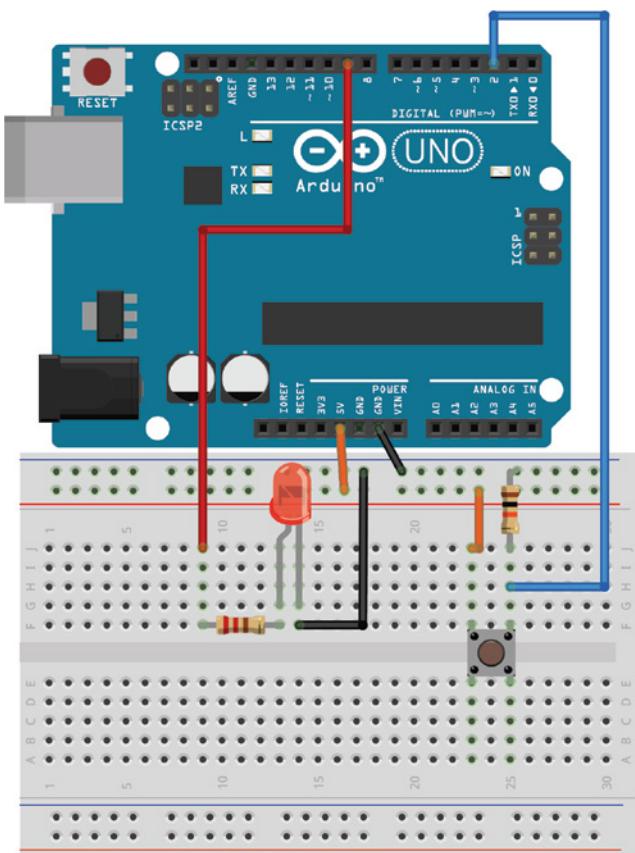


Figure 2-6: Wiring an Arduino to a button and an LED

Created with Fritzing

Before you write the code to read from the pushbutton, it is important to understand the significance of the pull-down resistor used with this circuit. Nearly all digital inputs use a pull-up or pull-down resistor to set the “default state” of the input pin. Imagine the circuit in Figure 2-5 without the $10\text{k}\Omega$ resistor. In this scenario, the pin will obviously read a high value when the button is pressed, because the button directly connects 5V to the input pin when depressed.

But, what happens when the button is not being pressed? In that scenario, the input pin you are reading is essentially connected to nothing—the input pin is said to be “floating.” And, because the pin is not physically connected to 0V or 5V, reading it could cause unexpected results as electrical noise on nearby pins causes its value to fluctuate between high and low. To remedy this, the pull-down resistor is installed as shown in the schematic (Figure 2-5).

Now, consider what happens when the button is not pressed with the pull-down resistor in the circuit: The input pin is connected through a $10\text{k}\Omega$ resistor to ground. While the resistor restricts the flow of current, there is still enough current flow to ensure that the input pin reads a low logic value. It is fairly common to use $10\text{k}\Omega$ as a pull-down resistor value. Larger values are said to be *weak pull-downs* because it is easier to overcome them, and smaller resistor values are said to be *strong pull-downs* because it is easier for more current to flow through them to ground. When the button is pressed, the input pin is directly connected to 5V through the button.

Now, the current has two options:

- It can flow through a nearly zero-resistance path to the 5V rail.
- It can flow through a high-resistance path to the ground rail.

Recall from the sidebar, “Ohm’s Law and the Power Equation,” that the current will always follow the path of least resistance in a circuit. In this scenario, the vast majority of the current flows through the button, and a high logic level is generated on the input pin, because that is the path of least resistance.

NOTE To be a little more pedantic, a tiny amount of “leakage” current will still flow through the 10K resistor when the button is pressed. But, the button path’s resistance is so close to zero when depressed that its effect on the measured voltage at the input pin is negligible. In production designs, especially battery-powered devices like smartphones, every nanoamp of current consumption is precious for conserving battery life. For this reason, those devices will often use the largest pull-down or pull-up resistor that still allows enough current to flow for the default state to be read by the input pin. This ensures that the least amount of power is wasted by the resistor.

NOTE This example uses a pull-down resistor, but you could also use a pull-up resistor by connecting the resistor to 5V instead of ground and by connecting the other side of the button to ground. In this setup, the input pin reads a high-logic value when the button is not pressed and a low-logic value when the button is being pressed.

Pull-down and pull-up resistors are important because they ensure that the button does not create a short circuit between 5V and ground when pressed and that the input pin is never left in a floating state.

Now it is time to write the program for this circuit! In this first example, you just have the LED stay on while the button is pressed, and you have it stay off while the button is not pressed (see Listing 2-4).

Listing 2-4

Simple LED control with a button-led_button.ino

```
const int LED=9;           // The LED is connected to pin 9
const int BUTTON=2;         // The Button is connected to pin 2

void setup()
{
    pinMode (LED, OUTPUT);    // Set the LED pin as an output
    pinMode (BUTTON, INPUT);   // Set button as input (not required)
}

void loop()
{
    if (digitalRead(BUTTON) == LOW)
    {
        digitalWrite(LED, LOW);
    }
    else
    {
        digitalWrite(LED, HIGH);
    }
}
```

Notice here that the code implements some new concepts, including `digitalRead` and `if/else` statements. A new `const int` statement has been added for the button pin. Further, this code defines the button pin as an input in the `setup` function. This is not explicitly necessary, though, because pins are inputs by default; it is shown for completeness. `digitalRead()` reads the value of an input. In this case, it is reading the value of the `BUTTON` pin. If the button is being pressed, `digitalRead()` returns a value of `HIGH`, or `1`. If the button is not being pressed, it returns `LOW`, or `0`. When you place `digitalRead()` in the `if()` statement, you're checking the state of the pin and evaluating if it matches the condition you've declared. In this `if()` statement, you're checking to see if the value returned by `digitalRead()` is `LOW`. The `==` is a comparison operator that tests whether the first item (`digitalRead()`) is equal to the second item (`LOW`). If this is true (that is, the button is not being pressed), then the code inside the brackets executes, and the LED is set to `LOW`. If this is not true (the button is being pressed), then the `else` statement is executed, and the LED is turned `HIGH`.

That's it! Program your circuit with this code and confirm that it works as expected.

Working with “Bouncy” Buttons

When was the last time you had to hold a button down to keep a light on? Probably never. It makes more sense to be able to click the button once to turn it on and to click the button again to turn it off. This way, you do not have to hold the button down to

keep the light on. Unfortunately, this is not quite as easy as you might first guess. You cannot just look for the value of the switch to change from low to high; you need to worry about a phenomenon called *switch bouncing*.

Buttons are mechanical devices that operate as a spring-damper system. In other words, when you push a button down, the signal you read does not just go from low to high; it bounces up and down between those two states for a few milliseconds before it settles. Figure 2-7 illustrates the expected behavior next to the actual behavior you might see when probing the button using an oscilloscope (though this figure was generated using a MATLAB script):

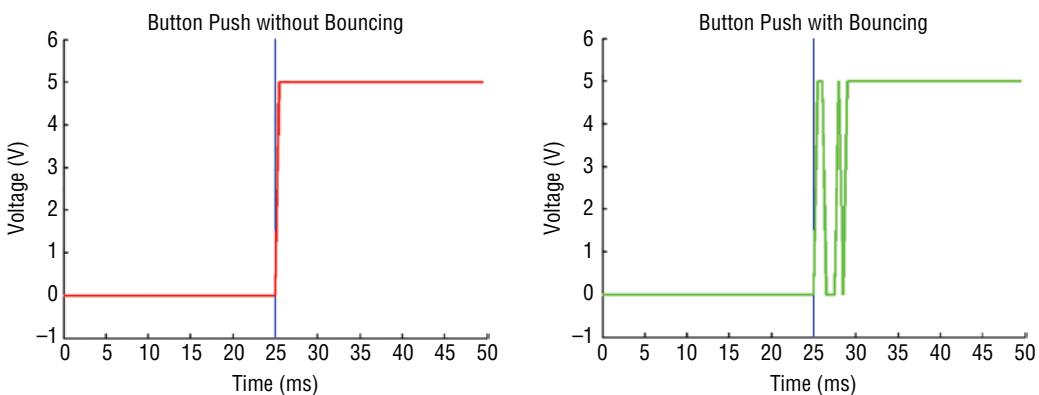


Figure 2-7: Bouncing button effects

Created with MATLAB

The button is physically pressed at the 25 ms mark. You would expect the button state to be immediately read as a high logic level, as the graph on the left shows. However, the button actually bounces up and down before settling, as the graph on the right shows.

If you know that the switch is going to do this, it is relatively straightforward to deal with it in software. Switch-debouncing software can look for a button state change, wait for the bouncing to finish, and then read the switch state again. This program logic can be expressed as follows:

1. Store a previous button state and a current button state (initialized to LOW).
2. Read the current button state.
3. If the current button state differs from the previous button state, wait 5 ms because the button must have changed state.
4. After 5 ms, reread the button state and use that as the current button state.
5. If the previous button state was low, and the current button state is high, toggle the LED state.
6. Set the previous button state to the current button state.
7. Return to step 2.

This is a perfect opportunity to explore using *functions* for the first time. *Functions* are blocks of code that can accept input arguments, execute code based on those arguments, and optionally return a result. Without realizing it, you've already been using predefined functions throughout your programs. For example, `digitalWrite()` is a function that accepts a pin and a state, and writes that state to the given pin. To simplify your program, you can define your own functions to encapsulate actions that you do over and over again.

Within the program flow (listed in the preceding steps) is a series of repeating steps that need to be applied to changing variable values. Because you'll want to repeatedly debounce the switch value, it's useful to define the steps for debouncing as a function that can be called each time. This function accepts the previous button state as an input and outputs the current debounced button state. The following program accomplishes the preceding steps and switches the LED state every time the button is pressed. You'll use the same circuit as the previous example for this. Try loading it onto your Arduino and see how it works (see Listing 2-5).

Listing 2-5

Debounced button toggling—debounce.ino

```
const int LED=9;           // The LED is connected to pin 9
const int BUTTON=2;         // The Button is connected to pin 2
boolean lastButton = LOW;   // Variable containing the previous
                           // button state
boolean currentButton = LOW; // Variable containing the current
                           // button state
boolean ledOn = false;      // The present state of the LED (on/off)

void setup()
{
    pinMode (LED, OUTPUT);    // Set the LED pin as an output
    pinMode (BUTTON, INPUT);  // Set button as input (not required)
}

/*
 * Debouncing Function
 * Pass it the previous button state,
 * and get back the current debounced button state.
 */
boolean debounce(boolean last)
{
    boolean current = digitalRead(BUTTON);        // Read the button state
    if (last != current)                          // if it's different...
    {
        delay(5);                                //Wait 5ms
```

```

        current = digitalRead(BUTTON);           //Read it again
    }
    return current;                         //Return the current value
}

void loop()
{
    currentButton = debounce(lastButton);      //Read debounced state
    if (lastButton == LOW && currentButton == HIGH) //if it was pressed...
    {
        ledOn = !ledOn;                      //Toggle the LED value
    }
    lastButton = currentButton;                //Reset button value
    digitalWrite(LED, ledOn);                 //Change the LED state
}

```

Now, break down the code in Listing 2-5. First, constant values are defined for the pins connected to the button and LED. Next, three Boolean *variables* are declared. When the `const` qualifier is not placed before a variable declaration, you are indicating that this variable can change within the program. By defining these values at the top of the program, you are declaring them as *global* variables that can be used and changed by any function within this sketch. The three Boolean variables declared at the top of this sketch are *initialized* as well, meaning that they have been set to an initial value (`LOW`, `LOW`, and `false` respectively). Later in the program, the values of these variables can be changed with an assignment operator (a single equals sign: `=`). Boolean variables can only have two states, true or false. In the Arduino language (and most programming languages), `true`, `HIGH`, and `1` are all equivalent; `false`, `LOW`, and `0` are also equivalent to each other.

Consider the function definition in the preceding code: `boolean debounce (boolean last)`. This function accepts a Boolean input variable called `last` and returns a Boolean value representing the current debounced pin value. This function compares the current button state with the previous (`last`) button state that was passed to it as an argument. The `!=` represents inequality and is used to compare the present and previous button values in the `if` statement. If they differ, then the button must have been pressed and the `if` statement will execute its contents. The `if` statement waits 5 ms before checking the button state again. This 5 ms gives sufficient time for the button to stop bouncing. The button is then checked again to ascertain its stable value. As you learned earlier, functions can optionally return values. In the case of this function, the `return current` statement returns the value of the current Boolean variable when the function is called. `current` is a *local* variable—it is declared and used only within the `debounce` function. When the `debounce` function is called from the main loop, the returned value is written to

the *global* `currentButton` variable that was defined at the top of the sketch. Because the function was defined as `debounce`, you can call the function by writing `currentButton = debounce(lastButton)` from within the `setup` or `loop` functions. `currentButton` will be set equal to the value that is returned by the `debounce` function.

After you've called the function and populated the `currentButton` variable, you can easily compare it to the previous button state by using the `if` statement in the code. The `&&` is a logical operator that means "AND." By joining two or more equality statements with an `&&` in an `if` statement, you are indicating that the contents of the `if` statement block should execute only if both of the equalities evaluate to true. If the button was previously `LOW` and is now `HIGH`, you can assume that the button has been pressed, and you can reassign the value of the `ledOn` variable. By putting an `!` in front of the `ledOn` variable, you reset the variable to the opposite of whatever it currently is. The loop is finished off by updating the previous button variable and writing the updated LED state.

This code should change the LED state each time the button is pressed. If you try to accomplish the same thing without debouncing the button, you will find the results unpredictable, with the LED sometimes working as expected and sometimes not.

Building a Controllable RGB LED Nightlight

In this chapter, you have learned how to control digital outputs, how to read debounced buttons, and how to use PWM to change LED brightness. Using those skills, you can now hook up an RGB LED and a debounced button to cycle through some colors for a controllable RGB LED nightlight. It's possible to mix colors with an RGB LED by changing the brightness of each color independently.

In this scenario, you use a common anode LED. That means that the LED has four leads. One of them is an anode pin that is shared among all three diodes, while the other three pins connect to the cathodes of each diode color. Wire that LED up to three PWM pins through current-limiting resistors on the Arduino, as shown in the wiring diagram in Figure 2-8. As with the single red LED, values of 220Ω will work well for current limiting.

Because this LED is a common anode, that means the cathode of each diode is being controlled by the Arduino, instead of the anode as in the example with the red LED. When an Arduino's pin is set as an output, it is really doing one of two things:

- When you set it `HIGH`, it "sources" current. Current is allowed to flow from the Arduino's 5V supply, out of the pin, and then through the attached load to ground.
- When you set it `LOW`, it "sinks" current. Current is permitted to flow into the pin, to the internal ground.

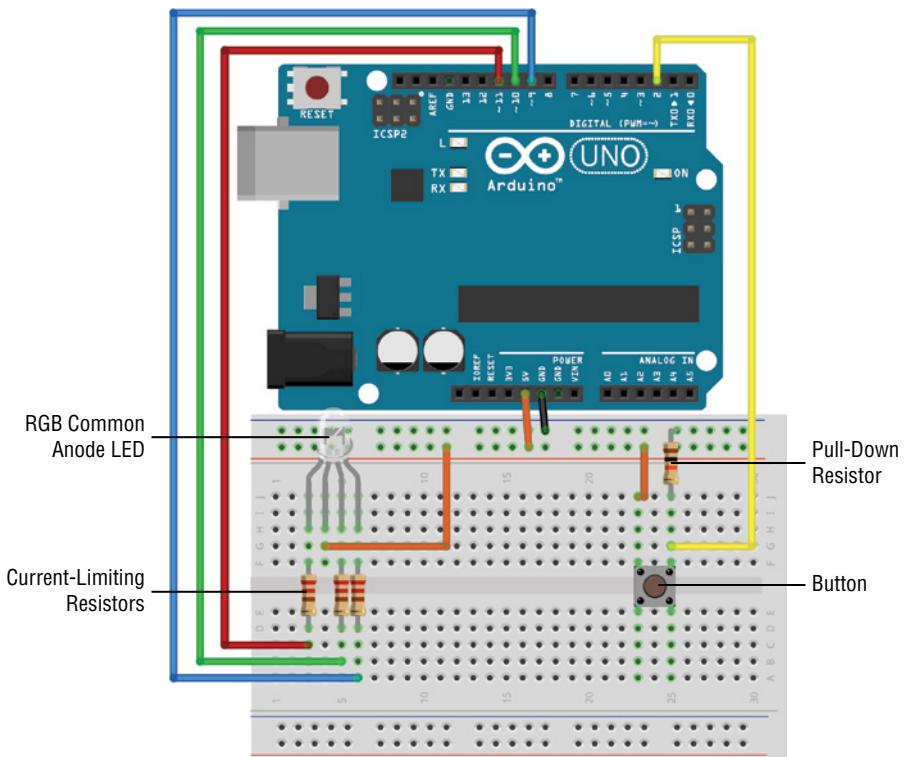


Figure 2-8: Nightlight wiring diagram

Created with Fritzing

Therefore, if an LED's anode is connected to 5V, and its cathode is connected to an Arduino pin configured as an output, its logic will be inverted. When you set the pin LOW, that will enable current to flow from 5V, through the resistor and LED, and into the Arduino's current sink. When you set the pin HIGH, it will be at the same (5V) potential as the anode of the LED, so no current will flow and the LED will turn off.

You can configure a debounced button to cycle through a selection of colors each time you press it. To do this, it is useful to add an additional function to set the RGB LED to the next state in the color cycle. In the following program (see Listing 2-6), I have defined a total of seven color states, plus one off state for the LED. Using the `analogWrite()` function, you can choose your own color-mixing combinations. The only change to the `loop()` from the previous example is that instead of flipping a single LED state, an LED state counter is incremented each time the button is pressed, and it is reset back to zero when you cycle through all the options. Upload this code to your Arduino connected to the circuit you just built and enjoy your nightlight. Modify the color states by changing the values of `analogWrite()` to make your own color options.

Listing 2-6

Toggling LED nightlight-rgb_nightlight.ino

```
const int BLED=9;      // Blue LED Cathode on Pin 9
const int GLED=10;     // Green LED Cathode on Pin 10
const int RLED=11;     // Red LED Cathode on Pin 11
const int BUTTON=2;    // The Button is connected to pin 2

boolean lastButton = LOW;    // Last Button State
boolean currentButton = LOW; // Current Button State
int ledMode = 0;             // Cycle between LED states

void setup()
{
  pinMode (BLED, OUTPUT);   // Set Blue LED as Output
  pinMode (GLED, OUTPUT);   // Set Green LED as Output
  pinMode (RLED, OUTPUT);   // Set Red LED as Output
  pinMode (BUTTON, INPUT);  // Set button as input (not required)
}

/*
 * Debouncing Function
 * Pass it the previous button state,
 * and get back the current debounced button state.
 */
boolean debounce(boolean last)
{
  boolean current = digitalRead(BUTTON);           // Read the button state
  if (last != current)                            // If it's different...
  {
    delay(5);                                     // Wait 5ms
    current = digitalRead(BUTTON);                // Read it again
  }
  return current;                                 // Return the current value
}

/*
 * LED Mode Selection
 * Pass a number for the LED state and set it accordingly
 * Note, since the RGB LED is COMMON ANODE, you must set the
 * cathode pin for each color LOW for that color to turn ON.
 */
void setMode(int mode)
{
  //RED
  if (mode == 1)
```

```
{  
    digitalWrite(RLED, LOW);  
    digitalWrite(GLED, HIGH);  
    digitalWrite(BLED, HIGH);  
}  
//GREEN  
else if (mode == 2)  
{  
    digitalWrite(RLED, HIGH);  
    digitalWrite(GLED, LOW);  
    digitalWrite(BLED, HIGH);  
}  
//BLUE  
else if (mode == 3)  
{  
    digitalWrite(RLED, HIGH);  
    digitalWrite(GLED, HIGH);  
    digitalWrite(BLED, LOW);  
}  
//PURPLE (RED+BLUE)  
else if (mode == 4)  
{  
    analogWrite(RLED, 127);  
    analogWrite(GLED, 255);  
    analogWrite(BLED, 127);  
}  
//TEAL (BLUE+GREEN)  
else if (mode == 5)  
{  
    analogWrite(RLED, 255);  
    analogWrite(GLED, 127);  
    analogWrite(BLED, 127);  
}  
//ORANGE (GREEN+RED)  
else if (mode == 6)  
{  
    analogWrite(RLED, 127);  
    analogWrite(GLED, 127);  
    analogWrite(BLED, 255);  
}  
//WHITE (GREEN+RED+BLUE)  
else if (mode == 7)  
{  
    analogWrite(RLED, 170);  
    analogWrite(GLED, 170);  
    analogWrite(BLED, 170);  
}  
//OFF (mode = 0)
```

```
else
{
    digitalWrite(RLED, LOW);
    digitalWrite(GLED, LOW);
    digitalWrite(BLED, LOW);
}

void loop()
{
    currentButton = debounce(lastButton);           // Read debounced state
    if (lastButton == LOW && currentButton == HIGH) // If it was pressed...
    {
        ledMode++;                                // Increment the LED value
    }
    lastButton = currentButton;                   // Reset button value
    // If you've cycled through the different options,
    // reset the counter to 0
    if (ledMode == 8) ledMode = 0;
    setMode(ledMode);                            // Change the LED state
```

This might look like a lot of code, but it is nothing more than a conglomeration of code snippets that you have already written throughout this chapter.

How else could you modify this code? You could add additional buttons to independently control one of the three colors. You could also add blink modes, using code from Chapter 1 that blinked the LED. The possibilities are limitless.

Summary

In this chapter, you learned about the following:

- How a breadboard works
- How to pick a resistor to current-limit an LED
- How to wire an external LED to your Arduino
- How to use PWM to control LED brightness
- How to read a pushbutton
- How to debounce a pushbutton
- How to use for loops
- How to utilize pull-up and pull-down resistors

3

Interfacing with Analog Sensors

What You'll Need for This Chapter

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

Half-size or full-size breadboard

Assorted jumper wires

220 Ω resistors ($\times 3$)

10k Ω resistors ($\times 2$)

10k Ω trim potentiometer

Photoresistor

An analog sensor (any of the following)

 TMP36 analog temperature sensor

 Sharp GP2Y0A21YK0F IR distance sensor with JST cable

 ADXL335, ADXL377, or ADXL326 triple-axis accelerometer

5 mm white LED

5 mm common-anode RGB LED

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch3

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

The world around you is analog. Even though you might hear that the world is “going digital,” most observable features in your environment will always be analog in nature. The world can assume an infinite number of potential states, whether you are considering the color of sunlight, the temperature of the ocean, or the concentration of contaminants in the air. This chapter focuses on developing techniques for discretizing these infinite possibilities into palatable digital values that can be analyzed with a microcontroller system like the Arduino.

In this chapter, you will learn about the differences between analog and digital signals and how to convert between the two, as well as a handful of the analog sensors that you can interface with your Arduino. Using skills that you acquired in the preceding chapter, you will add light sensors for automatically adjusting your nightlight. You will also learn how to send analog data from your Arduino to your computer via a USB-to-serial connection, which opens enormous possibilities for developing more complex systems that can transmit environmental data to your computer.

NOTE On the content web page for this chapter, you’ll find a video about how to read analog inputs, as well as an in-depth video that describes the differences between analog and digital signals: exploringarduino.com/content2/ch3.

Understanding Analog and Digital Signals

If you want your devices to interface with the world, they will inevitably be interfacing with analog data. Consider the projects you completed in the preceding chapter. You used a switch to control an LED. A switch is a digital input—it has only two possible states: on or off, high or low, 1 or 0, and so on. Digital information (what your computer or the Arduino processes) is a series of binary (or digital) data. Each bit has only one of two values.

The world around you, however, rarely expresses information in only two ways. Look out the window. What do you see? If it’s daytime, you probably see sunlight, trees moving in the breeze, and maybe cars passing or people walking around. All these things that you perceive cannot readily be classified as binary data. Sunlight is not on or off; its brightness varies over the course of a day. Similarly, wind does not just have two states; it gusts at different speeds and directions all the time.

Comparing Analog and Digital Signals

The graphs in Figure 3-1 show how analog and digital signals compare to each other. On the left is a square wave that varies between only two values: 0 and 5 volts. Just like with the button that you used in the preceding chapter, this signal is only a “logic

high” or “logic low” value. On the right is part of a cosine wave. Although its bounds are still 0 and 5 volts, the signal takes on an infinite number of values between those two voltages.

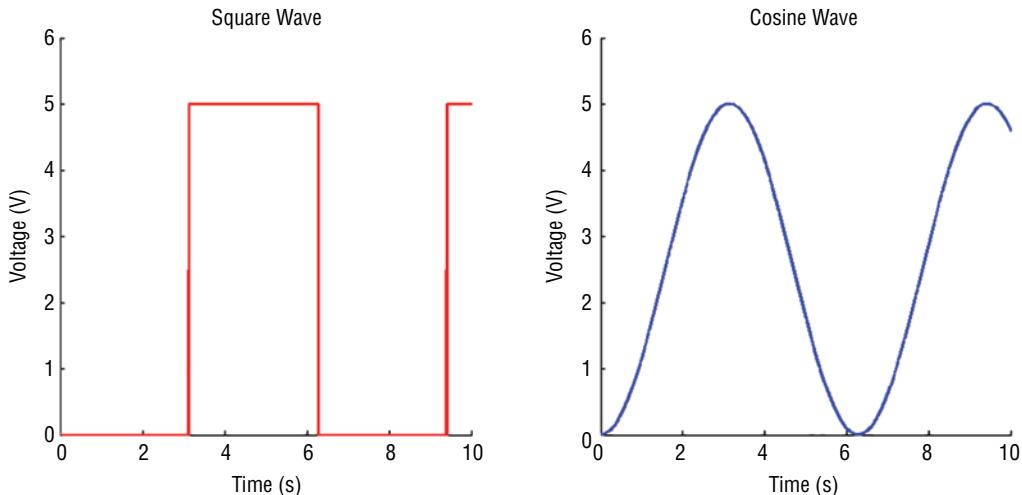


Figure 3-1: Analog and digital signals

Created with MATLAB

Analog signals are those that cannot be discretely classified; they vary within a range, theoretically taking on an infinite number of possible values within that range. Think about sunlight as an example of an analog input you may want to measure. Naturally, there is a reasonable range over which you might measure sunlight. Often measured in lux, or luminous flux per unit area, you can reasonably expect to measure values between 0 lux (for pitch black) and 130,000 lux in direct sunlight. If your measuring device were infinitely accurate, you could measure an infinite number of values between those two. An indoor setting might be 400 lux. If it were slightly brighter, it could be 401 lux, then 401.1 lux, then 401.11 lux, and so on.

A computer system could never feasibly measure an infinite number of decimal places for an analog value because memory and computational processing power must take on finite values. If that’s the case, how can you interface your Arduino with the “real world”? The answer is analog-to-digital converters, which can convert analog values into digital representations with a finite amount of precision and speed.

Converting an Analog Signal to Digital

Suppose that you want to measure the brightness of your room. Presumably, a good light sensor can produce a varying output voltage that changes with the brightness of the room. When it is pitch black, the device outputs 0V, and when it’s completely

saturated by light, it outputs 5V, with values in between corresponding to the varying amount of light. That's all well and good, but how do you go about reading those values with an Arduino to figure out how bright the room is? You can use the Arduino's analog-to-digital converter (ADC) pins to convert analog voltage values into number representations that you can work with.

The accuracy of an ADC is determined by its resolution. In the case of the Arduino Uno, there is a 10-bit ADC for doing your analog conversions. The designation *10-bit* means that the ADC can subdivide (or quantize) an analog signal into 2^{10} different values. If you do the math, you'll find that $2^{10} = 1024$; thus, the Arduino can assign a value from 0 to 1023 for any analog value that you give it. Although it is possible to change the reference voltage, you'll be using the default 5V reference for the analog work that you do in this book. The reference voltage determines the maximum voltage that you are expecting, and therefore the value that will be mapped to 1023. So, with a 5V reference voltage, putting 0V on an ADC pin returns a value of 0, 2.5V returns a value of 512 (half of 1023), and 5V returns a value of 1023. To better understand what's happening here, consider what a simpler, 3-bit ADC would do, as shown in Figure 3-2.

NOTE If you want to learn more about using your own reference voltage or using a different internal voltage reference, check out the *analogReference()* page on the Arduino website, at blum.fyi/arduino-analog-reference.

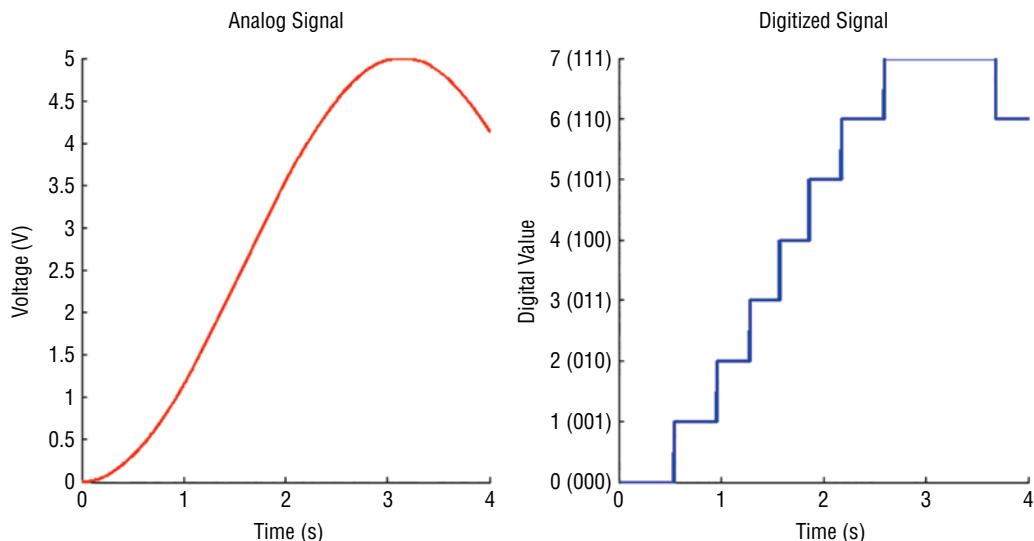


Figure 3-2: Three-bit analog quantization

Created with MATLAB

A 3-bit ADC has three bits of resolution. Because $2^3 = 8$, there are a total of eight logic levels, from 0 to 7. Therefore, any analog value that is passed to a 3-bit ADC will have to be assigned a value from 0 to 7. Looking at Figure 3-2, you can see that voltage levels are converted to discrete digital values that can be used by the microcontroller. The higher the resolution, the more steps that are available for representing each value. In the case of the Arduino Uno, there are 1024 steps rather than the 8 shown here.

The Arduino Due and Arduino Zero have 12-bit ADCs (0–4095), so they can quantize analog data with greater accuracy than the Uno. You can also buy external ADC chips with higher resolutions that communicate with the Arduino via an interface like I2C or SPI. (You'll learn about these communication buses in later chapters.)

Reading Analog Sensors with the Arduino: *analogRead()*

Now that you understand how to convert analog signals to digital values, you can integrate that knowledge into your programs and circuits. Different Arduinos have different numbers of analog input pins, but you read them all the same way, using the `analogRead()` command. First, you'll experiment with a potentiometer and a packaged analog sensor. Then, you'll learn how voltage dividers work, and how you can use them to make your own analog sensors from devices that vary their resistance in response to some kind of input.

Reading a Potentiometer

The easiest analog sensor to read is a simple potentiometer (a pot, for short). Odds are that you have tons of these around your home in your stereos, speakers, thermostats, cars, and elsewhere. Potentiometers are variable voltage dividers (discussed later in this chapter) that look like knobs. They come in a lot of sizes and shapes, but they all have three pins. You connect one of the outer pins to ground, and the other to the 5V pin from your Arduino. Potentiometers are symmetrical, so it doesn't matter which side you connect the 5V rail and ground to. You connect the middle pin to analog input 0 on your Arduino. Figure 3-3 shows how to properly hook up your potentiometer to an Arduino.

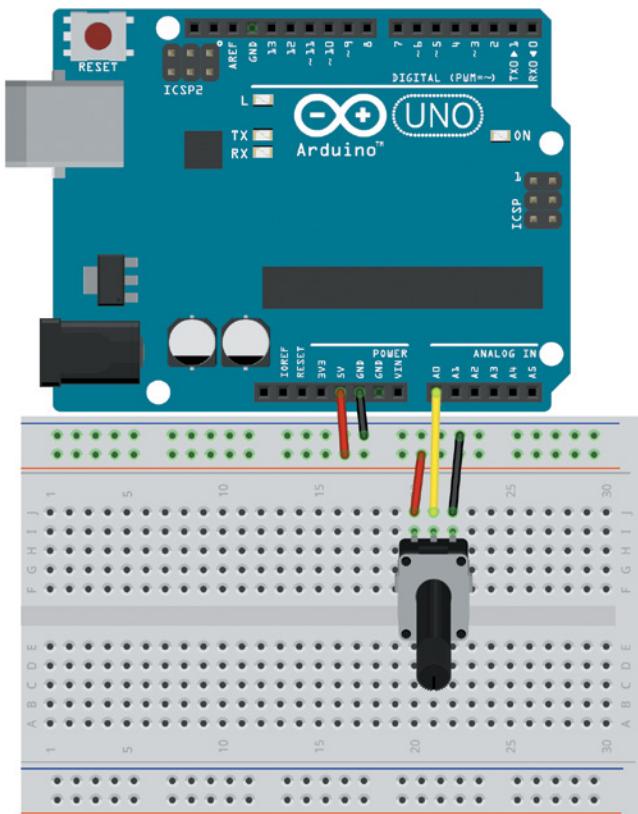


Figure 3-3: Potentiometer circuit

Created with Fritzing

As you turn the potentiometer, you're varying the voltage that you are feeding into analog input 0 between 0V and 5V. If you want, you can confirm this with a multimeter in voltage measurement mode by hooking it up as shown in Figure 3-4 and reading the display as you turn the potentiometer's knob. The red (positive) probe should be connected to the middle pin, and the black (negative) probe should be connected to whichever side is connected to ground. Note that your potentiometer and multimeter might look different than the ones shown here.

Before you use the potentiometer to control another piece of hardware, use the Arduino's serial communication functionality to print out the potentiometer's ADC value on your computer as it changes. Use the `analogRead()` function to read the value of the analog pin connected to the Arduino, and the `Serial.println()` function to print it to the Arduino IDE serial monitor. Start by writing and uploading the program in Listing 3-1 to your Arduino.



Figure 3-4: Multimeter measurement

Listing 3-1

Potentiometer reading sketch-pot.ino

```
// Potentiometer Reading Program

const int POT=0; // Pot on Analog Pin 0
int val = 0; // Variable to hold the analog reading from the POT

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

void loop()
{
    val = analogRead(POT);
    Serial.println(val);
    delay(500);
}
```

You'll investigate the functionality of the serial interface more in later chapters. For now, just be aware that the serial interface to the computer must be started in the `setup()` function. `Serial.begin()` takes one argument that specifies the communication speed, or baud rate. The *baud rate* specifies the number of bits being transferred per second. Faster baud rates enable you to transmit more data in less time, but can also introduce transmission errors in some communication systems. A common value is 9600 baud, which is what you will use throughout most of this book.

In each iteration through the loop, the `val` variable is set to the present value that the ADC reports from analog pin 0. The `analogRead()` command requires the number of the ADC pin to be passed to it. In this case, it's 0 because that's what you hooked the potentiometer up to. You can also pass `A0`, though the `analogRead()` function knows you must be passing it an analog pin number, so you can pass `0` as shorthand. After the value has been read (a number between 0 and 1023), `Serial.println()` prints that value over serial to the computer's serial terminal, followed by a "newline" that advances the cursor to the next line. The loop then delays for 500 milliseconds (so that the numbers don't scroll by faster than you can read them), and the process repeats.

After loading this program onto your Arduino, you'll notice that the TX LED on your Arduino is blinking every 500 ms (at least it should be). This LED indicates that your Arduino is transmitting data via the USB connection to the serial terminal on your computer. You can use a variety of terminal programs to see what your Arduino is sending, but the Arduino IDE conveniently has one built right in! Click the circled button shown in Figure 3-5 to launch the serial monitor.

After launching the serial monitor, you should see a window with numbers streaming by. Turn the dial; you'll see the numbers go up and down to correspond with the position of the potentiometer. If you turn it all the way in one direction, the numbers should approach 0, and if you turn it all the way in the other direction, the numbers should approach 1023. The Monitor output will look like the example shown in Figure 3-6.

NOTE If you're getting funky characters, make sure that you have the baud rate set correctly. Because you set it to 9600 in the code, you need to set it to 9600 in this window (using the drop-down menu on the bottom-right corner of the serial monitor window) as well.

You've now managed to successfully turn a dial and make some numbers change. Pretty exciting, right? No? Well, this is the just the first step. Next, you'll learn about other types of analog sensors and how you can use the data from analog sensors to control other pieces of hardware. For now, you use the familiar LED, but in later chapters, you will use motors and other output devices to display your analog inputs.

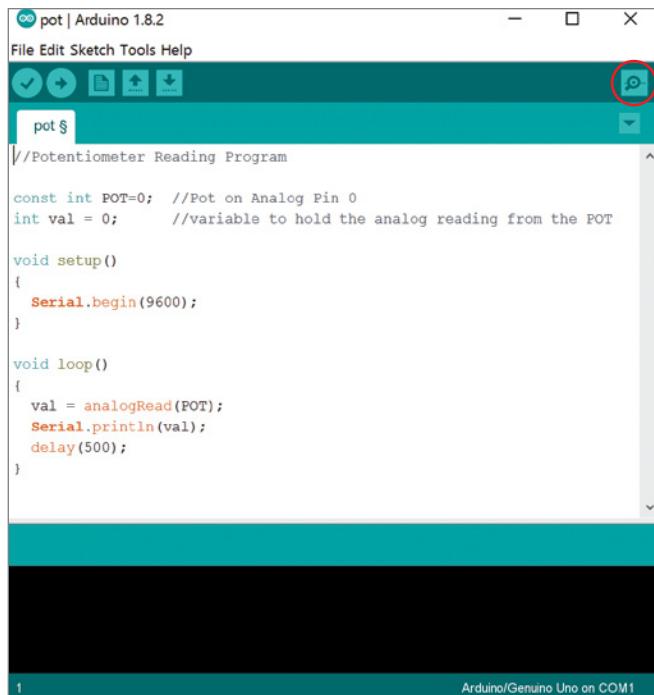


Figure 3-5: Click the serial monitor button.

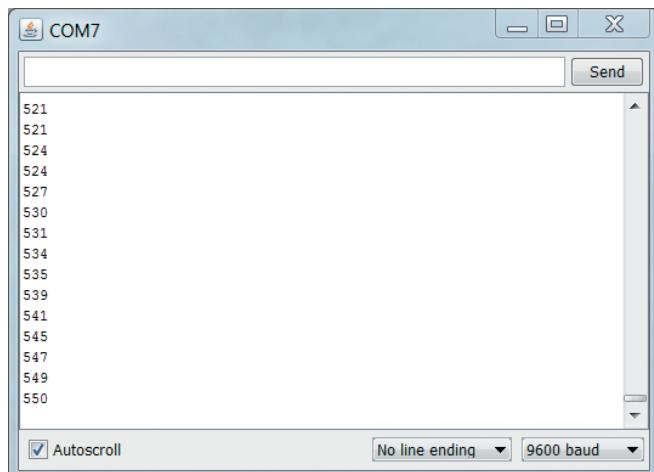


Figure 3-6: Incoming serial data

Using Analog Sensors

Although potentiometers generate an analog voltage value on a pin, they aren't really sensors in the traditional meaning. They "sense" your turning of the dial, but that gets boring pretty quickly. The good news is that all kinds of sensors generate analog output values corresponding to "real-world" actions. Examples include the following:

- Accelerometers that detect tilting (All modern smartphones have these.)
- Magnetometers that detect magnetic fields (A magnetometer in your phone is what enables your map app to tell what direction you are pointing in.)
- Infrared sensors that detect distance to an object
- Temperature sensors that can tell you about the operating environment of your project

Many of these sensors are designed to operate in a manner similar to the potentiometer you just experimented with: you provide them with a power (VCC) and ground (GND) connection, and they output an analog voltage between VCC and GND on the third pin that you hook up to your Arduino's ADC.

For this next experiment, you get to choose what kind of analog sensor you want to use. They all output a value between 0V and 5V when connected to an Arduino, so they will all work the same for your purposes. Here are some examples of sensors that you can use:

- **Sharp infrared proximity sensor with cable or carrier board** (exploringarduino.com/parts/IR-Distance-Sensor)

The Sharp infrared distance sensors are popular for measuring the distance between your project and other objects. As you move farther from the object you are aiming at, the voltage output decreases. In the datasheet from the part web page linked here, Figure 2 on page 5 shows the relationship between voltage and measured distance to a reflective object. There are multiple variants of this product, each with a different sensing range.

- **TMP36 temperature sensor** (exploringarduino.com/parts/TMP36)

The TMP36 temperature sensor easily correlates temperature readings in Celsius with voltage output levels. The voltage output from the TMP36 is 0V at -50°C and 1.75V at 125°C —it varies linearly between those values. Your Arduino can compute the temperature using the following formula derived from this linear relationship: $\text{Temperature (in }^{\circ}\text{C)} = (100 \times \text{voltage}) - 50$. The graph in Figure 3-7 (extracted from the datasheet) shows this conversion.

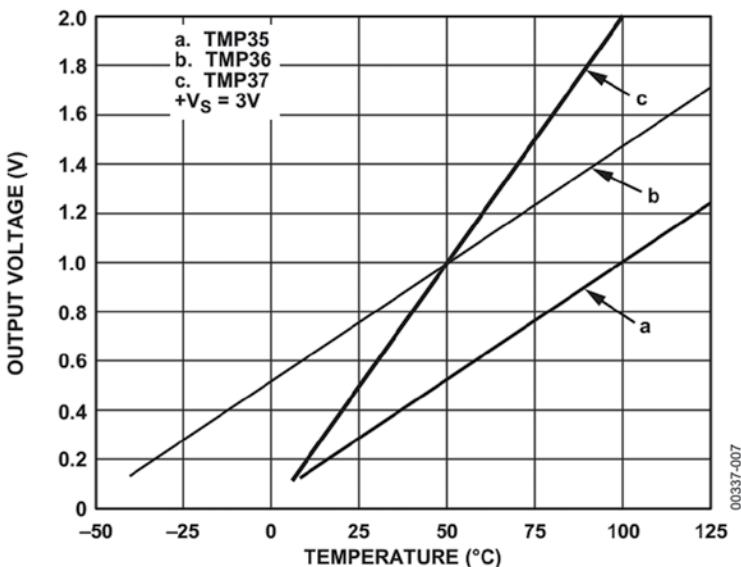


Figure 3-7: TMP36 output voltage to temperature correlation

Credit: Copyright © 2019, Analog Devices, Inc. All Rights Reserved.

- **ADXL335, ADXL377, ADXL326 triple-axis analog accelerometers** (exploringarduino.com/parts/TriAxis-Analog-Accelerometer)

Triple-axis accelerometers are great for detecting orientation. Analog accelerometers output an analog value corresponding to each axis of movement: X, Y, and Z (each on a different pin). Using some clever math (trigonometry and knowledge of gravity), you can use these voltage values to ascertain the position of your project in 3D space! Importantly, many of these sensors are 3.3V, so you will need to use the `analogReference()` command paired with the AREF pin to set a 3.3V voltage reference to enable you to get the full resolution out of the sensor.

Now that you've chosen a sensor, it's time to put that sensor to use. This simple example uses the TMP36 temperature sensor mentioned in the previous section. However, feel free to use any analog sensor you can get your hands on. Experiment with one of the examples listed earlier, or find your own. (It should be 5V compliant if you are using the Arduino Uno.) The following steps are basically the same for any analog sensor you might want to use.

To begin, wire up your common-anode RGB LED as you did in the preceding chapter, and wire the temperature sensor output up to analog input 0 as shown in the Figure 3-8. Be sure to connect the sensor's power and ground pins to 5V and GND respectively.

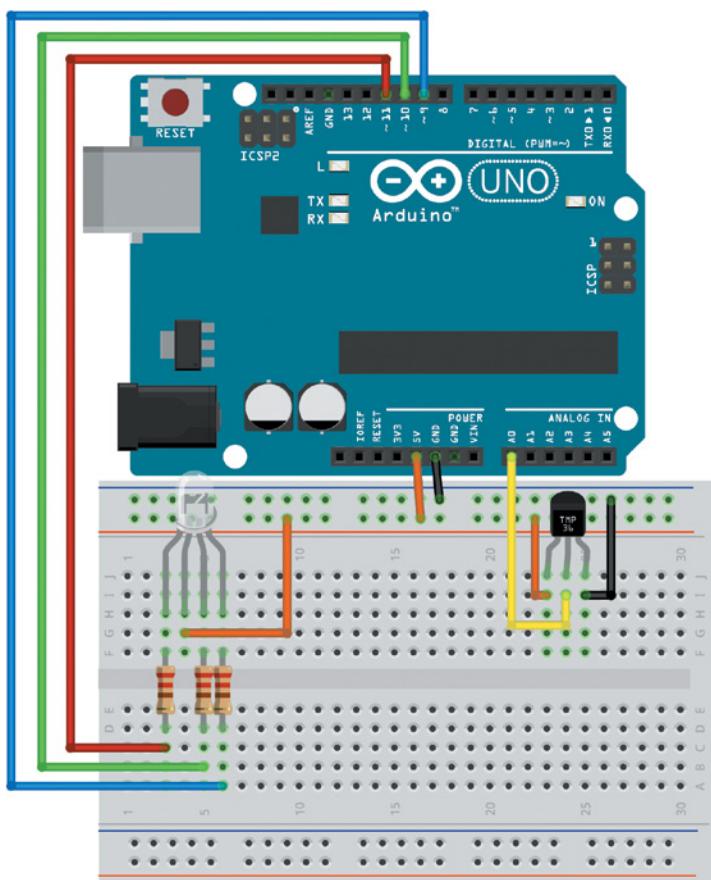


Figure 3-8: Temperature sensor circuit

Created with Fritzing

Using this circuit, you'll make a simple temperature alert system. The light will glow green when the temperature is within an acceptable range, will turn red when it gets too hot, and will turn blue when it gets too cold.

First things first: you need to ascertain what values you want to use as your cutoffs. Using the exact same sketch as in Listing 3-1 (“Potentiometer reading sketch”), use the serial monitor to figure out what analog values correspond to the temperature cutoffs you care about. My room is about 20°C, which corresponds to an analog reading of about 143. These numbers might differ for you, so launch the sketch from Listing 3-1, open the serial terminal, and take a look at the readings you are getting. You can confirm the values mathematically using the graph from Figure 3-7. In my case, a value of 143/1023 corresponds to a voltage input of about 700 mV. Deriving your own values

from the datasheet, you can use the following equation to convert between the temperature ($^{\circ}\text{C}$) and the voltage (mV):

$$\text{Temperature} \left(^{\circ}\text{C} \right) \times 10 = \text{voltage} \left(\text{mV} \right) - 500$$

Plugging in the value of 700 mV, you can confirm that it equates to a temperature of 20°C . Using this same logic (or by simply observing the serial window and picking a value), you can determine that 22°C translates to a digital value of 147 and 18°C translates to a digital value of 139. Those values will serve as the cutoffs that will change the color of the LED to indicate that it is too hot or too cold. Using the if statements, digitalWrite function, and analogRead function that you have now learned about, you can easily read the temperature, determine what range it falls in, and set the LED accordingly. Remember, because this is a common-anode LED, the control polarity is reversed. Setting the various red, green, and blue pins LOW turns that diode on, and setting them HIGH turns the diode off.

NOTE Before you copy the code in Listing 3-2, try to write it yourself and see whether you can make it work. After giving it a try, compare it with the code here. How did you do?

Listing 3-2

Temperature alert sketch-tempalert.ino

```
// Temperature Alert!
const int BLED=9;           // Blue LED Cathode on Pin 9
const int GLED=10;           // Green LED Cathode on Pin 10
const int RLED=11;           // Red LED Cathode on Pin 11
const int TEMP=A0;           // Temp Sensor is on pin A0

const int LOWER_BOUND=139;    // Lower Threshold
const int UPPER_BOUND=147;   // Upper Threshold

int val = 0;                // Variable to hold analog reading

void setup()
{
  pinMode (BLED, OUTPUT); // Set Blue LED as Output
  pinMode (GLED, OUTPUT); // Set Green LED as Output
  pinMode (RLED, OUTPUT); // Set Red LED as Output
}
```

```
void loop()
{
    val = analogRead(TEMP);

    // LED is Blue
    if (val < LOWER_BOUND)
    {
        digitalWrite(RLED, HIGH);
        digitalWrite(GLED, HIGH);
        digitalWrite(BLED, LOW);
    }
    // LED is Red
    else if (val > UPPER_BOUND)
    {
        digitalWrite(RLED, LOW);
        digitalWrite(GLED, HIGH);
        digitalWrite(BLED, HIGH);
    }
    // LED is Green
    else
    {
        digitalWrite(RLED, HIGH);
        digitalWrite(GLED, LOW);
        digitalWrite(BLED, HIGH);
    }
}
```

This code listing doesn't introduce any new concepts; rather, it combines what you have learned so far to make a system that uses both inputs and outputs to interact with the environment. To try it out, squeeze the temperature sensor with your fingers or exhale on it to heat it up. Blow on it to cool it down.

Using Variable Resistors to Make Your Own Analog Sensors

Thanks to physics, tons of devices change resistance as a result of physical action. For example, some conductive inks change resistance when squished or flexed (force sensors and flex sensors), some semiconductors change resistance when struck by light (photoresistors), and some polymers change resistance when heated or cooled (thermistors). These are just a few examples of components that you can take advantage of to build your own analog sensors. Because these sensors are changing resistance and not voltage, you need to create a voltage divider circuit so that you can measure their resistance change.

Using Resistive Voltage Dividers

A resistive voltage divider uses two resistors to output a voltage that is some fraction of the input voltage. The output voltage is a function directly related to the value of the two resistors. So, if one of the resistors is a variable resistor, you can monitor the change in voltage from the voltage divider that results from the varying resistance. The size of the other resistor can be used to set the sensitivity of the circuit, or you can use a potentiometer to make the sensitivity adjustable.

First, consider a fixed voltage divider and the equations associated with it, as shown in Figure 3-9. A0 in Figure 3-9 refers to analog pin 0 on the Arduino.

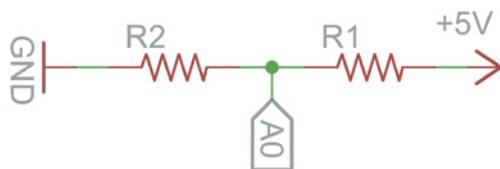


Figure 3-9: Simple voltage divider circuit

Created with EAGLE

The equation for a voltage divider is as follows:

$$V_{\text{out}} = V_{\text{in}} \left(\frac{R_2}{R_1 + R_2} \right)$$

In this case, the voltage input is 5V, and the voltage output is what you'll be feeding into one of the analog pins of the Arduino. In the case where R1 and R2 are matched (both $10k\Omega$, for example), the 5V is divided by 2 to make 2.5V at the analog input. Confirm this by plugging values into the equation:

$$V_{\text{out}} = 5V \left(\frac{10k\Omega}{10k\Omega + 10k\Omega} \right) = 5V \times 0.5 = 2.5V$$

Now, suppose one of those resistors is replaced with a variable resistor, such as a photoresistor. Photoresistors (see Figure 3-10) change resistance depending on the amount of light that hits them. In this case, I'll opt to use a $200k\Omega$ photoresistor. When in complete darkness, its resistance is about $200k\Omega$; when saturated with light, the resistance drops to around $5k\Omega$. Whether you choose to replace R1 or R2 and what value you choose to make the fixed resistor will affect the scale and precision of the readings you receive.

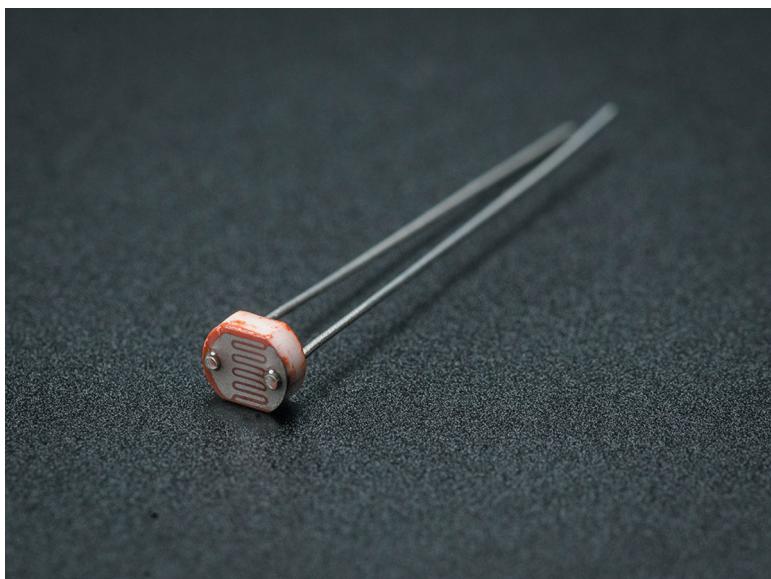


Figure 3-10: Photoresistor

Credit: Adafruit, adafruit.com

Try experimenting with different configurations and using the serial monitor to see how your values change. As an example, I will choose to replace R1 with the photoresistor, and I'll make R2 a $10\text{k}\Omega$ resistor (see Figure 3-11). Replace your common-anode RGB LED with a bright-white LED. White light is generally more practical for use as a nightlight. Connect the LED's anode to pin 9 (a PWM-capable pin). Note, this now means you're back to controlling the LED's anode, so an `analogWrite()` value of 255 will turn the LED to full brightness, and 0 will turn it off.

NOTE CdS (cadmium sulfide) photoresistors, like the one you'll use for this project, are not RoHS compliant. RoHS is an international standard aimed at either reducing or eliminating the use of hazardous substances in electronics manufacturing. RoHS regulations are aimed at manufacturers, not individuals like the readers of this book. There is no reason to be concerned about the quantities of cadmium in the photocell you are using. However, factories that make large quantities of products with substances like cadmium can have a negative impact on the environment as the parts they make eventually become e-waste and find their way into landfills. If you're planning to use your Arduino skills to eventually manufacture a mass-market project, and you need light sensing, consider using an ambient light sensor (ALS) IC or a phototransistor.

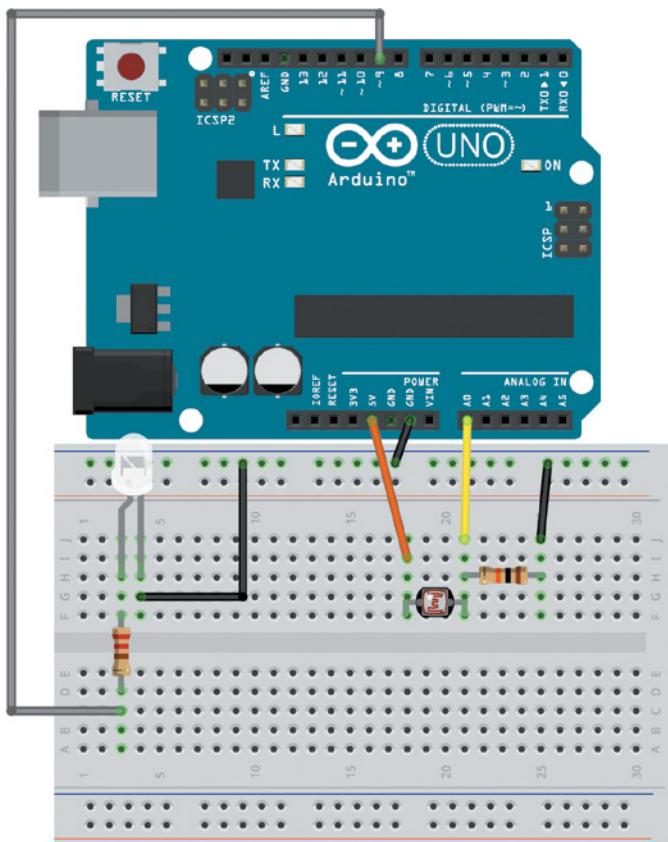


Figure 3-11: Photoresistor circuit

Created with Fritzing

Load up your trusty serial printing sketch again and try changing the lighting conditions over the photoresistor. Hold it up to a light and cup it with your hands. You aren't going to be hitting the full range from 0 to 1023 because the variable resistor will never have a resistance of zero. Rather, you can probably figure out the maximum and minimum values that you are likely to receive. You can use the data from your photoresistor to make a more intelligent nightlight. The nightlight should get brighter as the room gets darker, and vice versa. Using your serial monitor sketch, pick the values that represent when your room is at full brightness or complete darkness. In my case, I found that a dark room has a value of around 200 and a completely bright room has a value around 900. These values will vary for you based upon your lighting conditions, the resistor value you are using, and the value of your photoresistor.

Using Analog Inputs to Control Analog Outputs

Recall that you can use the `analogWrite()` command to set the brightness of an LED. However, it is an 8-bit value; that is, it accepts values between 0 and 255 only, whereas the ADC is returning values as high as 1023. Conveniently, the Arduino programming language has two functions that are useful for mapping between two sets of values: the `map()` and `constrain()` functions. The `map()` function looks like this:

```
output = map(value, fromLow, fromHigh, toLow, toHigh)
```

`value` is the information you are starting with. In your case, that's the most recent reading from the analog input. `fromLow` and `fromHigh` are the input boundaries. These are values you found to correspond to the minimum and maximum brightness in your room. In my case, they were 200 and 900. `toLow` and `toHigh` are the values you want to map the brightness values to. Because `analogWrite()` expects a value between 0 and 255, you use those values. However, you want a darker room to map to a brighter LED. Therefore, when the input from the ADC is a low value, you want the output to the LED's PWM pin to be a high value, and vice versa.

Conveniently, the `map` function can handle this automatically; simply swap the high and low values so that the low value is 255 and the high value is 0. The `map()` function creates a linear mapping. For example, if your `fromLow` and `fromHigh` values are 200 and 900, respectively, and your `toLow` and `toHigh` values are 255 and 0, respectively, 550 maps to 127 because 550 is halfway between 200 and 900 and 127 is halfway between 255 and 0. Importantly, however, the `map()` function does not constrain these values. So, if the photoresistor does measure a value below 200, it is mapped to a value above 255 (because you are inverting the mapping). Obviously, you don't want that because you can't pass a value greater than 255 to the `analogWrite()` function. You can deal with this by using the `constrain()` function. The `constrain()` function looks like this:

```
output = constrain(value, min, max)
```

If you pass the output from the `map` function into the `constrain` function, you can set the `min` to 0 and the `max` to 255, ensuring that any numbers above or below those values are constrained to either 0 or 255. Finally, you can then use those values to command your LED! Now, take a look at what that final sketch will look like (see Listing 3-3).

NOTE If your white LED is very bright, then make sure it is pointed away from your photocell. You want your photocell to be picking up ambient room brightness, not the light created by your LED when it turns on.

Listing 3-3

Automatic nightlight sketch-nightlight.ino

```
// Automatic Night Light

const int WLED=9;          // White LED Anode on pin 9 (PWM)
const int LIGHT=0;          // Light Sensor on Analog Pin 0
const int MIN_LIGHT=200;    // Minimum Expected light value
const int MAX_LIGHT=900;    // Maximum Expected Light value
int val = 0;                // Variable to hold the analog reading

void setup()
{
    pinMode(WLED, OUTPUT); // Set White LED pin as output
}

void loop()
{
    val = analogRead(LIGHT);           // Read the light sensor
    val = map(val, MIN_LIGHT, MAX_LIGHT, 255, 0); // Map the light reading
    val = constrain(val, 0, 255);       // Constrain light value
    analogWrite(WLED, val);           // Control the White LED
}
```

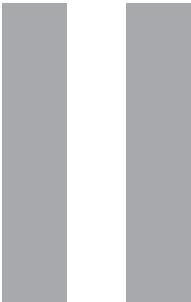
Note that this code reuses the `val` variable. You can alternatively use a different variable for each function call. In functions such as `map()`, where `val` is both the input and the output, the previous value of `val` is used as the input, and its value is reset to the updated value when the function has completed.

Play around with your nightlight. Does it work as expected? Remember, you can adjust the sensitivity by changing the minimum and maximum bounds of the mapping function or changing the fixed resistor value. Use the serial monitor to observe the differences with different settings until you find one that works the best. Can you combine this sketch with the color-selection nightlight that you designed in the preceding chapter? Try adding a button to switch between colors, and use the photoresistor to adjust the brightness of each color.

Summary

In this chapter, you learned about the following:

- The differences between analog and digital signals
- How to convert analog signals to digital signals
- How to read an analog signal from a potentiometer
- How to display data using the serial monitor
- How to interface with packaged analog sensors
- How to create your own analog sensors
- How to map and constrain analog readings to drive analog outputs



Interfacing with Your Environment

Chapter 4: Using Transistors and Driving DC Motors

Chapter 5: Driving Stepper and Servo Motors

Chapter 6: Making Sounds and Music

Chapter 7: USB Serial Communication

Chapter 8: Emulating USB Devices

Chapter 9: Shift Registers

4

Using Transistors and Driving DC Motors

Parts You'll Need for This Chapter:

- Arduino Uno or Adafruit METRO 328
- USB cable (Type A to B for Uno, Type A to Micro-B for METRO)
- Half-size breadboard
- Assorted jumper wires
- 1k Ω resistor
- 10k Ω resistors ($\times 2$)
- 10k Ω trim potentiometer
- Photoresistors ($\times 2$)
- 9V battery
- 9V battery clip
- L7805CV 5V voltage regulator
- 10 μ F 50V electrolytic capacitors ($\times 2$)
- 0.1 μ F ceramic capacitor
- 1N4001 diode
- PN2222 NPN bipolar junction transistor (BJT)
- Roving robot chassis kit with wheels and DC motors
- 9V DC motor
- TI L293D dual H-bridge motor driver

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at: exploringarduino.com/content2/ch4

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

You're now a master of *observing* information from the world around you. But how can you *control* that world? Blinking LEDs and automatically adjusting night-lights are a good start, but you can do so much more. Using assorted types of motors and actuators, and with the help of transistors, you can use your Arduino to generate physical action in the real world. By pairing motors with your Arduino, you can drive robots, build mechanical arms, add an additional degree of freedom to distance sensors, and much more.

In this chapter, you will learn how to control inductive loads like direct current (DC) motors, how to use transistors to switch high-current devices, and how to add integrated circuits (ICs) to your projects. At the end of this chapter, you will build a light-controlled car that you can control using only a flashlight!

NOTE If you want to learn all about motors and transistors, you can watch a video on this topic on this chapter's content web page: exploringarduino.com/content2/ch4.

WARNING In this chapter, you use a 9V battery so that you can run motors that require more power than the Arduino can provide. These voltages are still not high enough to pose a danger to you, but if hooked up improperly, these batteries can damage your electronics. As you make your way through the exercises in this chapter, follow the diagrams and instructions carefully. Avoid short circuits (connecting power directly to ground), and when you are sharing the ground line between power supplies, don't try to connect two separate voltage sources to each other. For example, don't try to hook both the 9V supply and the Arduino's 5V supply into the same supply row on the breadboard. Doing so could damage the 5V regulator on your Arduino, damaged the Arduino's microcontroller.

Driving DC Motors

DC motors, which you can find in numerous devices around your home, rotate continuously when a DC voltage is applied across them. These motors are commonly used

as the driving motors in radio control (RC) cars, in power drills and saws, and as the motors that make the discs spin in DVD players. DC motors are great because they come in a huge array of sizes and are generally very cheap. By adjusting the voltage you apply to them, you can change their rotation speed. Using a gearbox, you can trade their speed for torque. By reversing the direction of the voltage applied to them, you can change their direction of rotation as well. This is generally done using an H-bridge, which you will learn about later in this chapter.

Brushed DC motors, such as the ones you are using for this chapter, employ stationary magnets (the *stator*) and a spinning coil (the *rotor*). Electricity is transferred to the coil using “brushes,” hence the name *brushed* DC motors. Unlike *brushless* DC motors (such as the stepper motors that you’ll explore in the next chapter), brushed DC motors are cheap and offer easier speed control. However, brushed DC motors do not last as long because the brushes can wear out over time.

Some larger devices that rely on brushed motors, like corded power tools, have replaceable carbon brushes. Brushed DC motors work through an inductive force. When current passes through the spinning coil, it generates a magnetic field that is either attracted to or repelled by the stationary magnets, depending on the polarity. By using the brushes to swap the polarity each half-rotation, you can generate angular momentum.

The exact same configuration can be used to create a generator if you manually turn the armature. This generates a fluctuating magnetic field that, in turn, generates current. This is how hydroelectric generators work: falling water turns the shaft, and a current is produced. This capability to create current is why you will use a diode later in this chapter to ensure that the motor cannot send current back into your circuit when it is forcibly turned.

Handling High-Current Inductive Loads

DC motors are available in a variety of voltages and power ratings. First, you’ll experiment with 9V DC motors. Later in this chapter, you’ll use 5V geared motors. DC motors generally require more current than the Arduino’s built-in power supply can provide, and they can create harmful voltage spikes due to their inductive nature. To address this issue, you will first learn how to effectively isolate a DC motor from your Arduino, and then how to power it using a secondary supply. A transistor will allow the Arduino to switch the motor on and off safely, as well as to control the speed using the pulse-width modulation (PWM) techniques that you learned about in Chapter 3, “Interfacing with Analog Sensors.” Refer to the schematic shown in Figure 4-1 as you learn about the various components that go into connecting a DC motor to an Arduino with a secondary power supply. Make sure you understand all of these concepts before you actually start wiring.

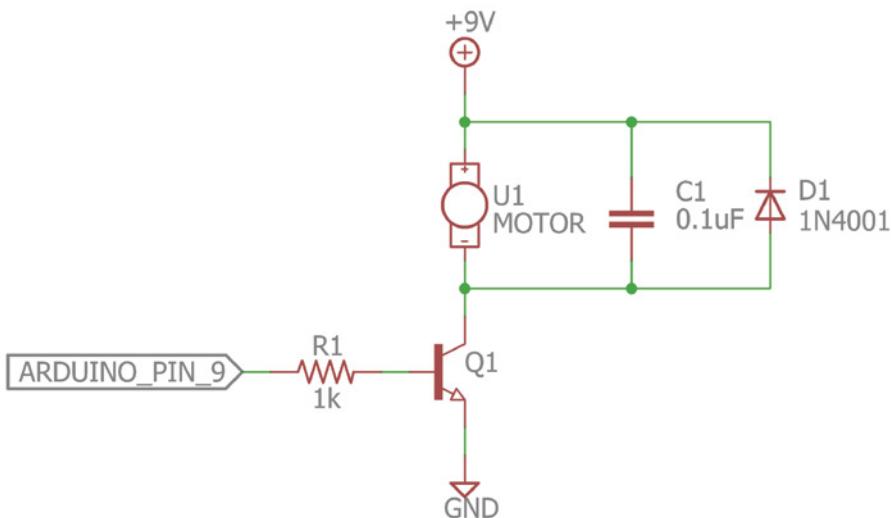


Figure 4-1: DC motor control schematic

Created with EAGLE

Before you hook up your DC motor, it's important to understand what all these components are doing:

- Q1 is an NPN bipolar junction transistor (BJT) used for switching the separate 9V supply to the motor. There are two types of BJTs, NPN and PNP, which refer to the different semiconductor “doping” techniques used to create the transistor. This book will focus on using NPN BJTs. You can simply think of an NPN transistor as an electrically controlled switch that allows you to inhibit or allow current flow.
- A $1\text{k}\Omega$ resistor is used to separate the transistor's base pin from the control pin of the Arduino. It limits the current that flows to and from the gate pin.
- U1 is the DC motor.
- C1 is for filtering noise caused by the motor.
- D1 is a diode used to protect the power supply from reverse voltage caused by the motor acting like an inductor. This is commonly called a *flyback*, *snubber*, or *freewheeling* diode.

Using Transistors as Switches

Transistors can be used for a multitude of tasks, from making amplifiers to making up the CPU inside your computer and smartphone. You can use a single transistor to make a simple electrically controlled switch. Every BJT has three pins (see Figure 4-2):

the emitter (E), the collector (C), and the base (B). Note that the order of the pins on the physical package is not always the same as the order shown in Figure 4-2; be sure to read the datasheet for your specific model of transistor.

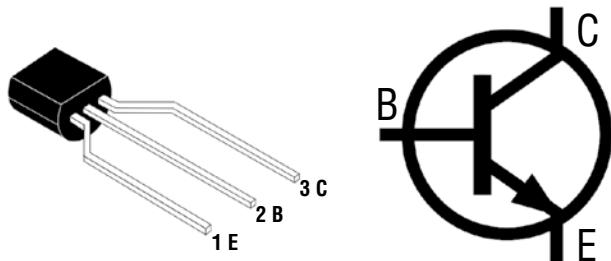


Figure 4-2: An NPN BJT

Credit: Wikipedia (Public Domain)

Current flows in through the collector and out of the emitter. By modulating the base pin, you can control whether current is permitted to flow. When a sufficiently high voltage and current are applied to the base, the transistor will operate in “saturation mode,” effectively allowing current to flow freely from the collector to the emitter as if flowing through a closed switch. When in saturation mode, the motor, when connected as shown in Figure 4-1, will spin. The 5V generated by the Arduino I/O pins biases the transistor base at a sufficiently high voltage to turn on the transistor.

By taking advantage of PWM, you can control the speed of the motor by rapidly turning the transistor on and off. Because the motor can maintain momentum, the duty cycle of the PWM signal determines the motor’s speed. The transistor is essentially connecting and disconnecting one terminal of the motor from the ground and determining when a complete circuit can be made with the battery.

Using Protection Diodes

It is important to consider issues caused by DC motors acting like inductors. (*Inductors* are electrical devices that store energy in their magnetic fields and resist changes in current.) As the DC motor spins, energy is built up and stored in the inductance of the motor coils. If power is instantaneously removed from the motor, the energy is dissipated in the form of an inverted voltage spike, which could prove harmful to the power supply. That’s where a protection (or flyback) diode comes in. By putting the diode across the motor, you ensure that the current generated by the motor flows through the diode and that the reverse voltage cannot exceed the forward voltage of the diode (because diodes allow current to flow in one direction only). This will also absorb any current that is generated if you forcibly turn the motor.

Using a Secondary Power Source

Note, from the circuit diagram shown in Figure 4-1, that the power supply to the motor is 9V, instead of the usual 5V from the USB connection that you've been using. For the purposes of this experiment, a 9V battery suffices, but you could also use an AC/ DC wall adapter. There are two reasons for using a power source separate from the Arduino's built-in 5V supply:

1. By using a separate supply, you reduce the possibility that improper wiring of a higher-power circuit could harm your Arduino.
2. You can take advantage of higher current limits and higher voltages.

Some DC motors can consume more current than the Arduino 5V supply can offer. Further, many motors are rated at voltages higher than 5V. Although they might spin at 5V, you can reach their maximum speed at only 9V or 12V (depending on the motor specifications).

All of the wiring diagrams illustrated in this chapter show the use of a battery clip with two free wire leads that can be plugged directly into your breadboard. If you instead have a 9V battery clip that came with a barrel jack connector, you can plug that into the barrel jack on your Arduino. This will expose the 9V from the battery on the "Vin" pin of your Arduino. You can use a jumper wire to connect the Vin pin to the collector of the transistor.

Note that you must connect the ground of both your secondary power supply and the Arduino ground. This connection ensures a common reference point between the voltage levels in the two parts of the circuit.

Wiring the Motor

Now that you understand the intricacies of controlling a brushed DC motor, it's time to get it wired up on your breadboard. Try to wire it by only referencing the schematic shown in Figure 4-1. After you've tried to assemble the circuit using only the schematic, reference the graphical version shown in Figure 4-3 to confirm that you wired it correctly.

It's important to become proficient at reading electrical schematics without having to look at a graphical layout. Did you wire it correctly? Remember to check for the following as you wire up the circuit:

1. Make sure that you've connected the ground from your 9V battery to the ground from your Arduino. You might want to use the horizontal bus on the breadboard to accomplish this, as shown in Figure 4-3.

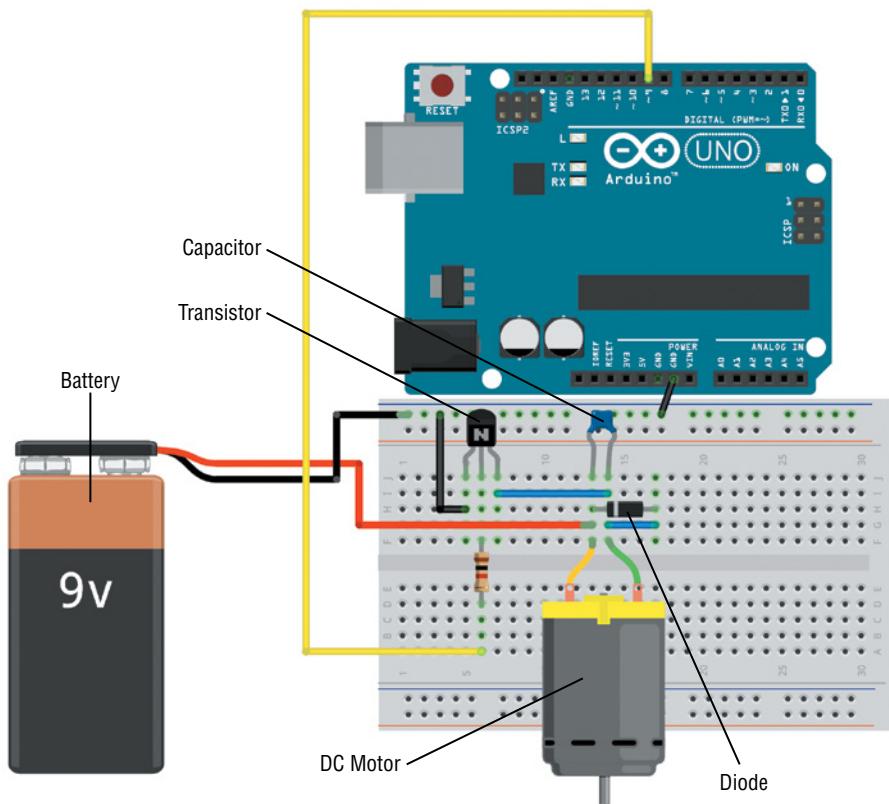


Figure 4-3: DC motor wiring

Created with Fritzing

2. Make sure that the 9V supply is not connected to the 5V supply. In fact, you don't even need to wire the 5V supply to the breadboard for this exercise.
3. Make sure that the orientation of your transistor is correct. If you aren't using the same NPN BJT listed in the parts list for this chapter, reference the datasheet to ensure that the emitter, base, and collector are connected to the same pins. If they are not, adjust your wiring.
4. Make sure that the orientation of the diode is correct. Current flows from the side without a stripe to the side with the stripe. The stripe on the physical device matches the line in the schematic symbol. In addition to the diode, a ceramic capacitor is also installed across the motor terminals to reduce electrical noise created by the bushes. Ceramic capacitors are not polarized, so you don't have to worry about the capacitor's insertion direction like you do with the diode.

Next up, it's time to get this motor spinning. You might want to attach a piece of tape or a wheel to the end of the motor so that you can more easily see the speed at which it is spinning. Before you write the program, you can confirm that the circuit is working correctly by providing power to the Arduino over the USB connection, plugging in the 9V battery, and connecting the transistor's base pin (after the resistor) directly to 5V from the Arduino. This simulates a logic high command and should make the motor spin. Connecting that same wire to ground will ensure that it does not spin. If this doesn't work, check your wiring before moving on to the next step: programming.

Controlling Motor Speed with PWM

First up, to adjust the speed of your motor, you can use a program very similar to the one you used to adjust the LED brightness of your nightlight in Chapter 3. By instructing a PWM-capable pin on your Arduino to send varying duty-cycle signals to the transistor, the current flow through the motor rapidly starts and stops, resulting in a change in velocity. Try out the program in Listing 4-1 to repeatedly ramp the motor speed up and down.

Listing 4-1

Automatic speed control-motor.ino

```
//Simple Motor Speed Control Program

const int MOTOR=9;      //Motor on Digital Pin 9

void setup()
{
    pinMode (MOTOR, OUTPUT);
}

void loop()
{
    for (int i=0; i<256; i++)
    {
        analogWrite(MOTOR, i);
        delay(10);
    }
    delay(2000);
    for (int i=255; i>=0; i--)
    {
        analogWrite(MOTOR, i);
        delay(10);
    }
    delay(2000);
}
```

If everything is hooked up correctly, this code should slowly ramp the motor speed up, then back down again in a loop. Using these techniques, you could easily make a simple roving robot.

Next up, you can combine your new knowledge of DC motors with your knowledge of analog sensors. Using a potentiometer, you can manually adjust the motor speed. To begin, add a potentiometer to analog pin 0, as shown in Figure 4-4. Note that you must connect the 5V pin from the Arduino to the power rail on the breadboard if you want to connect the potentiometer to that row on the board.

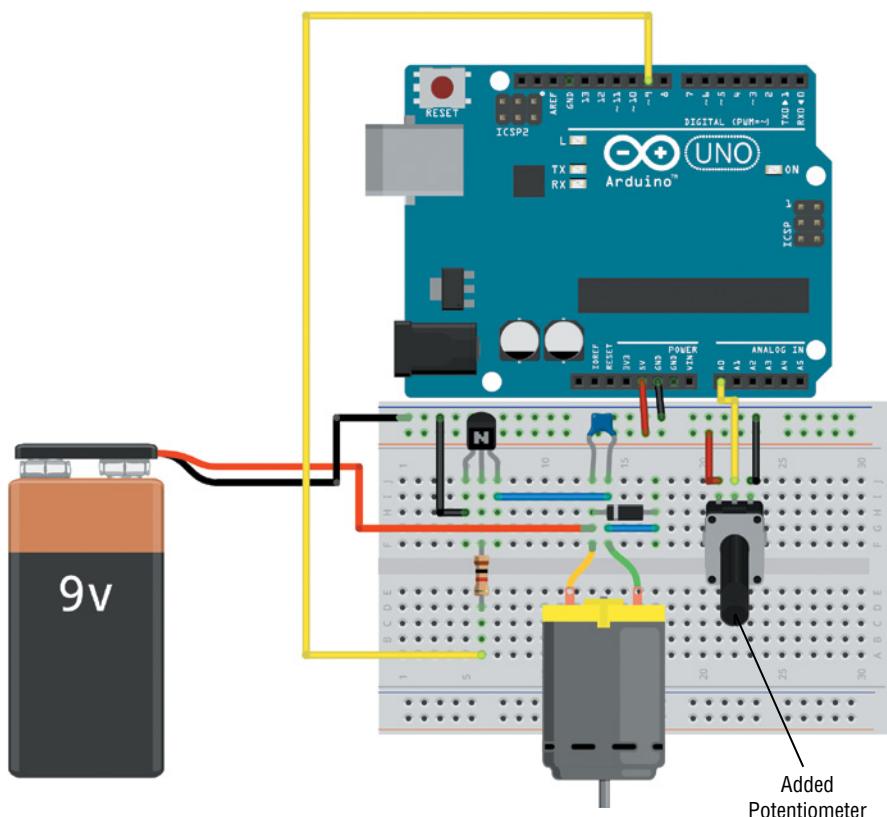


Figure 4-4: Adding a potentiometer

Created with Fritzing

You can now modify the program to control the motor speed based on the present setting of the potentiometer. With the potentiometer at zero, the motor stops; with the potentiometer rotated fully, the motor runs at full speed. Recall that the Arduino is running quite fast; it's actually running through the loop several thousand times every second! Therefore, you can simply check the potentiometer speed each time through the loop and adjust the motor speed after each check. It checks often enough

that the motor speed adjusts in real time with the potentiometer. The code in Listing 4-2 allows you to do this. Create a new sketch (or update your previous sketch to match this code) and upload it to your Arduino from the integrated development environment (IDE).

Listing 4-2

Adjustable speed control-motor_pot.ino

```
//Motor Speed Control with a Pot

const int MOTOR=9; //Motor on Digital Pin 9
const int POT=0; //POT on Analog Pin 0

int val = 0;

void setup()
{
    pinMode (MOTOR, OUTPUT);
}

void loop()
{
    val = analogRead(POT);
    val = map(val, 0, 1023, 0, 255);
    analogWrite(MOTOR, val);
}
```

A lot of this code should look familiar from your previous experience with analog sensors. Note that the `constrain()` function is not required when you're using a potentiometer, because you can use the entire input range, and the value will never go below 0 or above 1023. After uploading the code to your Arduino, adjust the pot and observe the speed of the motor changing accordingly.

Using an H-Bridge to Control DC Motor Direction

So, now you can change DC motor speed. This is great for making wheels turn on an Arduino-controlled robot—as long as you only want it to drive forward. Any useful DC motor needs to be able to spin in two directions. To accomplish this, you can use a handy device called an *H-bridge*. The operation of an H-bridge can best be explained with a diagram, as shown in Figure 4-5.

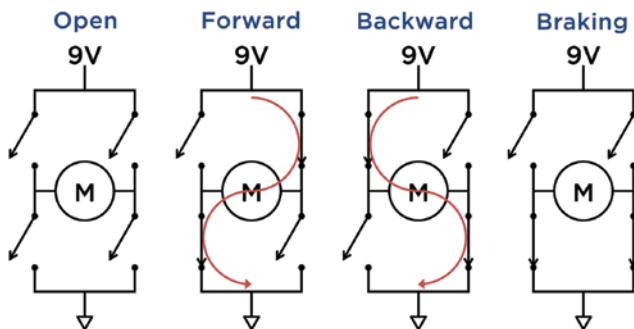


Figure 4-5: H-bridge operation

Can you figure out why it's called an H-bridge? Notice that the motor in combination with the four switches forms an uppercase *H*. Although the diagram shows them as switches, the switching components are actually transistors, similar to the ones you used in the previous exercise. Some additional circuitry, including protection diodes, is also built in to the H-bridge integrated circuit.

The H-bridge has four main states of operation: open, forward, backward, and braking. In the open state, all the switches are open and the motor doesn't spin. In the forward state, two diagonally opposing switches are engaged, causing current to flow from 9V, through the motor, and down to ground. When the opposing switches are flipped, current then runs through the motor in the opposite direction, causing it to spin backward. If the H-bridge is put in the braking state, all residual motion caused by momentum is ceased, and the motor stops.

CREATING SHORT CIRCUITS WITH H-BRIDGES

Be aware of one extremely important consideration when using H-bridges: what would happen if both switches on the left or both switches on the right were closed? It would cause a direct short between 9V and ground. If you've ever shorted a 9V battery before, you know that this is not something you want to do. A shorted battery heats up very quickly, and, in rare circumstances, could burst or leak. Furthermore, a short could destroy the H-bridge or other parts of the circuit. Using an H-bridge can lead to a rare scenario where you could potentially destroy a piece of hardware by programming something incorrectly.

For this experiment, you use an L293D quadruple half-H driver from Texas Instruments. This chip has a built-in thermal shutdown that should kick in before a short circuit destroys anything, but it's still a good idea to be cautious. This chip also has built-in flyback diodes, so there is no need to include them externally as you did when using the single transistor.

WARNING To ensure that you don't blow anything up, *always* disable the chip before flipping the states of any of the switches. This ensures that a short cannot be created even when you quickly switch between motor directions. You'll use three control pins: one for controlling the top two gates, one for controlling the bottom two gates, and one for enabling the circuit.

Building an H-Bridge Circuit

With the preceding considerations in mind, it's time to build the circuit. The H-bridge chip you will use is the L293D quadruple half-H driver. Two half-H drivers are combined into one full-H driver, such as the one shown in Figure 4-5. For this exercise, you just use two of the four half-H drivers to drive one DC motor. If you want to create something more advanced, you can use this chip to control two DC motors. For example, to make an RC car, you would use one for the left wheels and one for the right wheels. Before you actually get it wired up, take a look at the pin-out and logic table from the part's datasheet, shown in Figure 4-6.

Pin numbering on integrated circuits (ICs) always starts at the top-left pin and goes around the part counterclockwise. Chips will always have some kind of indicator to show which pin is pin 1, so that you don't plug the IC in upside-down. On through-hole parts (which you will use exclusively in this chapter), a half circle on one end of the chip indicates the top of the chip (where pin 1 is located). Some chips may have a small circle marked next to pin 1 on the plastic casing in addition to, or instead of, the half circle.

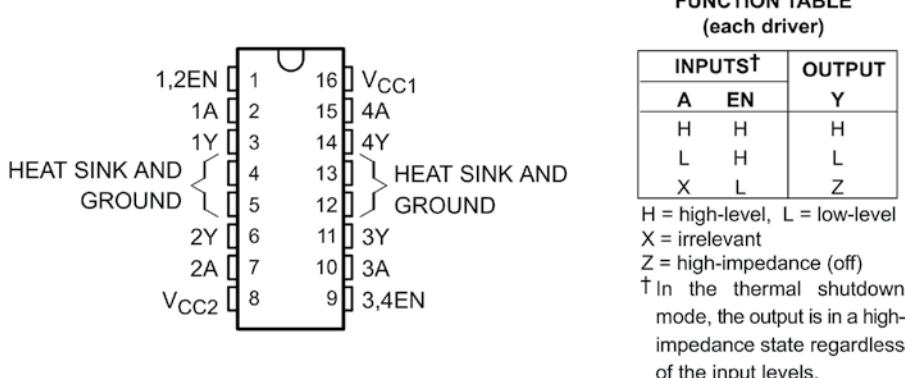


Figure 4-6: H-bridge pin-out and logic table

Credit: Courtesy of Texas Instruments Incorporated

Let's run through the pins and how you'll be using them:

- **GROUND/HEATSINK (pins 4, 5, 12, and 13).** The four pins in the middle connect to a shared ground between your 9V and 5V supplies. They also heatsink the driver into the ground of your circuit. On a printed circuit board, chips are often designed to shunt excess heat to ground because the ground often has the most copper surface area.
- **VCC2 (pin 8).** VCC2 supplies the motor current, so you connect it to 9V.
- **VCC1 (pin 16).** VCC1 powers the chip's logic, so you connect it to 5V.
- **1Y and 2Y (pins 3 and 6).** These are the outputs from the left driver. The motor wires connect to these pins.
- **1A and 2A (pins 2 and 7).** The states of the switches on the left are controlled by these pins, so they are connected to I/O pins on the Arduino for toggling.
- **1,2EN (pin 1).** This pin is used to enable or disable the left driver. It is connected to a PWM pin on the Arduino, so that speed can be controlled dynamically.
- **3Y and 4Y (pins 11 and 14).** These are the outputs from the right driver. Because you are using the left driver only, you can leave them disconnected.
- **3A and 4A (pins 10 and 15).** The states of the switches on the right are controlled by these pins, but you are using only the left driver in this example, so you can leave them disconnected.
- **3,4EN (pin 9).** This pin is used to enable or disable the right driver. Because you will not be using the right driver, you can disable it by connecting this pin directly to ground.

For reference, confirm your wiring with Figure 4-7. Keep the potentiometer wired as it was before.

You can confirm that the circuit is working before you program it by hooking up the enable pin to 5V, hooking up one of the A pins to ground, and connecting the other A pin to 5V. You can reverse direction by swapping what the A pins are connected to.

WARNING You should disconnect the 9V battery while swapping the A pins to ensure that you don't cause an accidental short circuit within the H-bridge.

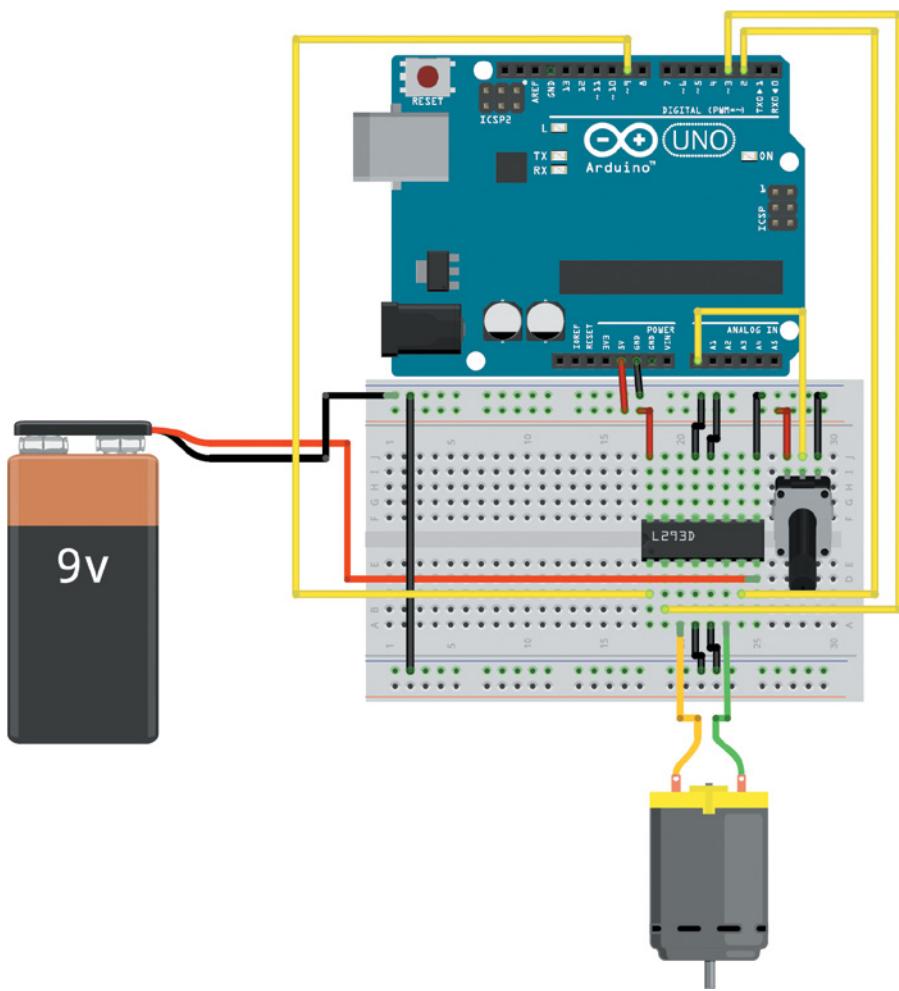


Figure 4-7: H-bridge wiring diagram

Created with Fritzing

Operating an H-Bridge Circuit

Next up, you write a program to control the motor's direction and speed using the potentiometer and the H-bridge. Setting the potentiometer in a middle range stops the motor; setting the potentiometer in a range above the middle increases the speed forward; and setting the potentiometer in a range below the middle increases the speed backward. This is another perfect opportunity to employ functions in your Arduino program. You can write a function to stop the motor, a function to cause it to spin forward at a set speed, and a function to cause it to spin backward at a set speed. Ensure that you correctly disable

the H-bridge at the beginning of the function before changing the motor mode; doing so reduces the probability that you will make a mistake and accidentally short out the H-bridge.

Following the logic diagram from Figure 4-6, you can quickly figure out how you need to control the pins to achieve the desired results:

- To stop current flow through the device, set the enable pin low.
- To set the switches for rotation in one direction, set one switch high and the other switch low.
- To set switches for rotation in the opposite direction, swap which one is high and which is low.
- To cause the motor to stop immediately, set both switches low.

NOTE Always disable the current flow before changing the state of the switches to ensure that a momentary short cannot be created as the switches flip.

First, you should devise the functions that safely execute the previously described motions. Create a new Arduino sketch and start by writing your new functions:

```
//Motor goes forward at given rate (from 0-255)
void forward (int rate)
{
    digitalWrite(EN, LOW);
    digitalWrite(MC1, HIGH);
    digitalWrite(MC2, LOW);
    analogWrite(EN, rate);
}

//Motor goes backward at given rate (from 0-255)
void reverse (int rate)
{
    digitalWrite(EN, LOW);
    digitalWrite(MC1, LOW);
    digitalWrite(MC2, HIGH);
    analogWrite(EN, rate);
}

//Stops motor
void brake ()
{
    digitalWrite(EN, LOW);
    digitalWrite(MC1, LOW);
    digitalWrite(MC2, LOW);
    digitalWrite(EN, HIGH);
}
```

Note that at the beginning of each function, the *EN* pin is always set low, and then the *MC1* and *MC2* pins (motor control pins) are adjusted. When that is done, the current flow can be reenabled. To vary the speed, just use the same technique you did before. By using PWM, you can change the duty cycle with which the *EN* pin is toggled, thus controlling the speed. The *rate* variable must be between 0 and 255. The main loop takes care of setting the right *rate* from the input potentiometer data.

Next, consider the main program loop:

```
void loop()
{
    val = analogRead(POT);

    //go forward
    if (val > 562)
    {
        velocity = map(val, 563, 1023, 0, 255);
        forward(velocity);
    }

    //go backward
    else if (val < 462)
    {
        velocity = map(val, 461, 0, 0, 255);
        reverse(velocity);
    }

    //brake
    else
    {
        brake();
    }
}
```

In the main loop, the potentiometer value is read, and the appropriate function can be called based on the potentiometer value. Recall that analog inputs are converted to digital values between 0 and 1023. Refer to Figure 4-8 to better understand the control scheme and compare that with the preceding loop code.



Figure 4-8: Motor control plan

When the potentiometer is within the 100-unit range surrounding the midpoint, the brake function is called. As the potentiometer value increases from 562 to 1023, the speed forward increases. Similarly, the speed increases in the reverse direction between potentiometer values of 462 and 0. The `map` function should look familiar to you from the previous chapter. Here, when determining the reverse speed, note the order of the variables: 461 is mapped to 0, and 0 is mapped to 255; the `map` function can invert the mapping when the variables are passed in descending order. Putting the loop together with the functions, and the `setup()`, you get a completed program that looks like the one shown in Listing 4-3. Ensure that your program matches the one here and load it onto your Arduino.

Listing 4-3

H-bridge potentiometer motor control-hbridge.ino

```
//H-bridge Motor Control
const int EN=9;      //Half Bridge 1 Enable
const int MC1=3;    //Motor Control 1
const int MC2=2;    //Motor Control 2
const int POT=0;    //POT on Analog Pin 0

int val = 0;        //for storing the reading from the POT
int velocity = 0;   //For storing the desired velocity (from 0-255)

void setup()
{
    pinMode(EN, OUTPUT);
    pinMode(MC1, OUTPUT);
    pinMode(MC2, OUTPUT);
    brake(); //Initialize with motor stopped
}

void loop()
{
    val = analogRead(POT);

    //go forward
    if (val > 562)
    {
        velocity = map(val, 563, 1023, 0, 255);
        forward(velocity);
    }

    //go backward
    else if (val < 462)
```

```
{  
    velocity = map(val, 461, 0, 0, 255);  
    reverse(velocity);  
}  
  
//brake  
else  
{  
    brake();  
}  
}  
  
//Motor goes forward at given rate (from 0-255)  
void forward (int rate)  
{  
    digitalWrite(EN, LOW);  
    digitalWrite(MC1, HIGH);  
    digitalWrite(MC2, LOW);  
    analogWrite(EN, rate);  
}  
  
//Motor goes backward at given rate (from 0-255)  
void reverse (int rate)  
{  
    digitalWrite(EN, LOW);  
    digitalWrite(MC1, LOW);  
    digitalWrite(MC2, HIGH);  
    analogWrite(EN, rate);  
}  
  
//Stops motor  
void brake ()  
{  
    digitalWrite(EN, LOW);  
    digitalWrite(MC1, LOW);  
    digitalWrite(MC2, LOW);  
    digitalWrite(EN, HIGH);  
}
```

Does everything work as expected? If not, make sure that you wired up your circuit correctly. As an additional challenge, grab a second DC motor and hook it up to the other half of the H-bridge chip. You should be able to drive two motors simultaneously with minimal effort.

Building a Roving Robot

Now that you've learned how to use an H-bridge to drive one or two DC motors forward and backward, you can apply that knowledge to building a simple roving robot!

Choosing the Robot Parts

Building a robot can seem like a daunting task, but you'll find that you already have all the skills to make one. At its simplest, a robot just needs two things: sensors or inputs that tell it what to do, and actuators that translate its intents into physical actions.

Selecting a Motor and Gearbox

You likely noticed from your previous H-bridge circuit that the 9V motors rotate very fast. In fact, they move too fast to reasonably drive the wheels on a small robot. That's where gearboxes come in. Cars use a sophisticated gearbox called a transmission to balance speed with torque. When you change gears on a bike or in a car, you are adjusting a gear ratio that exchanges speed for rotational torque. The simplest gearbox is made up of just two gears—one connected to a motor shaft, and the other connected to the wheel (or anything else you want to turn).

The difference in size between two meshed gears results in one gear turning more slowly than the other. Consider an example with two gears where one has twice as many teeth as the other. If the smaller gear is connected to the motor shaft, then the larger gear will only complete half a rotation for every full rotation that the smaller gear completes. In this way, you've reduced the wheel drive speed to one-half of the motor speed, while increasing torque. See blum.fyi/gear-ratio for an animated example of this principle.

By combining a bunch of these gear reductions, you get a gearbox that can drive robot wheels at a reasonable speed. A quick web search will turn up hundreds of vendors that sell DC motors with various-sized gearboxes already attached. The larger the reduction ratio, the more motor power you'll realize, at a slower speed. For the following example project, I recommend using the Adafruit geared DC motors in the servo body—they are included with the robot chassis kit in the parts list for this chapter. You can use this kit, or you can get more creative and build a robot body out of wooden craft sticks and other household items. Whatever you decide to use for the robot chassis, you'll want some form of gear-reduced DC motors.

Powering Your Robot

Even though the DC motors with gearboxes that you'll be using for your robot operate at 5V (unlike the 9V DC motors used earlier in this chapter), you should still not power them directly from the Arduino's 5V output pin. The key metric on DC motors that you must check for is *stall current*. This represents the maximum current that the motor will consume when mechanically prevented from spinning. The recommended geared motors have a stall current between 550 mA and 650 mA, depending on their operating voltage. Even if you assume the 550 mA value, this would mean that they could consume over 1A of current at 5V when simultaneously stalled. This is more power than the Arduino's onboard voltage regulator can provide, so you must power

the motors in a different way. One option would be to power the Arduino through its barrel jack with the 9V battery, and to use a separate 5V battery pack for the motors. But putting two separate battery packs on a robot can feel a bit clunky. Thankfully, there's a better way.

You can use your 9V battery, paired with your own linear regulator to generate a separate 5V supply to be used only for the motors, and the Arduino will still be powered by its own onboard regulator (also feeding from the 9V battery). A linear regulator is an extremely simple device that generally has three pins: input voltage, output voltage, and ground. The ground pin is connected to both the ground of the input supply and the ground of the output. In the case of linear-voltage regulators, the input voltage must always be higher than the output voltage, and the output voltage is set at a fixed value depending on the regulator you use.

The voltage drop between the input and the output is burned off as heat, and the regulator ensures that the output always remains the same, even as the voltage of the input drops (in the case of a battery discharging over time). For these experiments, you use an L7805CV 5V voltage regulator, which is capable of supplying up to 1.5A at 5V. Figure 4-9 shows a schematic of how to hook up the regulator.

Note the capacitors on each side of the regulator. These are called *decoupling capacitors*, and are used to smooth out each voltage supply by charging and discharging to oppose ripples in the voltage. Ripples are small fluctuations to the nominal voltage caused by loads increasing and decreasing in the circuit. Most linear regulator data-sheets include a suggested circuit that includes ideal values and types for these capacitors based on your use case scenario. Also keep in mind that the 5V rail created by this regulator should be kept separate from the 5V power rail of the Arduino. Their grounds, however, should be tied together.

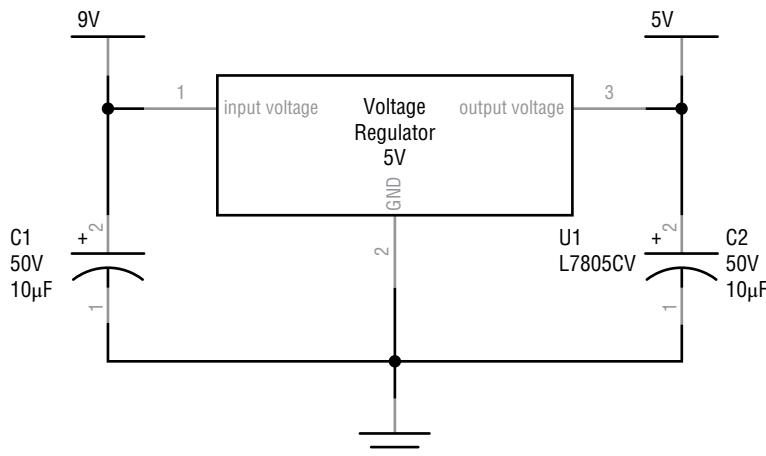


Figure 4-9: 5V linear regulator schematic

Created with Fritzing

UNDERSTANDING LINEAR REGULATORS AND THE LIMITS OF ARDUINO POWER SUPPLIES

Why is it necessary to use an external power supply when certain items require more current? There are few reasons. The I/O pins of your Arduino cannot supply more than 40 mA each. Because a motor can consume hundreds of millamps, the I/O pins are not capable of driving them directly. Even if they were, you wouldn't want to because of the damage that can be caused by inductive voltage spikes.

It makes sense that you need to use an external supply with a 9V DC motor because you need the higher voltage, but why does a 5V motor need an external supply if it is at the same voltage as the Arduino? The Arduino generates the 5V used for the logic either directly from the USB or by using a built-in linear regulator with the DC barrel jack as the supply voltage. When you use USB, a maximum of 500 mA is available to the Arduino and all its peripherals, because that is what the USB specification allows. When you use an external supply of sufficient current, the built-in regulator can supply up to about 1A to the components on the 5V rail. Some of this is used by the Arduino's onboard microcontroller. The rest is available to peripherals, but may still not be enough for components like motors.

Motors create current *spikes*—brief periods while spinning up where current consumption is very high. These current spikes can ripple on the 5V line, and can even be seen in other components, like LEDs. By keeping the supply for the motor on a separate rail, you ensure that this does not happen.

Insufficient current for a motor (DC or any other type) might also cause it to behave erratically.

Constructing the Robot

Figure 4-10 shows a schematic representation of the circuit that you'll use. Most of it should look familiar. Can you construct the circuit using only the schematic?

The similarly named flags on the schematic represent pins that are wired together. This helps to show connectivity without making the schematic a sloppy mess with lines criss-crossing everywhere. Similarly, all the pins with the ground symbol are connected to the same ground, and all the pins connected to the 5V rail are connected together and getting 5V from the linear regulator. The circuit uses the same H-bridge circuit that you've been using. Now, however, the H-bridge input power will come from the new 5V motor supply that you are generating with the linear regulator because these DC motors are 5V, not 9V.

While you're wiring, keep in mind a few important things. First, ensure that you have the orientation of the regulator correct. With the metal tab on the side farthest from you, connect the battery voltage (V_{in} , 9V) to the leftmost pin, the ground to the

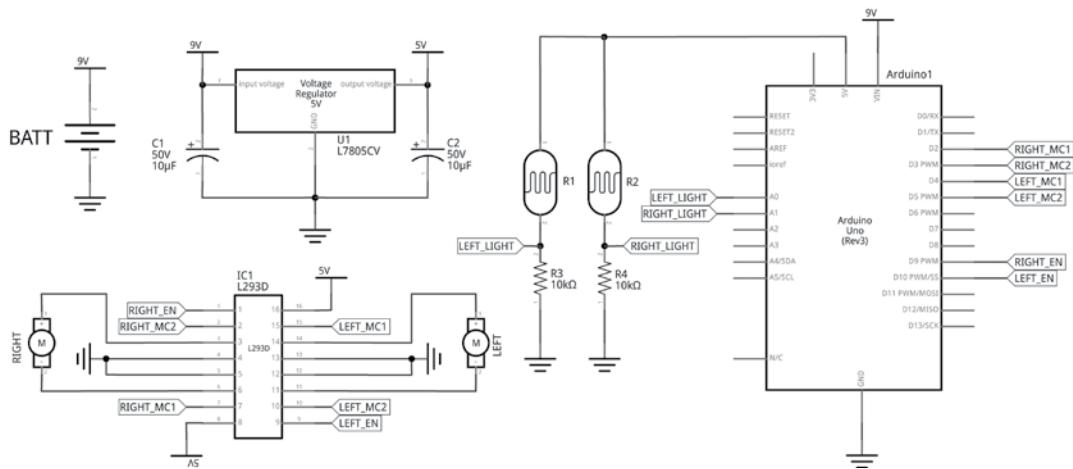


Figure 4-10: Roving robot schematic

Created with Fritzing

center pin, and the 5V motor output power line to the right pin. Second, if you're using polarized electrolytic capacitors, make sure to put them in the correct direction. The stripe indicates the negative terminal and should be connected to the common ground. Make sure that the pins don't touch; otherwise, it could cause a short.

You'll use two photoresistors as sensors that will drive the car. Aiming a flashlight at the right sensor will make the car turn right, and aiming a flashlight at the left sensor will make the car turn left. Aiming your light at the front of the car should equally drive both the left and right motors, making the car go straight. You'll use two analog input pins for these sensors. One pin of the voltage divider circuit that you use for these sensors connects to the Arduino's 5V output pin. Note that this should be kept separate from the 5V rail that you are generating with the linear regulator on your breadboard.

After you're all wired up, move on to the next section to write the software for your robot. Figure 4-11 shows a visual representation of the breadboard layout for you to cross-check against.

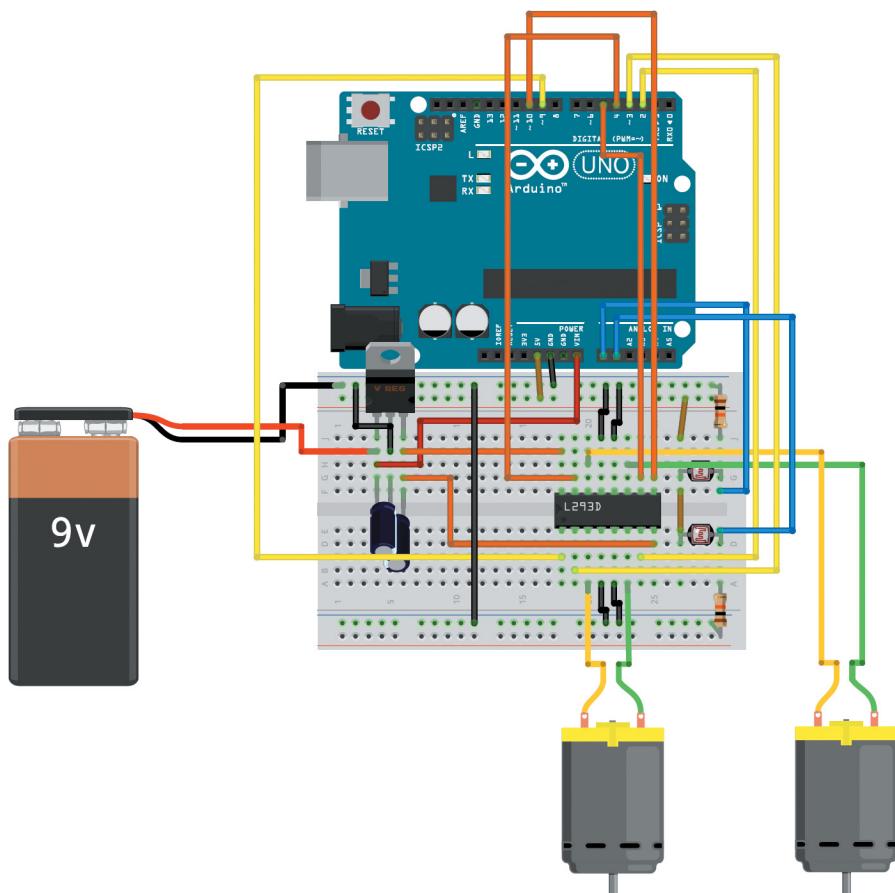


Figure 4-11: Roving robot breadboard

Created with Fritzing

I recommend doing a dry run to get the software right and the breadboard laid out correctly before you mount it into your robot chassis. Figure 4-12 shows a photo of the completed breadboard, Arduino, and battery. Note how the photoresistors are pointed forward, but angled slightly away from the center to ensure that you can clearly differentiate between light coming from the left or right of the robot.

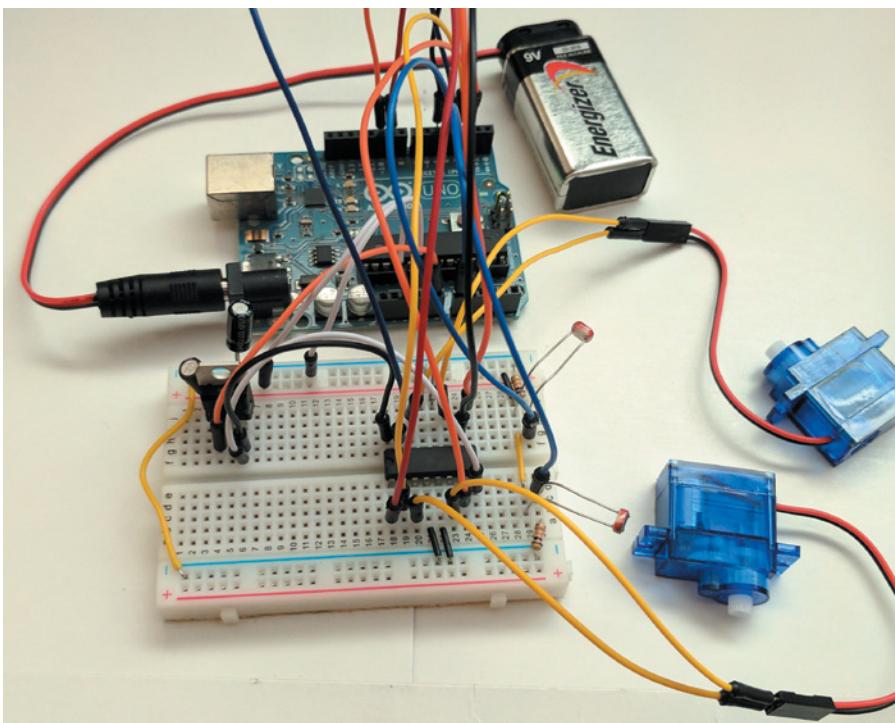


Figure 4-12: Roving robot electronics

Writing the Robot Software

Now that you've got your circuit built, you'll need to adapt your previous code to control your self-driving roving robot. The complete code needed to program your car is in Listing 4-4. Much of it is the same as your code from previous examples in this and prior chapters. The motor control movement functions have been reused, with one added argument that defines which motor is to be controlled—left or right. The listed code only drives the motors forward, but you can modify it—for example, if you want to make the robot back up in low-light conditions.

Constants at the top of the code define the light levels and motor speeds to be used. You will need to calibrate these constants to the light levels that you observe in your environment. To do so, the program includes Serial print statements that will print the present light levels sensed by each photoresistor. Load the program onto your Arduino and open the serial terminal. Shine a flashlight at each of the sensors to determine what minimum and maximum analog values you see. Then use those values to set the minimum and maximum light threshold levels. This concept is shown in the video that accompanies this project on the Exploring Arduino website.

The logic for actually driving the car can all be found in the main loop(). Each sensor is read, and the map() and constrain() functions are used to map the brightness level to the drive speed of the opposing motor. When there is more light on the left sensor, the right motor will move faster to turn the car towards the left, and vice versa.

Listing 4-4

Self-driving roving robot-car.ino

//Self-Driving Car - Follows Light!

```
//H-Bridge Pins
const int RIGHT_EN = 9; //Half Bridge Enable for Right Motor
const int RIGHT_MC1 = 2; //Right Bridge Switch 1 Control
const int RIGHT_MC2 = 3; //Right Bridge Switch 2 Control
const int LEFT_EN = 10; //Half Bridge Enable for Left Motor
const int LEFT_MC1 = 4; //Left Bridge Switch 1 Control
const int LEFT_MC2 = 5; //Left Bridge Switch 2 Control

//Light Sensor Pins
const int LEFT_LIGHT_SENSOR = 0; //Photoresistor on Analog Pin 0
const int RIGHT_LIGHT_SENSOR = 1; //Photoresistor on Analog Pin 1

//Movement Thresholds and Speeds
const int LIGHT_THRESHOLD_MIN = 810; //The min light level reading to
cause movement
const int LIGHT_THRESHOLD_MAX = 1100; //The max light level reading to
cause movement
const int SPEED_MIN = 150; //Minimum motor speed
const int SPEED_MAX = 255; //Maximum motor speed

void setup()
{
    //The H-Bridge Pins are Outputs
    pinMode(RIGHT_EN, OUTPUT);
    pinMode(RIGHT_MC1, OUTPUT);
    pinMode(RIGHT_MC2, OUTPUT);
```

```
pinMode(LEFT_EN, OUTPUT);
pinMode(LEFT_MC1, OUTPUT);
pinMode(LEFT_MC2, OUTPUT);

//Initialize with both motors stopped
brake("left");
brake("right");

//Run a Serial interface for helping to calibrate the light levels.
Serial.begin(9600);
}

void loop()
{
    //Read the light sensors
    int left_light = analogRead(LEFT_LIGHT_SENSOR);
    int right_light = analogRead(RIGHT_LIGHT_SENSOR);

    //A small delay of 50ms so the Serial Output is readable
    delay(50);

    //For each light sensor, set speed of opposite motor proportionally.
    //Below a minimum light threshold, do not turn the opposing motor.
    //Note: Left Sensor controls right motor speed, and vice versa.
    //      To turn left, you need to speed up the right motor.
    Serial.print("Right: ");
    Serial.print(right_light);
    Serial.print(" ");

    if (right_light >= LIGHT_THRESHOLD_MIN)
    {
        //Map light level to speed and constrain it
        int left_speed = map(right_light,
                             LIGHT_THRESHOLD_MIN, LIGHT_THRESHOLD_MAX,
                             SPEED_MIN, SPEED_MAX);
        left_speed = constrain(left_speed, SPEED_MIN, SPEED_MAX);
        Serial.print(left_speed); //Print the drive speed
        forward("left", left_speed); //Drive opposing motor at computed speed
    }
    else
    {
        Serial.print("0");
        brake("left"); //Brake the opposing motor when light is below the min
    }

    Serial.print("\tLeft: ");
    Serial.print(left_light);
    Serial.print(" ");
    if (left_light >= LIGHT_THRESHOLD_MIN)
```

```
{  
    //Map light level to speed and constrain it  
    int right_speed = map(left_light,  
                           LIGHT_THRESHOLD_MIN, LIGHT_THRESHOLD_MAX,  
                           SPEED_MIN, SPEED_MAX);  
    right_speed = constrain(right_speed, SPEED_MIN, SPEED_MAX);  
    Serial.println(right_speed); //Print the drive speed  
    forward("right", right_speed); //Drive opposing motor at computed speed  
}  
else  
{  
    Serial.println("0");  
    brake("right"); //Brake the opposing motor when light is below the min  
}  
}  
  
//Motor goes forward at given rate (from 0-255)  
//Motor can be "left" or "right"  
void forward (String motor, int rate)  
{  
    if(motor == "left")  
    {  
        digitalWrite(LEFT_EN, LOW);  
        digitalWrite(LEFT_MC1, HIGH);  
        digitalWrite(LEFT_MC2, LOW);  
        analogWrite(LEFT_EN, rate);  
    }  
    else if(motor == "right")  
    {  
        digitalWrite(RIGHT_EN, LOW);  
        digitalWrite(RIGHT_MC1, HIGH);  
        digitalWrite(RIGHT_MC2, LOW);  
        analogWrite(RIGHT_EN, rate);  
    }  
}  
  
//Stops motor  
//Motor can be "left" or "right"  
void brake (String motor)  
{  
    if(motor == "left")  
    {  
        digitalWrite(LEFT_EN, LOW);  
        digitalWrite(LEFT_MC1, LOW);  
        digitalWrite(LEFT_MC2, LOW);  
        digitalWrite(LEFT_EN, HIGH);  
    }  
}
```

```
else if(motor == "right")
{
    digitalWrite(RIGHT_EN, LOW);
    digitalWrite(RIGHT_MC1, LOW);
    digitalWrite(RIGHT_MC2, LOW);
    digitalWrite(RIGHT_EN, HIGH);
}
}
```

Bringing It Together

Adafruit sells robot parts and a chassis that make this project easy to complete. In this section, you'll learn how to use that kit to assemble the roving robot. However, you might also want to be a bit more adventurous, and build your own chassis, and that's great! You can print a chassis on a 3D printer, or even use household items to build your chassis. Regardless of which option you go with, I still recommend that you use the geared DC motors and wheels available from Adafruit.

Start by attaching the geared DC motors to the chassis. Use the provided nuts and bolts if you're using the chassis, or simply hot glue them to the side of your own chassis. Then, attach the wheels and screw them into the motor gearbox shafts. Place your breadboard on top of the chassis, and wire the motors to the breadboard. Later, you may find that you need to swap the polarity of the motor connections or the motor control pins if one of the motors is spinning backward instead of forward. Figure 4-13 shows what your finished car should look like (the googly eyes are optional).

As you're putting the finishing touches on your car, keep a few things in mind. If you're using the chassis from Adafruit (or any metal chassis), make sure you isolate the bottom of the Arduino from the chassis. As shown in Figure 4-13, I accomplished this by using bolts and plastic nuts to keep the bottom of the Arduino separated from the metal. Even if your chassis is anodized, you should isolate your Arduino from it using spacers or simple tape. If any of the anodizing gets scratched off by the pins on the underside of the Arduino, you could create a short circuit on your Arduino! Similarly, be careful with the capacitors, voltage regulator, and other tall components on your breadboard. If you bend the leads to make them fit, make sure you don't short the leads together by mistake.

When you're finished putting everything together, plug in the 9V battery and use a flashlight to guide your self-driving car! It should follow your flashlight. If you find that it's moving too slowly, is over- or under-sensitive to light, or is acting strangely, plug it back into your USB port, and use the USB serial printouts to analyze the light levels and motor speeds. You may need to further adjust your software thresholds as described earlier, or you may need to re-aim your photoresistors. If your room is very bright, consider turning off the lights and closing the shades so that it's easier to

differentiate the light from your flashlight. If one of the motors is spinning backward when it should be spinning forward, just flip the polarity of its pins.

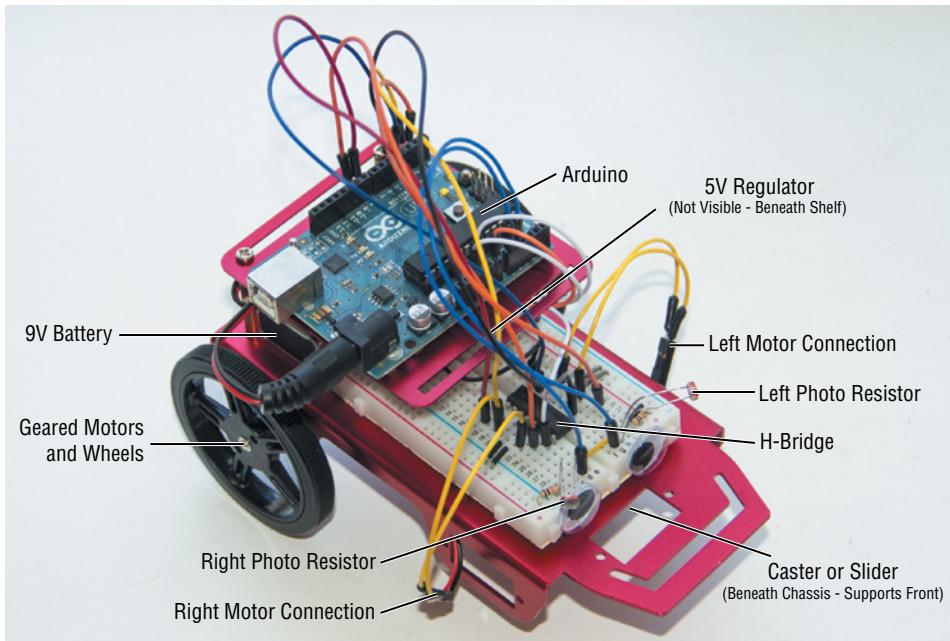


Figure 4-13: Fully built autonomous rover

NOTE You can watch a demo video of the roving robot online at exploringarduino.com/content2/ch4. You can also find this video on the Wiley website shown at the beginning of this chapter.

Summary

In this chapter, you learned the following:

- DC motors use electromagnetic induction to create mechanical action from changes in current.
- Motors are inductive loads that must utilize proper protection and power circuitry to interface safely with your Arduino.
- DC motor speed and direction can be controlled with PWM and an H-bridge.
- You can combine actuators like motors with analog inputs to your Arduino to build interactive robots and projects.

5

Driving Stepper and Servo Motors

Parts You'll Need for This Chapter:

- Arduino Uno or Adafruit METRO 328
- USB cable (Type A to B for Uno, Type A to Micro-B for METRO)
- Half-size or full-size breadboard
- Assorted jumper wires
- Pushbuttons ($\times 2$)
- $1k\Omega$ resistors ($\times 4$)
- $10k\Omega$ trim potentiometer
- 5 mm Blue LEDs ($\times 4$)
- 9V battery
- 9V battery clip
- L7805CV 5V voltage regulator
- $10\mu F$ 50V electrolytic capacitors ($\times 2$)
- TI L293D dual H-bridge motor driver
- 12V (> 500 mA) DC wall adapter
- Sharp GP2Y0A21YK0F IR distance sensor with JST cable
- Standard 5V servo motor
- NEMA-17 bipolar stepper motor
- Hot glue or tape
- Circular “clock face” (This can be a blank CD, or just paper.)

Binder clip

Popsicle stick

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch5

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

In Chapter 4, “Using Transistors and Driving DC Motors,” you mastered the art of driving DC motors. DC motors serve as excellent drive motors, but they are not recommended for precision work because they have no built-in feedback mechanism and they are velocity-controlled rather than being position-controlled. Without using an external encoder or positioning system of some kind, you will never know the absolute position of a DC motor. In contrast, servo motors, or servos, are unique in that you command them to rotate to an angular position and they stay there until you tell them to move to a new position. This is important for when you need to move your system to a known orientation. Examples include actuating door locks, moving armatures to specific rotations, and precisely controlling the opening of an aperture. Stepper motors are another kind of motor that “step” in precise increments; they’re perfect for building things like 3D printer gantries and precision gauges and instruments. In this chapter, you will learn about both servo motors and stepper motors. You’ll control both from your Arduino.

Driving Servo Motors

Servo motors are very popular for hobbyist and professional robotics work. You’ll find them in all sorts of products from RC airplanes to Internet-controlled door locks. They are available in a wide range of sizes and capabilities, with some modified for continuous rotation, and others designed for rotation over a small range with high torque.

Understanding the Difference between Continuous Rotation and Standard Servos

You can buy both standard and continuous rotation servos. Unmodified servos always have a fixed range (usually from 0 to 180 degrees) because there is a potentiometer in line with the drive shaft, which is used for reporting the present position. Servo control is achieved by sending a pulse of a particular length. In the case of a standard rotation servo, the length of the pulse determines the absolute position that the servo will rotate

to. If you remove the potentiometer, however, the servo is free to rotate continuously, and the pulse length sets the speed of the motor instead.

In this book, you use standard servos that rotate to an absolute position. You can experiment with continuous rotation servos either by opening a standard servo and carefully removing the potentiometer, or by buying premodified servos configured for continuous rotation.

Understanding Servo Control

Unlike their DC motor counterparts, servo motors have three pins: power (usually red), ground (usually brown or black), and signal or control (usually white or orange). These wires are color-coded, typically in the same order, and generally look like the ones shown in Figure 5-1. Some manufacturers may use non-standard ordering, so always be sure to check the datasheet to ensure you are wiring the servo correctly.

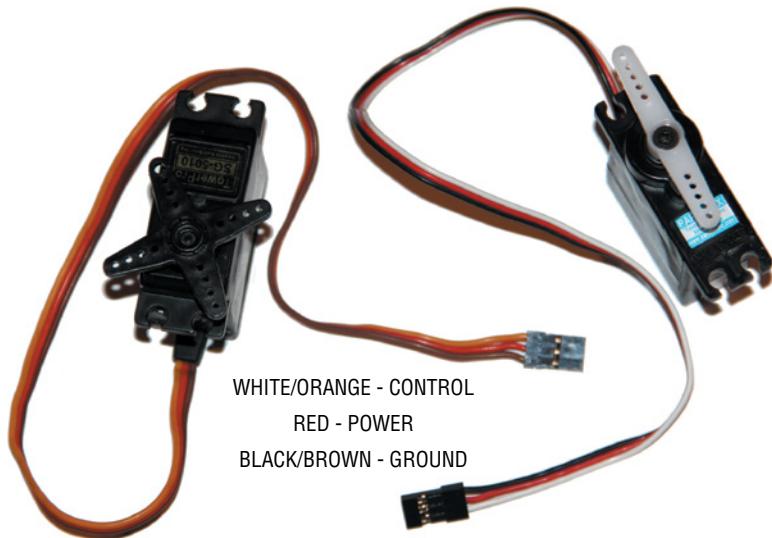


Figure 5-1: Servo motors

The color-coding might vary slightly between servos, but the color schemes just listed are the most common. (Check the servo's documentation if you're unsure.) Like DC motors, servos can draw quite a bit of a current (usually more than the Arduino can supply). Although you can sometimes run one or two small servos directly from the Arduino's 5V supply, you will generate a separate 5V power supply for the servos so that you have the option to add more if you need to (the same way you did for the 5V DC motors in the last chapter).

Unlike DC motors, servos have a dedicated control pin that instructs them what position to turn to. The power and ground lines of a servo should always be connected to a steady power source.

Servos are controlled using adjustable pulse widths on the signal line. For a standard servo, sending a 1 ms 5V pulse turns the motor to 0 degrees, and sending a 2 ms 5V pulse turns the motor to 180 degrees, with pulse lengths in the middle scaling linearly. A 1.5 ms pulse, for example, turns the motor to 90 degrees. Once a pulse has been sent, the servo turns to that position and stays there until another pulse instruction is received. However, if you want a servo to “hold” its position (resist being pushed on and try to maintain the exact position), you just resend the command once every 20 ms. The Arduino servo commands that you will later employ take care of this for you. To better understand how servo control works, study the timing diagram shown in Figure 5-2.

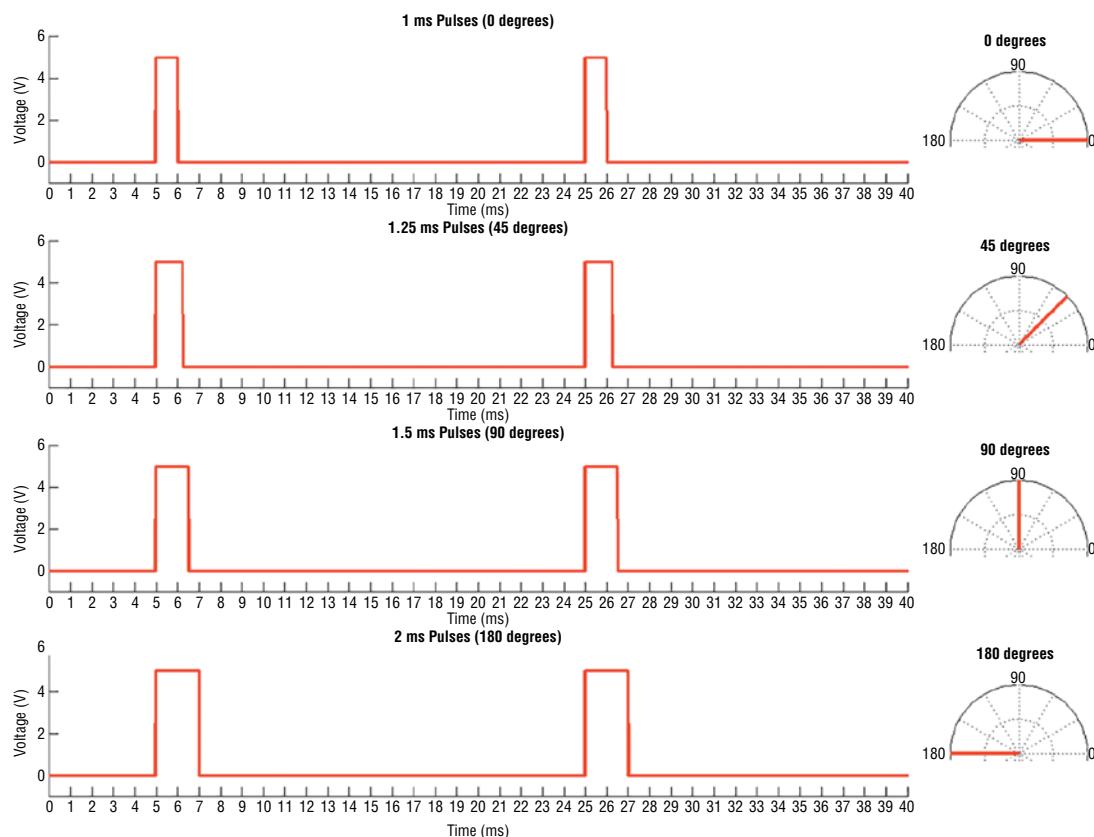


Figure 5-2: Servo motor timing diagram

Created with MATLAB

Note that in each of the examples in Figure 5-2, the pulse is sent every 20 ms. As the pulse length increases from 1 ms to 2 ms, the angle of rotation of the motor (shown to the right of the pulse graph) increases from 0 to 180 degrees.

As mentioned before, servos can draw more current than your Arduino may be able to provide. Most servos are designed to run at 5V. Just like you did with the small 5V DC motors that you used to build your roving car in the last chapter, you'll want to use a separate power source that can supply more current. To do this, you can employ the same L7805CV 5V voltage regulator circuit that you designed in Chapter 4, paired with a 9V battery.

NOTE Keep in mind that the 5V rail created by this regulator should be kept separate from the 5V power rail of the Arduino. Their grounds, however, should be tied together to ensure that they are working off the same reference.

Using all this information, it's time to wire up a servo. Referencing Figure 5-3, wire the servo, the 5V regulator, and the potentiometer. Connect the potentiometer to analog pin 0, connect the servo control pin to pin 9, and ensure that the 5V regulator's output supplies the servo's power.

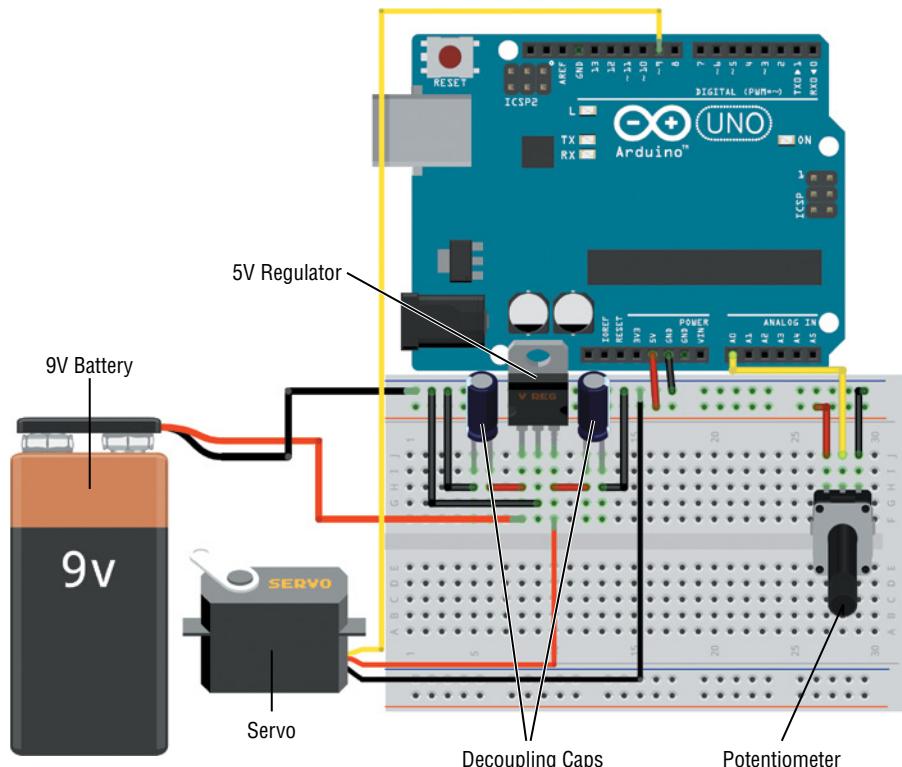


Figure 5-3: Servo experiment wiring diagram

Created with Fritzing

While you're wiring, keep a few important things in mind. First, recall what you learned about wiring the regulator in the last chapter: with the metal tab on the side farthest from you, connect the battery to the leftmost pin, the ground to the center pin, and the servo's power line to the rightmost pin. Second, if you're using polarized electrolytic capacitors (as in Figure 5-3), make sure to put them in the correct direction. The stripe indicates the negative terminal and should be connected to the common ground. Make sure that the pins don't touch; otherwise, it could cause a short. After you're all wired up, move on to the next section to learn how to program the servo controller.

Controlling a Servo

The Arduino IDE includes a built-in library that makes controlling servos a breeze. A software library is a collection of code that is useful, but not always needed in sketches. The Arduino IDE contains a number of libraries for common tasks. The servo library abstracts the timing routines you would need to write out on your own for pulsing the servo pin. All you have to do is attach a servo "object" to a particular pin and give it an angle to rotate to. The library takes care of the rest, even setting the pin as an output. The simplest way to test out the functionality of your servo is to map the potentiometer directly to servo positions. Turning the potentiometer to 0 moves the servo to 0 degrees, and moving it to 1023 moves the servo to 180 degrees. Create a new sketch with the code from Listing 5-1 and load it onto your Arduino to see this functionality in action.

Listing 5-1

Servo potentiometer control-servo.ino

```
//Servo Potentiometer Control
#include <Servo.h>

const int SERVO = 9; //Servo on Pin 9
const int POT    = 0; //POT on Analog Pin 0

Servo myServo;
int val = 0; //For storing the reading from the POT

void setup()
{
    //Attach the Servo Object
    myServo.attach(SERVO);
}
```

```
void loop()
{
    val = analogRead(POT);           //Read Pot
    val = map(val, 0, 1023, 0, 179); //scale it to servo range
    myServo.write(val);            //sets the servo
    delay(15);                   //waits for the servo
}
```

The include statement at the top of the program adds the functionality of the servo library to your sketch. `Servo myServo` makes a servo object called `myServo`. In your code, whenever you want to tell the servo what to do, you'll refer to `myServo`. In `setup()`, attaching the servo initializes everything necessary to control the servo. You can add multiple servos by calling the objects different things and attaching a different pin to each one. In `loop()`, the pot is read, scaled to an appropriate value for the servo control, and then “written” to the servo by pulsing the appropriate pin. The 15 ms delay ensures that the servo reaches its destination before you try to send it another command.

Building a Sweeping Distance Sensor

Now, you will combine your new servo skills with your knowledge from the past few chapters to build a light-up sweeping distance sensor. The system consists of an infrared (IR) distance sensor mounted on a servo motor and four LEDs. As the servo motor cycles, it pans the distance sensor around the room, allowing you to roughly determine where objects are close and where they are far. The four LEDs correspond to four quadrants of the sweep and change brightness depending on how close an object is in that quadrant.

Because IR light is a part of the electromagnetic spectrum that humans cannot see, a system like this can be implemented to create “night vision.” The IR distance sensor works by shining an IR LED and using some fairly complex circuitry to calculate the angle at which that IR light returns to a photo sensor mounted next to the IR LED. Using analog voltages created by the IR photo sensor readings, the distance is calculated and converted to an analog voltage signal that you can read into the microcontroller. Even if the room is dark and you cannot see how close an object is, this sensor can because it is using a wavelength of light that the human eye cannot detect.

Different models of IR rangefinders may have different interfaces. If you're using a rangefinder that is different than the one used in this example, check the datasheet to make sure it sends out a variable voltage as an output.

NOTE You can watch a demo video of the sweeping distance sensor online, at exploringarduino.com/content2/ch5. You can also find this video on the Wiley website mentioned at the beginning of this chapter.

Start by hot-gluing your distance sensor to the top of a servo motor, as shown in Figure 5-4. I like to use hot glue because it holds well and is easy to remove if you need to. However, you could also use super glue, putty, or tape to get the job done.

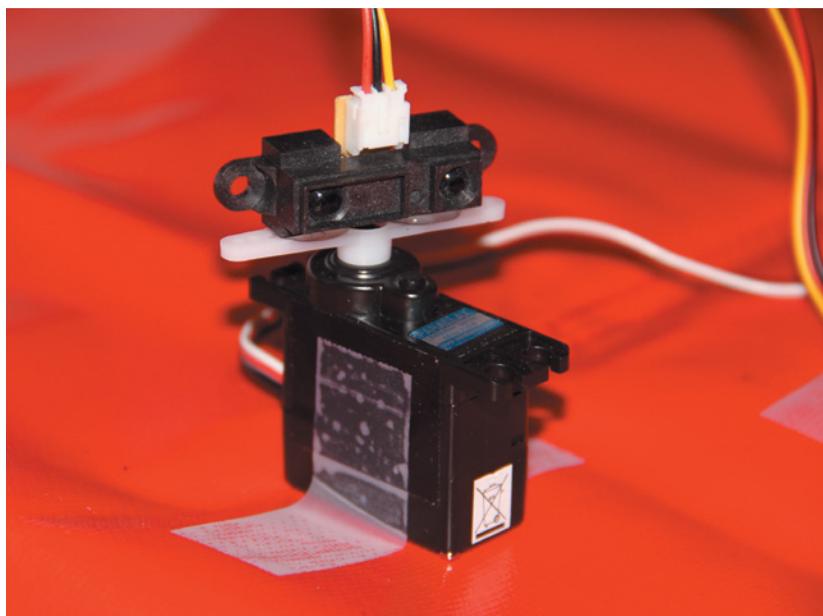


Figure 5-4: IR distance sensor mounted to the servo

Next, hook your servo up to your Arduino, using the 5V regulator to power it, just as you did before. The IR distance sensor replaces the potentiometer and plugs into analog pin 0. Four LEDs plug into pins 3, 5, 6, and 11 through $1k\Omega$ resistors. The Arduino Uno has a total of six PWM pins, but pins 9 and 10 cannot create PWM signals (using `analogWrite`) when you are using the servo library. This is because the servo library uses the same hardware timer as the one used to control PWM on those two pins. Hence, the other four PWM pins were chosen. (If you want to do this project with more LEDs, you can either use the Arduino Mega or implement a software PWM solution, something this book does not cover.) Follow the wiring diagram in Figure 5-5 to confirm that you have everything wired up correctly. I chose to use blue LEDs, but you can use any color you want. If your distance sensor wires are not connectorized, you should strip some insulation off the ends of the wires, twist them, and insert them into the breadboard and Arduino. After you have it all wired up, consider taping it down, as shown in Figure 5-4.

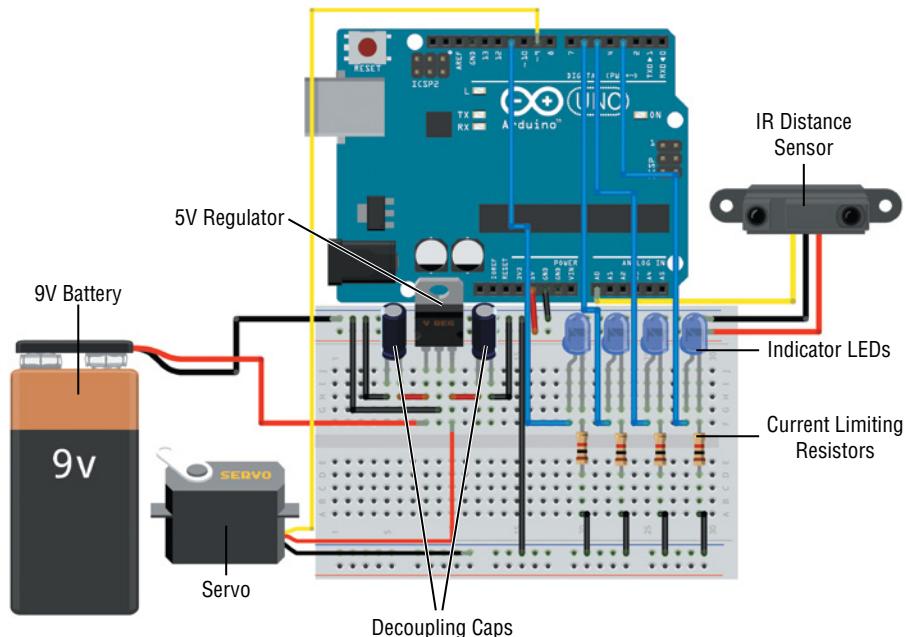


Figure 5-5: Sweeping distance sensor wiring diagram

Created with Fritzing

The last step is to program the sensor. The system works in the following manner: rotate to a given position, measure the distance, convert it to a value that can be used for the LED, change that LED's brightness, move to the next position, and so on. Listing 5-2 shows the code to accomplish this. Copy it into a new sketch and upload it to your Arduino.

Listing 5-2

Sweeping distance sensor—sweep.ino

```
//Sweeping Distance Sensor
#include <Servo.h>

const int SERVO = 9; //Servo on Pin 9
const int IR = 0; //IR Distance Sensor on Analog Pin 0
const int LED1 = 3; //LED Output 1
const int LED2 = 5; //LED Output 2
const int LED3 = 6; //LED Output 3
const int LED4 = 11; //LED Output 4
```

```

Servo myServo;      //Servo Object
int dist1 = 0;      //Quadrant 1 Distance
int dist2 = 0;      //Quadrant 2 Distance
int dist3 = 0;      //Quadrant 3 Distance
int dist4 = 0;      //Quadrant 4 Distance

void setup()
{
    myServo.attach(SERVO); //Attach the Servo
    pinMode(LED1, OUTPUT); //Set LED to Output
    pinMode(LED2, OUTPUT); //Set LED to Output
    pinMode(LED3, OUTPUT); //Set LED to Output
    pinMode(LED4, OUTPUT); //Set LED to Output
}

void loop()
{
    //Sweep the Servo into 4 regions and change the LEDs
    dist1 = readDistance(15);      //Measure IR Distance at 15 degrees
    analogWrite(LED1, dist1);     //Adjust LED Brightness
    delay(300);                  //delay before next measurement

    dist2 = readDistance(65);      //Measure IR Distance at 65 degrees
    analogWrite(LED2, dist2);     //Adjust LED Brightness
    delay(300);                  //delay before next measurement

    dist3 = readDistance(115);     //Measure IR Distance at 115 degrees
    analogWrite(LED3, dist3);     //Adjust LED Brightness
    delay(300);                  //delay before next measurement

    dist4 = readDistance(165);     //Measure IR Distance at 165 degrees
    analogWrite(LED4, dist4);     //Adjust LED Brightness
    delay(300);                  //delay before next measurement
}

int readDistance(int pos)
{
    myServo.write(pos);          //Move to given position
    delay(600);                 //Wait for Servo to move
    int dist = analogRead(IR);   //Read IR Sensor
    dist = map(dist, 50, 500, 0, 255); //scale it to LED range
    dist = constrain(dist, 0, 255); //Constrain it
    return dist;                //Return scaled distance
}

```

The program employs a simple function that rotates the servo to the requested degree, takes the distance measurement, scales it, and then returns it to the loop().

Which map you choose for the LED range depends on the setup of your system. I found that the closest object I wanted to detect registered around 500, and the farthest object was around 50, so the `map()` was set accordingly. `loop()` executes this function for each of the four LEDs, then repeats. When complete, your system should function similarly to the one shown in the demo video listed at the beginning of this section.

Understanding and Driving Stepper Motors

I could easily write an entire book about the intricacies of choosing, building, driving, and integrating stepper motors. However, there are a lot of things to learn, so this book will only focus on driving bipolar four-wire stepper motors. Stepper motors are extremely versatile *brushless* DC motors that work by energizing coils of wire in “phases” around a central, rotating permanent magnet. As these phases are turned on and off in succession, a changing magnetic field is generated that “pulls” the central permanent magnet with it as it moves.

A stepper motor moves one “step” at a time; the distance of the step is highly repeatable and defined by the electromechanical characteristics of the motor—the number of coils/phases, the design of the rotor magnet, and so on. As a result, stepper motors are excellent for tasks where accurate positioning is important. They also have high torque at low speed, which is a major advantage over brushed DC motors. You’ll often find them in robots, industrial automation systems, 3D (and 2D) printers, CNC (computer numerical control) gantries, and instrument panels.

I am the Director of Engineering and the lead electrical engineer at Shaper Tools (shapertools.com), where we use stepper motors in our *Origin* handheld power tool to enable responsive, real-time CNC positioning.

Figures 5-6a and 5-6b show the NEMA-17 bipolar stepper motor that you’ll be using shortly. Unipolar motors only energize each phase with one direction of current flow. This makes it slightly easier to design drivers for them, but it means that you can only ever get half of their conceivable drive torque! On the other hand, bipolar motors (like the NEMA-17 stepper motor shown here) energize each phase in both orientations, resulting in twice as much torque as unipolar configurations. This necessitates the use of an H-bridge, but you’re already an expert on those from the last chapter.

NOTE NEMA-17 only defines the mounting template (size) of the stepper motor, not the drive characteristics of the actual motor. NEMA-17 motors are available in a huge array of power and torque ratings.

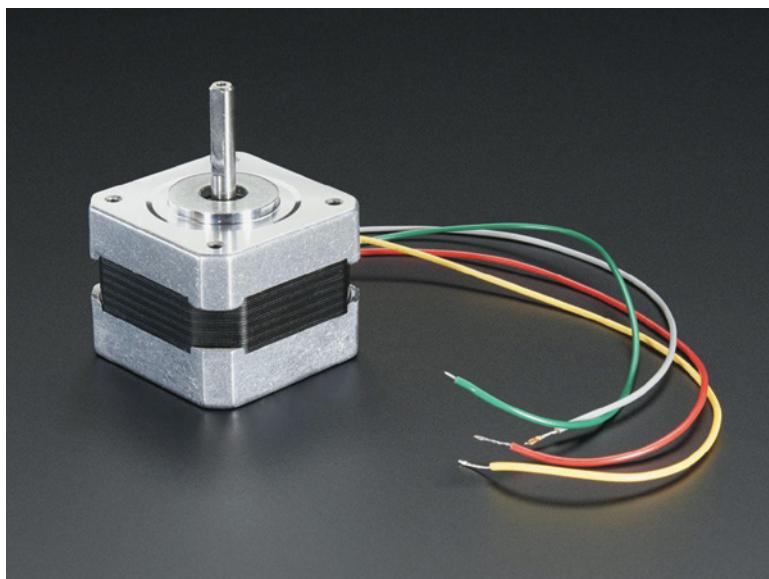


Figure 5-6a: NEMA-17 Stepper Motor (Outside)

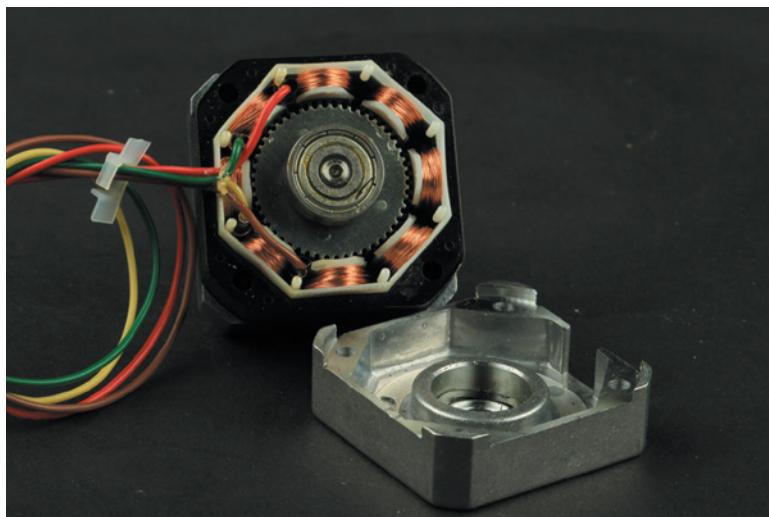


Figure 5-6b: NEMA-17 Stepper Motor (Inside)

Credit: Adafruit, adafruit.com

How Bipolar Stepper Motors Work

Bipolar stepper motors are a popular choice for getting maximal torque in a convenient form factor. They employ two *phases*, each made of multiple coils wired together. Each of these copper wire coils is wrapped around a soft metal core, creating a small electromagnet that generates a magnetic field when the current flows through the copper wire. These coils are then placed in a circular pattern around the rotating, permanent magnet core (Figure 5-6b shows eight coils). The coils attached to each phase are alternated around the perimeter of the motor. Firing them in sequence pulls the magnetized central core around in a circle.

Figure 5-7 shows a simplified illustration of how a bipolar stepper motor works. The four wires coming out of your NEMA-17 stepper motor would be connected to the four copper wires exiting from the end of each electromagnetic coil in the figure.

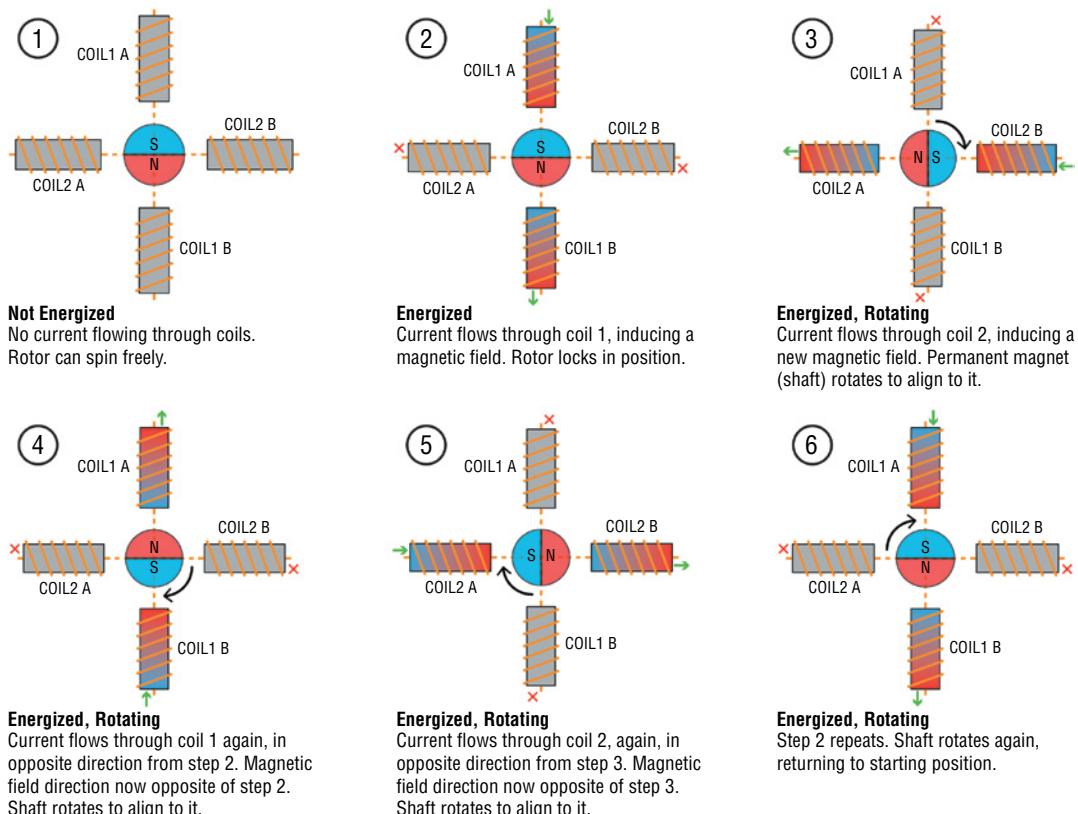


Figure 5-7: Stepper motor movement flow chart

1. In the first step, there is no current flowing through any of the coils. As a result, no magnetic fields are generated and the central magnet is not acted on by any magnetic force. It can be turned freely.
2. The first phase is energized, with current flowing into COIL 1 A and out of COIL 1 B. This current flowing through the coil around the soft metal cores generates a magnetic field that locks the central permanent magnet in place.
3. COIL 1 is turned off. COIL 2 is turned on, with current flowing from side B to A. This results in a magnetic field oriented 90 degrees clockwise from the one generated by COIL 1. The center permanent magnet is attracted to it (opposite magnetic poles always attract each other), and it rotates to it as a result.
4. COIL 2 is turned off. COIL 1 is energized again. However, this time, current is flowing in the opposite direction through the coils (this is facilitated by driving the coil with an H-bridge). The opposing direction of current flow means the magnetic field now points in a direction opposite from step 2. The center magnet rotates to match it.
5. COIL 1 is turned off. COIL 2 is turned back on, opposite from the orientation that was used in step 3. The center magnet rotates to match it.
6. The process repeats, with COIL 1 now energizing in the original current flow direction.

NOTE A lesson on the intricacies of classical electromagnetism is out of the scope of this book. However, if you want to learn more about why running a current through a coil around a metal core generates a magnetic field, search online for “Maxwell Equations” and “Ampere’s Law.”

HOW REAL STEPPERS COMPARE WITH THE SIMPLIFIED EXAMPLE

Figure 5-7 shows a simplified example of how a bipolar stepper works. In this example, there are only two coils for each motor phase (instead of the four shown in Figure 5-6b). Furthermore, this example shows a simple central magnet with only one north and one south pole. This example motor would only have a total of four steps per rotation! Your NEMA-17 motor achieves 200 steps per rotation by having more coils and a central magnet that has many alternating north and south poles at each of the “bumps” that you can see in Figure 5-6b. As a result, that motor will only move a small amount with each coil energizing. The example shown in Figure 5-7 will move a full 90 degrees when each sequential coil energizes.

Making Your Stepper Move

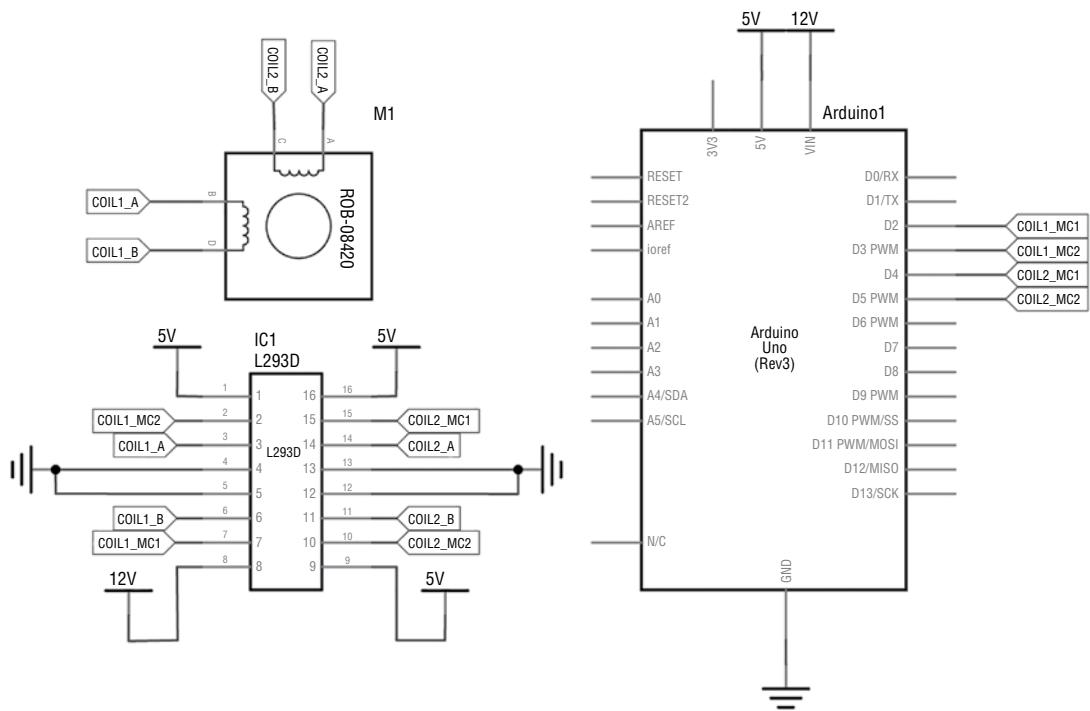
Now that you understand what's happening *inside* a stepper motor, you can build the electronics necessary to drive one. You may have noticed in the previous descriptions that a bipolar stepper motor is driven just like two brushed DC motors that you want to drive bidirectionally. The only difference is that instead of the coils being in two different motors, they are both inside your stepper motor. Each coil needs to be driven in two directions as described in Figure 5-7. This can be accomplished using the same H-bridge circuit that you used in the last chapter to drive the two DC motors on your roving robot.

There are two important deviations from your H-bridge circuit that you used in the last chapter. First, your stepper motor is probably designed to be driven at a voltage higher than 5V (check the datasheet to be sure). If you've ordered the recommended NEMA-17 motor from Adafruit, then it should be driven at 12V. I recommend using a 12V DC wall adapter for this purpose. (Steppers use a lot of power and will burn through batteries quickly.) Simply plug the DC wall adapter into the barrel jack of the Arduino. The Arduino's onboard voltage regulator will generate a 5V rail to power the microcontroller, as well as the logic power input of the H-bridge. The VIN pin of the Arduino can be used to deliver the 12V power from the wall adapter directly to the motor voltage pin of the H-bridge (pin 8).

The second change from the last chapter is that you can directly attach the enable pins to logic HIGH (5V) (this will leave the driver always enabled). The stepper library that you'll use will ensure that the H-bridge switches are not engaged in a way that can cause a short. Use the schematic in Figure 5-8 to wire up the H-bridge driver to your Arduino.

Were you able to wire it up using only the schematic? If you're not sure which wire from the stepper motor belongs to which phase, consult the datasheet or the website where you bought it. If you still can't figure it out, you can use a multimeter to quickly determine the correct wires. Put your multimeter in ohmmeter or continuity testing mode. Pick any two wires from the stepper motor. If those two wires have a low resistance (<10 ohms), then they are two wires from the same phase. Repeat this as necessary until you've found the two phase pairs. In each phase, it doesn't matter which side of the phase you put in which pin (as long as they are both on the same bridge of the H-bridge chip). Reversing them will only reverse the default rotation direction of the motor.

Consult Figure 5-9 to confirm that you've properly wired your stepper motor to your Arduino.

**Figure 5-8:** Stepper motor wiring schematic

Created with Fritzing

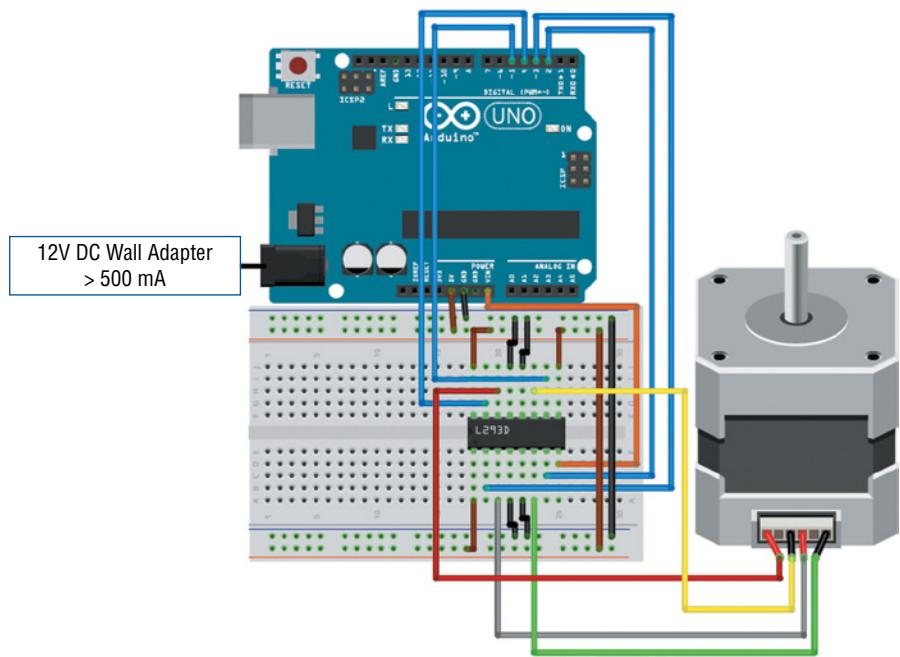


Figure 5-9: Stepper motor wiring diagram

Created with Fritzing

Now that your stepper is wired up, it's time to make it move! You'll start with a simple back-and-forth sweep that illustrates how the stepper library works. Just like with the servo library, you can import the stepper library to your sketch to enable easy control of stepper motors. Copy the code in Listing 5-3 into a sketch to load onto your Arduino. Before actually loading it on, make sure the 12V power is connected to your Arduino's barrel jack. Otherwise, the stepper won't move correctly when the sketch starts.

Listing 5-3

Simple stepper control-stepper.ino

```
//Simple Stepper Control with an H-Bridge  
  
#include <Stepper.h>  
  
//Motor Constants  
//Most NEMA-17 Motors have 200 steps/revolution  
const int STEPS_PER_REV = 200; //200 steps/rev
```

```
//H-Bridge Pins
const int COIL1_MC1 = 2; //COIL 1 Switch 1 Control
const int COIL1_MC2 = 3; //COIL 1 Switch 2 Control
const int COIL2_MC1 = 4; //COIL 2 Switch 1 Control
const int COIL2_MC2 = 5; //COIL 2 Switch 2 Control

// Initialize the stepper library - pass it the Switch control pins
Stepper myStepper(STEPS_PER_REV, COIL1_MC1, COIL1_MC2, COIL2_MC1, COIL2_MC2);

void setup()
{
    //Set the stepper speed
    myStepper.setSpeed(60); // 60 RPM
}

void loop()
{
    // step one revolution in one direction:
    myStepper.step(STEPS_PER_REV);
    delay(500);

    // step one revolution in the other direction:
    myStepper.step(-STEPS_PER_REV);
    delay(500);
}
```

Take a moment to understand this code. The `#include <Stepper.h>` statement imports the stepper motor Arduino library. Next, a constant representing the number of steps in one full rotation is created for easy reference later in the sketch. All the H-bridge control pins are assigned accordingly. `Stepper myStepper()` creates a stepper motor object called `myStepper`. You can change `myStepper` to any name you want, if you reference it in place of `myStepper` later in the sketch. This object constructor takes five arguments: the number of steps in a full revolution, and the four Arduino pins connected to the H-bridge controller. In the `setup()` function, the `myStepper` object is set up with a default speed in rotations per minute (RPM). When calling `myStepper.step()` later in the program, the library will take care of driving the motor at that speed for the specified number of steps. In the `loop()`, `myStepper.step()` takes just one argument that tells the stepper library to move that stepper the specified number of steps at the previously defined speed. The `step()` function is “blocking,” meaning that the next command will not execute until the stepper has finished the requested movement. In this program, the stepper should do one full rotation forward, followed by one full rotation backward. It will repeat this forever.

NOTE Is your motor just wiggling or not rotating? Check that the 12V supply is plugged in and properly connected to the motor voltage pin of the H-bridge chip. If it's not plugged in, but the USB cable is, you'll be feeding your stepper insufficient voltage and it won't move properly. If that's not the issue, then check the phase wiring; the wires from the same phase should be on the same side of the H-bridge chip.

Building a “One-Minute Chronograph”

As you learned at the start of this chapter, stepper motors can be found in a wide range of products. For instance, you'll often find them in the analog dials of a car or an airplane's instrument panel. Along the same lines, you'll use your new knowledge of stepper motors to make an accurate chronograph, capable of running an incrementing timer for a precise amount of time. Stepper motors are well suited to this project because they repeatedly move a fixed amount with each step. Knowing the number of steps in a full rotation, you can time their movement to ensure they complete one full rotation in exactly the desired amount of time. Doing that with a brushed DC motor would be impossible without some sort of feedback mechanism to report position.

Wiring and Building the Chronograph

Your chronograph will need start and stop buttons, so add those to your existing stepper motor drive circuit, as shown in Figure 5-10.

Is something missing in this diagram? Where are the pull-up resistors for the buttons? In this project, you'll learn how to use an often underutilized feature of the Arduino (and most microcontrollers). The ATmega microcontroller at the heart of the Arduino has configurable I/O pin modes; you already know this because you've learned how to use `pinMode()` to switch them between INPUT and OUTPUT. However, there are actually additional settings available for these pins. Notably, you can set pins to INPUT_PULLUP mode. Setting a pin to this mode will make it an input, and enable a pull-up resistor inside the chip itself! This pull-up generally has a value somewhere between $20\text{K}\Omega$ and $150\text{K}\Omega$, depending on the exact board that you are using—consult the datasheet to be sure. Configuring a pin in this mode saves you from having to use an external resistor. Just wire the button so it shorts to GND when pressed, and that's the entire circuit. The code you'll use shortly enables the internal pull-up in the `setup()` function.

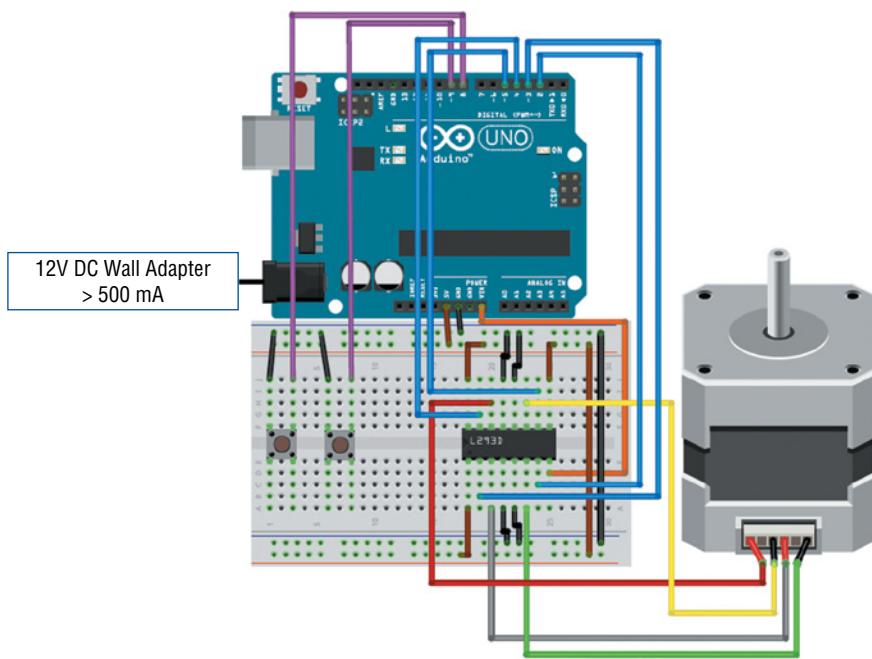


Figure 5-10: Chronograph wiring diagram

Created with Fritzing

NOTE Using an internal pull-up works great for things like buttons, but it may not be the best solution in all scenarios. The pull-up will not be activated until the Arduino has executed the bootloader and started running your code. Thus, if that pin is connected to some other integrated circuit that cannot have that pin in a floating state (even for a few seconds), then you'll want to use a hardware pull-up resistor.

Now that you have your circuit built, you can construct your actual chronograph face and hand. Reference Figure 5-11 for a simple example of how to do this. I hot-glued a blank CD to the face of the stepper (be sure not to get glue into the part that rotates). I slid a Popsicle stick into a binder clip and clipped it onto the rotating motor shaft. I then marked the seconds on the face of the “clock.” While the motor is unpowered, you can manually turn the shaft to the “0” position so that the chronograph starts at the correct point.

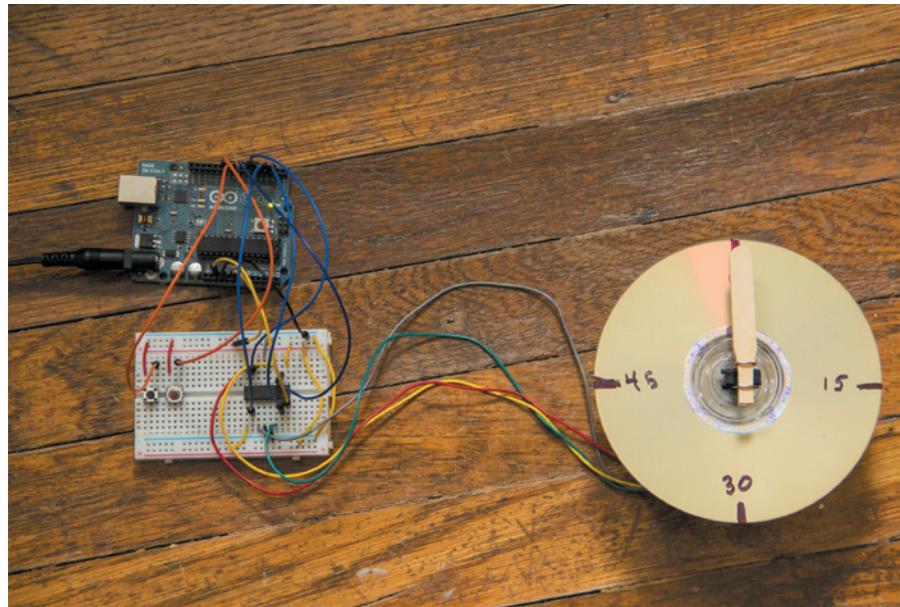


Figure 5-11: One-minute chronograph project

Programming the Chronograph

Writing the software for the chronograph will use much of what you've already learned. Start with your `stepper.ino` as a baseline, and add to it. You'll want to keep the parts at the top: the `#include` statement, the `STEPS_PER_REV` definition, and the pin definitions. In addition to that, add definitions for the START and STOP buttons (connected to pins 8 and 9, respectively). Next, you'll also want to define the number of milliseconds required between each step. You want the chronograph to complete one full rotation in exactly 60 seconds. Given that 60 seconds is 60,000 milliseconds, you can divide 60,000 ms by 200 steps to get 300 milliseconds per step. So, you'll be instructing the stepper to advance one step every 300 milliseconds. With all that, the top of your program should look something like this:

```
#include <Stepper.h>

//Most NEMA-17 Motors have 200 steps/revolution
const int STEPS_PER_REV = 200; //200 steps/rev
```

```

//To do one rotation in a minute,
//we need to know the milliseconds required between steps:
//60 seconds * 1000ms / 200 steps = 300 ms/step
const int MS_PER_STEP = 300;

//H-Bridge PinsW
const int COIL1_MC1 = 2; //COIL 1 Switch 1 Control
const int COIL1_MC2 = 3; //COIL 1 Switch 2 Control
const int COIL2_MC1 = 4; //COIL 2 Switch 1 Control
const int COIL2_MC2 = 5; //COIL 2 Switch 2 Control

//Button Pins
const int START = 8; //Start Button
const int STOP = 9; //Stop Button

//Initialize the stepper library - pass it the Switch control pins
Stepper chronograph(STEPS_PER_REV, COIL1_MC1, COIL1_MC2,
COIL2_MC1, COIL2_MC2);

```

That should look familiar. Note that the Stepper object is now called `chronograph` instead of `myStepper`. In order to keep track of elapsed time, you'll use the Arduino language's `millis()` function to ensure you step once every 300 ms. The `millis()` function takes no arguments, and just returns the amount of time in milliseconds since the Arduino started running code. Thus, if you keep track of the value returned the last time it was run, you can easily tell when 300 ms has elapsed. To do this, you use global variables defined at the top of the code where you can store the times returned by `millis()`. You'll also need to use a global variable to keep track of how many steps have been taken, so you can stop when you get to 200, or so you can reset the timer to the correct location based on how far it has travelled. Initialize these global variables anywhere at the top of your file (above the `setup()` function):

```

//Tracking Variables
unsigned long last_time = 0;
unsigned long curr_time = 0;
int steps_taken = 0;

```

The time-tracking variables are `unsigned longs` because that is the variable type that can hold the largest positive number in the Arduino language; as you might imagine, the value returned by `millis()` can get quite large if the Arduino has been running for a long time.

In the `setup()` function, you should set the default speed for the stepper motor, as you did in your last program. The exact value isn't that important because you'll only be moving it one step at time (you're setting the pace at 300 milliseconds per step).

However, when you reset your chronograph, this value will control how quickly it returns to the start position. Thus, I recommend something fast—somewhere between 50 and 200 RPM. (The stepper will likely not be able to keep up if you try to go faster than that.) Also, don't forget to enable the pull-up resistors on your button inputs in the setup function:

```
void setup()
{
    //Set the stepper speed high so each "tick" is fast
    chronograph.setSpeed(200); //200 RPM

    //Setup Pullups on Buttons
    pinMode(START, INPUT_PULLUP);
    pinMode(STOP, INPUT_PULLUP);
}
```

Finally, you're ready to write the main loop. Here's the general flow:

1. Wait until the START button is pressed. This can be accomplished by using a one-line `while()` loop with a “;” at the end. When a `while()` loop has no contents, the Arduino will just endlessly check its conditions. As long as the conditions in the loop definition are met, it won't move on to the next line of code. The loop should be checking for the START button to be pressed. You're only waiting for it to go `LOW`, so you don't need to debounce it.
2. Once the START button has been pressed, get the current time from `millis()` and save it to the `last_time` variable so you can compare against it in a future step.
3. Enter a `while()` loop that will keep going until one minute has elapsed, or until the STOP button state has changed (been pressed).
 - Get the current time with `millis()` and compare it to `last_time`. Once the difference between the two has reached 300 ms, it's time to move the stepper motor by one step.
 - Increment the step tracking variable, set `last_time` to the current time (so you can repeat this loop), and step the motor by one step.
4. If the code is at this step, then it means a full minute has elapsed, or the STOP button was pressed. If the STOP button was pressed, then return the dial to the starting position. Reset the step counter to zero so the process can begin again.

Were you able to write all the logic for that on your own? Give it a shot and try to debug it yourself, before reading through the completed code example that follows.

When you're ready, compare what you've written to the program in Listing 5-4, and load it onto your Arduino. Remember to plug in the 12V wall adapter!

Listing 5-4

One-minute chronograph project-chronograph.ino

```
//One Minute Chronograph with Start/Stop/Reset

#include <Stepper.h>

//Most NEMA-17 Motors have 200 steps/revolution
const int STEPS_PER_REV = 200; //200 steps/rev

//To do one rotation in a minute,
//we need to know the milliseconds required between steps:
//60 seconds * 1000ms /200 steps = 300 ms/step
const int MS_PER_STEP = 300;

//H-Bridge Pins
const int COIL1_MC1 = 2; //COIL 1 Switch 1 Control
const int COIL1_MC2 = 3; //COIL 1 Switch 2 Control
const int COIL2_MC1 = 4; //COIL 2 Switch 1 Control
const int COIL2_MC2 = 5; //COIL 2 Switch 2 Control

//Button Pins
const int START = 8; //Start Button
const int STOP = 9; //Stop Button

//Tracking Variables
unsigned long last_time = 0;
unsigned long curr_time = 0;
int steps_taken = 0;

//Initialize the stepper library - pass it the Switch control pins
Stepper chronograph(STEPS_PER_REV, COIL1_MC1, COIL1_MC2, COIL2_MC1, COIL2_MC2);

void setup()
{
    //Set the stepper speed high so each "tick" is fast
    chronograph.setSpeed(200); //200 RPM

    //Setup Pullups on Buttons
    pinMode(START, INPUT_PULLUP);
    pinMode(STOP, INPUT_PULLUP);
}
```

```
void loop()
{
    //Endless Loop - wait here until start is pressed
    //The Semicolon after the while loop definitions keeps us
    //here until the condition is no longer met
    while(digitalRead(START) == HIGH);

    last_time = millis(); //Get the time when we started

    //Keep Going in this loop until stopped, or minute has elapsed
    while(digitalRead(STOP) == HIGH && steps_taken < STEPS_PER_REV)
    {
        curr_time = millis();

        //If enough time has passed, go one step
        if(curr_time - last_time >= MS_PER_STEP)
        {
            chronograph.step(1); //Move one step
            steps_taken++; //Increment the steps_taken variable
            last_time=curr_time; //Set the last time equal to the current time
        }
    }

    //If we get here, the stop button has been pressed or a minute elapsed.
    //If we didn't go the full rotation, return to start
    if (steps_taken < STEPS_PER_REV) chronograph.step(-steps_taken);
    //Reset the step tracker
    steps_taken = 0;
}
```

Your chronograph should now be fully functional! Press the START button to start timing. The dial will advance through 360 degrees of rotation in exactly one minute. If you press the STOP button, the chronograph will reset itself back to the start position. Try experimenting with different total times. Can you make a two-minute chronograph? Can you turn it into a timer that counts down instead of up? What about making a simple lap timer?

NOTE You can watch a demo video of the One-Minute Chronograph project online, at exploringarduino.com/content2/ch5. You can also find this video on the Wiley website shown at the beginning of this chapter.

Summary

In this chapter, you learned the following:

- Servo motors enable precise positioning and can be controlled using the Arduino servo library.
- IR distance sensors return analog values representing distances detected by bouncing infrared light off objects.
- Code commenting is critical for easing debugging and sharing.
- The Arduino has internal pull-up resistors than can be enabled on input pins.
- Stepper motors take advantage of electromagnetism to precisely step through positions.
- You can use the `millis()` function to track elapsed time in your Arduino sketches.

6

Making Sounds and Music

Parts You'll Need for This Chapter

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

Half-size or full-size breadboard

Assorted jumper wires

Pushbuttons ($\times 5$)

220Ω resistor

$10k\Omega$ resistors ($\times 5$)

$10k\Omega$ potentiometer

8Ω loudspeaker

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch6

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

Humans have five senses. As you might have guessed, you won't be interfacing your sense of taste with too many electronics; licking your Arduino is a bad idea. Similarly, smell won't generally come into play. In fact, if you can smell your electronics, something is probably burning (and you should stop what you're doing). That just leaves the senses of touch, sight, and sound. You've already interfaced with potentiometers and buttons that take advantage of your sense of touch, and you've hooked up LEDs

that interface with your sense of sight. Now, what about your auditory senses? This chapter focuses on using the Arduino to make sounds so that you can more easily gather feedback from your projects.

You can generate sound with an Arduino in a number of ways. The simplest method is to use the `tone()` function, which this chapter focuses on most heavily. However, you can also use various shields that add more complex, music-playing capabilities to your Arduino with the help of some external processing. (Shields are add-on boards that attach to the top of your Arduino to add specific functionality.) If you own the Arduino Due, you can use its true digital-to-analog converter (DAC) to produce sounds.

Understanding How Speakers Work

Before you can make sounds with your Arduino, you need to understand what sounds are and how humans perceive them. In this first section, you will learn about how sound waves are generated, their properties, and how manipulation of those properties can produce music, voices, and so on.

The Properties of Sound

Sound is transmitted through the air as a pressure wave. As an object such as a speaker, a drum, or a bell vibrates, that object also vibrates the air around it. As the air particles vibrate, they transfer energy to the particles around them, vibrating these particles as well. In this fashion, a pressure wave is transferred from the source to your eardrum, by creating a chain reaction of vibrating particles. So, why do you need to know this to understand how to make sounds with your Arduino?

You can control two properties of these vibrating particles with your Arduino: frequency and amplitude. The *frequency* represents how quickly the air particles vibrate back and forth, and the *amplitude* represents the magnitude of their vibrations. In the physical sense, higher amplitude sounds are louder, and lower amplitude sounds are quieter. High-frequency sounds are a higher pitch (like a soprano), and low-frequency sounds are a lower pitch (like a bass). Consider the diagram in Figure 6-1, which shows sinusoidal representations of sound waves of various amplitudes and frequencies.

Figure 6-1 shows three piano notes: low, middle, and soprano C. Each graph shows the given frequencies at both low and high amplitudes. As an example, to understand frequency and amplitude, focus on middle C. Middle C has a frequency of 261.63 Hertz (Hz). In other words, a speaker, a guitar string, or a piano string will complete 261.63 oscillations per second to produce the middle C sound. By taking the reciprocal of that value, you can find the period of the wave, which is easy to see in Figure 6-1. The width

of one complete oscillation in the graph is represented by $1/261.63$, which equals 3.822 milliseconds. Using the Arduino, you can set that period for a square wave and thus adjust the tone of the note.

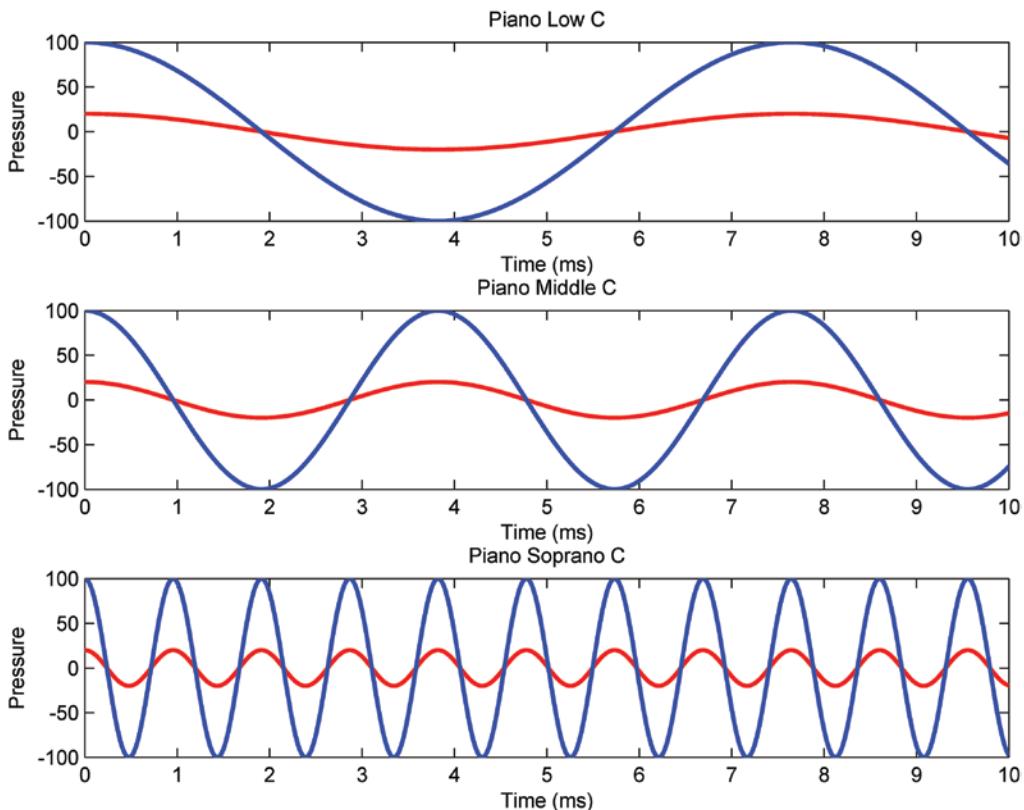


Figure 6-1: Sound waves of varying amplitudes and frequencies

Created with MATLAB

Importantly, the Arduino (excluding the Due's true DAC) cannot actually make a sinusoidal wave that you might observe in the real world. A square wave is a digital periodic wave—it also oscillates between a high and a low value, but it switches nearly instantaneously, instead of slowly like a sine wave. This still creates a pressure wave that results in sound, but it isn't quite as “pretty” sounding as a sinusoidal wave.

As for the amplitude, you can control that by changing the amount of the current permitted to flow through the speaker. Using a potentiometer in line with the speaker, you can dynamically adjust the volume level of the speaker.

How a Speaker Produces Sound

Speakers, much like the motors that you learned about in the preceding chapter, take advantage of electromagnetic forces to turn electricity into motion. Try holding a piece of metal up to the rear of your speaker. Did you notice anything interesting? The metal probably sticks to the rear of your speaker, because all speakers have a sizeable permanent magnet mounted to the back. Figure 6-2 shows a cross section of a common speaker.

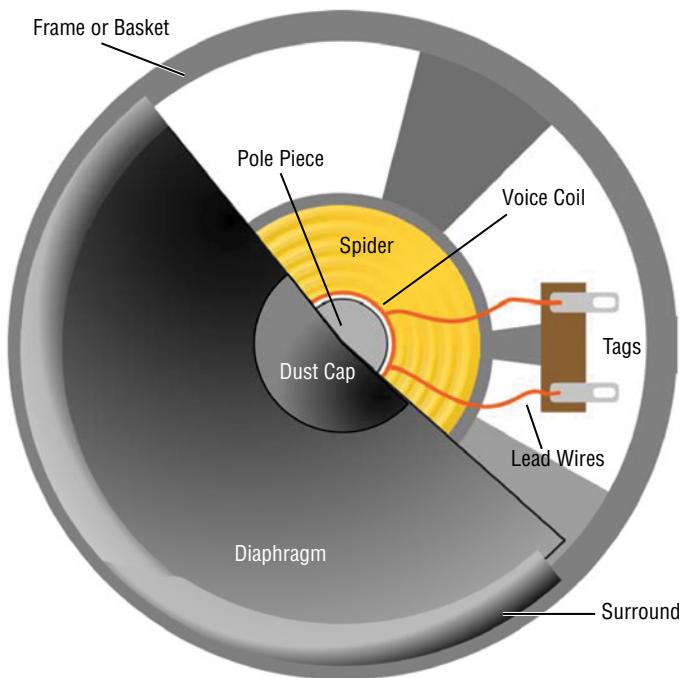


Figure 6-2: Speaker cross section

Credit: Wikipedia (GNU Free Documentation License)

The permanent magnet is mounted behind the voice coil and pole piece shown in the image. As you send a sinusoidal voltage signal (or a square wave in the case of the Arduino) into the leads of the coil, the changing current induces a magnetic field that causes the voice coil and diaphragm to vibrate up and down as the permanent magnet is attracted to and then repulsed by the magnetic field that you have generated. This back-and-forth vibration, in turn, vibrates the air in front of the speaker, effectively creating a sound wave that can travel to your eardrum.

Using `tone()` to Make Sounds

The Arduino IDE includes a built-in function for easily making sounds of arbitrary frequencies. The `tone()` function generates a square wave of the selected frequency on the output pin of your choice. The `tone()` function accepts three arguments, though the last one is optional:

- The first argument sets the pin to generate the tone on.
- The second argument sets the frequency of the tone.
- The third (optional) argument sets the duration of the tone. If the third argument is not set, the tone continues playing until you call `noTone()`.

Because `tone()` uses one of the ATmega's hardware timers, you can start a tone and do other things with your Arduino while it continues to play sound in the background.

In the following sections, you will learn how to play arbitrary sound sequences. Once you have that working, you can use `tone()` as a response to various inputs (buttons, distance sensors, accelerometers, and so on). At the end of the chapter, you will build a simple five-button piano that you can play.

Including a Definition File

When it comes to playing music, a definition file that maps frequencies to note names proves useful. This makes it more intuitive to play simple musical clips. If you are familiar with reading sheet music, you know that notes are indicated with letters representing their pitch. The Arduino IDE includes a header file that correlates each of these notes with its respective frequency. Instead of digging through the Arduino install directory to find it, just visit the Exploring Arduino Chapter 6 web page, and download the code for this chapter, including the pitch file, to your desktop (exploringarduino.com/content2/ch6). You'll place it in your sketch directory after you've created it.

Next, open your Arduino IDE and save the blank sketch that is automatically created when you open the IDE. As you've probably already noticed, when you save a sketch, it actually saves a folder with that name and places an `.ino` file inside of that folder. By adding other files to that folder, you can include them in your program, all while keeping your code better organized. Copy the `pitches.h` file you saved to the desktop into the folder created by the IDE; then close the Arduino IDE. Open your `.ino` file in the Arduino IDE, and notice the two tabs that now appear (see Figure 6-3).

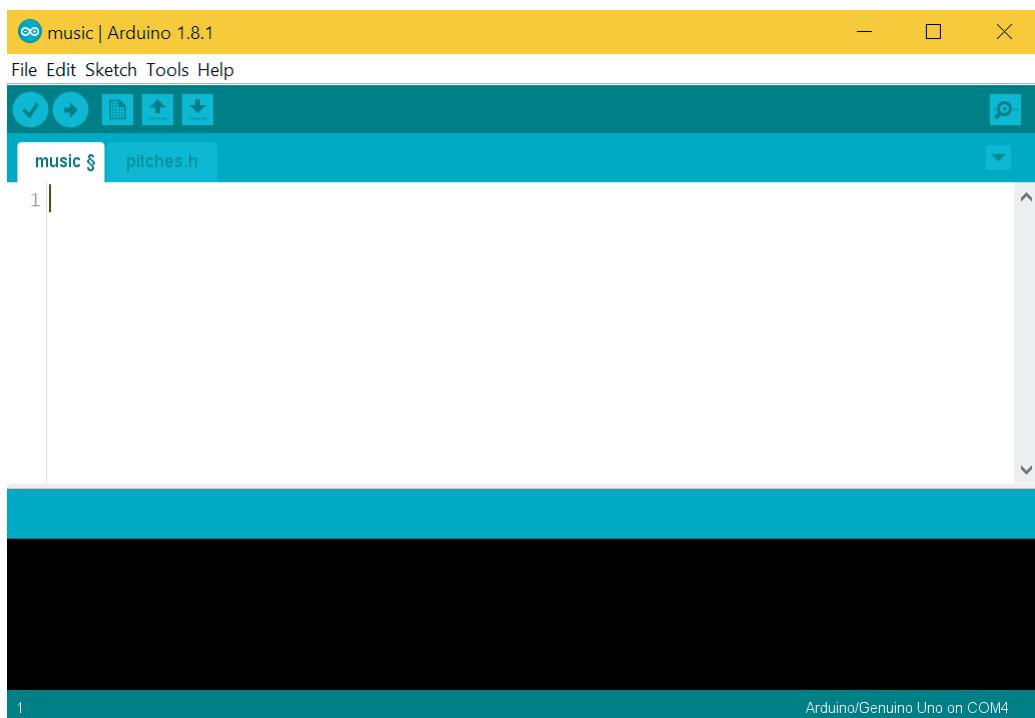


Figure 6-3: Arduino IDE with a secondary header file

Click the pitches.h tab to see the contents of the file. Notice that it's just a list of definition statements, which map human-readable names to given frequency values. Simply having the header file in the IDE does not suffice, though. To ensure that the compiler actually uses those definitions when compiling your program for the Arduino, you need to tell the compiler to look for that file. Doing so is easy. Just add this line of code to the top of your .ino file:

```
#include "pitches.h" //Header file with pitch definitions
```

To the compiler, this is essentially the same thing as copying and pasting the contents of the header file into the top of your main file. However, this keeps the file neater and easier for you to read. In the following sections, you will write the code for the rest of this file so that you can actually use the pitch definitions that you have just imported.

Wiring the Speaker

Now that you have your pitches header file included, you're ready to build a test circuit and to write a simple program that can play some music. The electrical setup is fairly

simple and just involves hooking up a speaker to an output pin of your Arduino. However, remember what you've learned in previous chapters about current-limiting resistors.

Just as with LEDs, you want to put a current-limiting resistor in series with the speaker to ensure that you don't try to draw too much current from one of the Arduino's I/O pins. As you learned previously, each I/O pin can supply only a maximum of 40 mA, so pick a resistor that prevents you from exceeding that. The speaker that I recommend has an internal resistance of 8Ω (as do most loudspeakers that you can buy); this resistance comes from the windings of wire that make up the electromagnet. If you use a speaker with a different resistance, be sure to substitute that value into the following calculations. Recall that Ohm's law states that $V = IR$. In this scenario, the I/O pin is outputting 5V, and you don't want to exceed 40 mA. Solving for R, you find that the minimum resistance must be: $R = 5V / 40 \text{ mA} = 125\Omega$. A resistance of 8Ω is already accounted for by the speaker, so your inline resistor must be at least $125\Omega - 8\Omega = 117\Omega$. The nearest common higher resistor values are 150Ω and 220Ω , so you can use either of those values. Using a 150Ω resistor will result in slightly more volume than a 220Ω resistor, but probably not enough for you to be able to discern the difference. By further adjusting the series resistance, you can change the volume of the speaker. To make this as easy as possible, you can use a potentiometer in line with the fixed-value resistor, as shown in Figure 6-4. In the schematic, R1 is the 220Ω resistor, and R2 is the potentiometer.

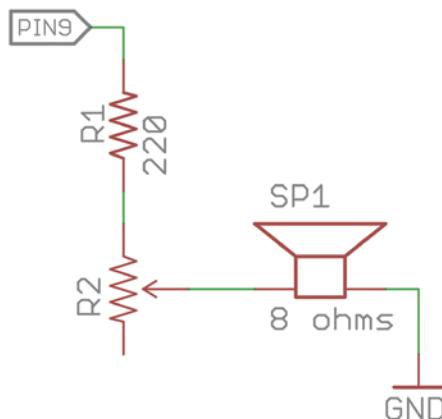


Figure 6-4: Speaker wiring with volume adjustment knob

Created with EAGLE

Note that unlike your previous uses of potentiometers, this configuration uses only two pins: the middle (or wiper) pin goes to the speaker, and either one of the end pins connects to the 220Ω resistor. When the knob is turned all the way toward the unconnected terminal, the entire resistance of the potentiometer is added to the series resistance

of the 220Ω resistor, and the volume lowers. When the knob is turned all the way toward the connected end terminal, it adds no resistance to the series, and the speaker is at maximum volume. Referencing the schematic in Figure 6-4, wire your speaker to the Arduino. Then, confirm your wiring using the diagram in Figure 6-5. If your speaker does not already have wires attached to its two terminals, you can solder wires to them. If you don't have a soldering iron handy, carefully and tightly wrapping a solid core wire through the terminal eyelet will work as well (but soldering is recommended).

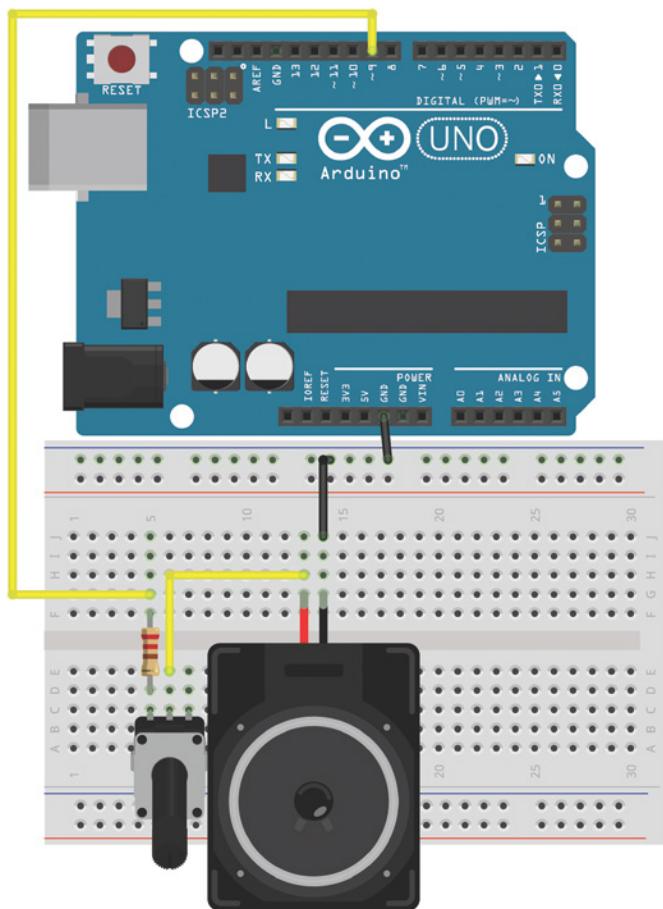


Figure 6-5: Speaker wiring diagram

Created with Fritzing

Speakers do not have a polarity; you can connect them in either direction. After wiring your speaker successfully, you're ready to make music!

Making Sound Sequences

To play back some songs, you will first learn about using arrays to store multiple values easily. You will then implement a simple loop to iterate through the arrays of notes and play them back on the speaker.

Using Arrays

An array is a sequence of values that are related in some way. By grouping them together, it is an ideal format to iterate through. You can think of an array as a numbered list. Each position has an index that indicates its location in the list, and each index has a value that you want to store. You use an array here to store the list of notes that you want to play, in the order that you want to play them.

To ensure that the Arduino's memory is properly managed, you need to declare arrays with a known length. You can do this either by explicitly specifying the number of items or by simply populating the array with all the values you are interested in. For example, if you want to make an array that will contain four integer values, you could create it like this:

```
int numbers[4];
```

You can optionally initialize the values when you declare the array. If you initialize the values, specifying the length in the brackets is optional. If unspecified, the length is assumed to equal the number of elements that you initialized:

```
//Both of these are acceptable
int numbers[4] = {-7, 0, 6, 234};
int numbers[] = {-7, 0, 6, 234};
```

Note that arrays are zero indexed. In other words, the first number is at position 0, the second is at position 1, and so forth. You can access the elements in an array at any given index by putting the index of the relevant value in a square bracket after the variable name. If you want to set the brightness of an LED connected to pin 9 to the third entry in an array, for example, you can do so like this:

```
analogWrite(9, numbers[2]);
```

Note that because numbering starts at zero, the index of 2 represents the third value in the array. If you want to change one of the values of the array, you can do so in a similar fashion:

```
numbers[2] = 10;
```

Next, you will use arrays (as shown in these examples) to create a structure that can hold the sequence of notes that you want to play on your speaker.

Making Note and Duration Arrays

To store the information about the song you want to play, you can use two arrays of the same length. The first contains the list of pitches, and the second contains the list of durations for which each note should play in milliseconds. You can then iterate through the indices of these arrays and play back your tune.

Using the meager musical skills that I've retained from my music classes back in high school, I've assembled a short and catchy tune:

```
//Note Array
int notes[] = {
    NOTE_A4, NOTE_E3, NOTE_A4, 0,
    NOTE_A4, NOTE_E3, NOTE_A4, 0,
    NOTE_E4, NOTE_D4, NOTE_C4, NOTE_B4, NOTE_A4, NOTE_B4, NOTE_C4, NOTE_D4,
    NOTE_E4, NOTE_E3, NOTE_A4, 0
};

//The Duration of each note (in ms)
int times[] = {
    250, 250, 250, 250,
    250, 250, 250, 250,
    125, 125, 125, 125, 125, 125, 125, 125,
    250, 250, 250, 250
};
```

Note that both arrays are the same length: 20 items. Notice also that some of the notes are specified as 0. These are musical rests (unplayed beats). Each note pairs with a duration from the second array. If you are familiar with music theory, you'll see that I've made quarter notes 250 ms and eighth notes 125 ms. The song is in "four-four time," in musical terms.

Try out this given note sequence first; then try to create your own!

NOTE Listen to a recording of this tune, played by an Arduino, at exploringarduino.com/content2/ch6.

Completing the Program

The last step is to actually add playback functionality to the sketch. This can be accomplished with a simple for loop that goes through each index in the array, and plays the given note for the given duration. Because you presumably don't want to listen to this over and over again, you can put the playback functionality in the setup() function so that it only happens once. You can restart playback by pressing the Reset button. Listing 6-1 shows the complete playback program.

Listing 6-1

Arduino music player-music.ino

```
//Plays a song on a speaker

#include "pitches.h" //Header file with pitch definitions

const int SPEAKER=9; //Speaker Pin

//Note Array
int notes[] = {
    NOTE_A4, NOTE_E3, NOTE_A4, 0,
    NOTE_A4, NOTE_E3, NOTE_A4, 0,
    NOTE_E4, NOTE_D4, NOTE_C4, NOTE_B4, NOTE_A4, NOTE_B4, NOTE_C4, NOTE_D4,
    NOTE_E4, NOTE_E3, NOTE_A4, 0
};

//The Duration of each note (in ms)
int times[] = {
    250, 250, 250, 250,
    250, 250, 250, 250,
    125, 125, 125, 125, 125, 125, 125, 125,
    250, 250, 250, 250
};

void setup()
{
    //Play each note for the right duration
    for (int i = 0; i < 20; i++)
    {
        tone(SPEAKER, notes[i], times[i]);
        delay(times[i]);
    }
}

void loop()
{
    //Press the reset button to play again.
}
```

If you want to make your own music, make sure that the arrays remain at an equal length and that you change the upper bound on the `for()` loop. Because the `tone()` function can run in the background, it's important to use the `delay()` function. By delaying the code for an amount of time equal to the duration of the note, you ensure that the Arduino doesn't play the next note until the previous note has finished playing for the time you specified.

Understanding the Limitations of the `tone()` Function

The `tone()` function does have a few limitations to be aware of. Like the servo library, `tone()` relies on a hardware timer that is also used by the board's pulse-width modulation (PWM) functionality. If you use `tone()`, PWM does not work correctly on pins 3 and 11 (on boards other than the Mega).

Also remember that the Arduino I/O pins are not digital-to-analog converters (DACs). Hence, they output only a square wave at the provided frequency, not a sine wave. Although this suffices for making tones with a speaker, you'll find it undesirable for playing back music. If you want to play back WAV files, your options include using a music-playing shield (such as the Adafruit Wave Shield or the SparkFun MP3 shield), implementing a DAC converter, or using the built-in DAC available on the Arduino Due using the Due-only audio library.

The final limitation is that you can use the `tone()` function on only one pin at a time, so it isn't ideal for driving multiple speakers. If you want to drive multiple speakers at the same time from a standard Arduino, you have to use manual timer interrupt control, something you will learn more about in Chapter 13, "Interrupts and Other Special Functions."

NOTE To read a tutorial on advanced multi-speaker control with an Arduino, visit blum.fyi/five-speakers.

Building a Micro Piano

Playing back sequences of notes is great for adding audio feedback to projects you've already created. For example, consider replacing or augmenting a green confirmation LED with a confirmation sound. But, what if you want to dynamically control the sound? To wrap up this chapter, you will build a simple pentatonic piano. The pentatonic scale consists of just five notes per octave rather than the usual seven. Interestingly, the notes of a pentatonic scale have minimal dissonance between pitches, meaning they always sound good together. So, it makes a lot of sense to use pentatonic notes to make a simple piano.

NOTE The SudoGlove, among others things, is a control glove that can synthesize music using the pentatonic scale. You can learn more about it at sudoglove.com.

To make your Arduino piano, you use this pentatonic scale: C, D, E, G, A. You can choose which octave to use based on your preference. I chose to use the fourth octave from the header file.

First, wire five buttons up to your Arduino. As with the buttons in Chapter 2, “Digital Inputs, Outputs, and Pulse-Width Modulation,” you use $10\text{k}\Omega$ pull-down resistors with the buttons. In this scenario, you do not need to debounce the buttons because the note will be played only while the desired button is held down. Wire the buttons as shown in Figure 6-6 and keep the speaker wired as you had it previously.

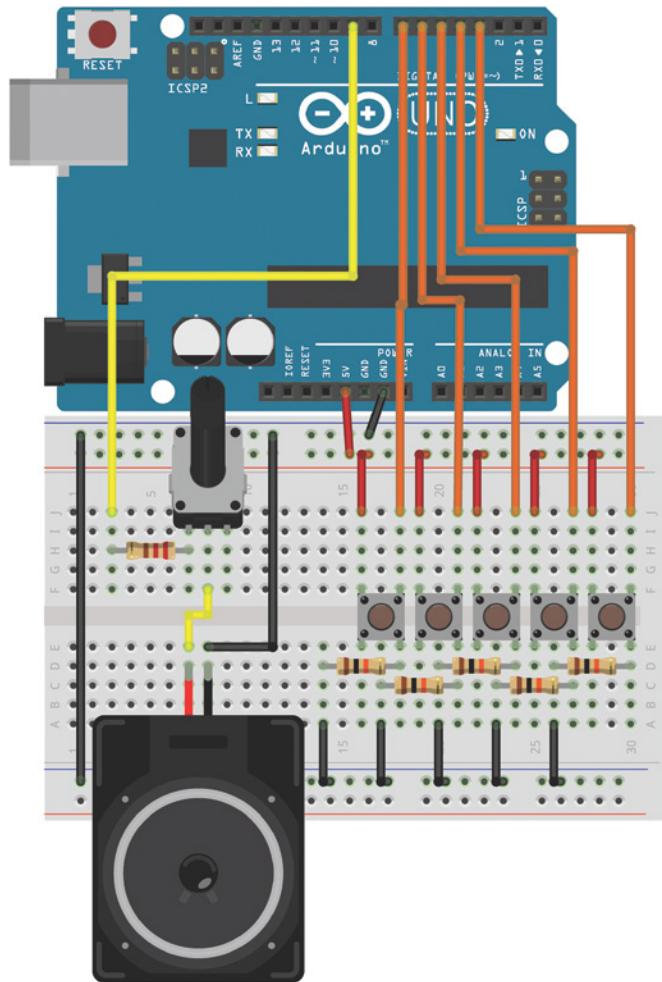


Figure 6-6: Micro piano wiring diagram

Created with Fritzing

The code for the piano is actually very simple. In each iteration through the loop, each button is checked. So long as a button is pressed, a note is played. Here, `tone()` is

used without a duration because the note will play as long as the button is held. Instead, noTone() is called at the end of loop() to ensure that the speaker stops making noise when all the buttons have been released. Because only a few notes are needed, you can copy the values from the header file that you want to use directly into the main program file. In a new sketch, paste in the code shown in Listing 6-2 and upload it to your Arduino. Then, jam away on your piano!

Listing 6-2

Pentatonic micro piano-piano.ino

```
//Pentatonic Piano
//C D E G A

#define NOTE_C 262 //Hz
#define NOTE_D 294 //Hz
#define NOTE_E 330 //Hz
#define NOTE_G 392 //Hz
#define NOTE_A 440 //Hz

const int SPEAKER=9; //Speaker on Pin 9

const int BUTTON_C=7; //Button Pin
const int BUTTON_D=6; //Button Pin
const int BUTTON_E=5; //Button Pin
const int BUTTON_G=4; //Button Pin
const int BUTTON_A=3; //Button Pin

void setup()
{
    //No setup needed
    //Tone function sets outputs
}

void loop()
{
    while (digitalRead(BUTTON_C))
        tone(SPEAKER, NOTE_C);
    while(digitalRead(BUTTON_D))
        tone(SPEAKER, NOTE_D);
    while(digitalRead(BUTTON_E))
        tone(SPEAKER, NOTE_E);
    while(digitalRead(BUTTON_G))
        tone(SPEAKER, NOTE_G);
    noTone(SPEAKER);
}
```

```
while(digitalRead(BUTTON_A))  
  tone(SPEAKER, NOTE_A);  
  
 //Stop playing if all buttons have been released  
 noTone(SPEAKER);  
}
```

Each `while()` loop will continuously call the `tone()` function at the appropriate frequency for as long as the button is held down. The button can be read within the `while()` loop evaluation to avoid having to first save the reading to a temporary value. `digitalRead()` returns a Boolean “true” whenever a button input goes high; the value can be evaluated directly by the `while()` loop. To keep your code neater, you don’t need to use brackets for the contents of a loop if the contents are only one line, as in this example. If you have multiple lines, you must use curly brackets as you have in previous examples.

NOTE To watch a demo video of the micro piano, visit exploringarduino.com/content2/ch6.

Summary

In this chapter, you learned the following:

- Speakers create a pressure wave that travels through the air and is perceived as sound by your ears.
- Changing electric current induces a magnetic field that can be used to create sound from a speaker.
- The `tone()` function can be used to generate sounds of arbitrary frequencies and durations.
- The Arduino programming language supports the use of arrays for iterating through sequences of data.
- Speaker volume can be adjusted using a potentiometer in series with a speaker.

7

USB Serial Communication

Parts You'll Need for This Chapter

- Arduino Uno or Adafruit METRO 328
- USB cable (Type A to B for Uno, Type A to Micro-B for METRO)
- Half-size or full-size breadboard
- Assorted jumper wires
- 220 Ω resistors ($\times 3$)
- 10k Ω trim potentiometer
- 5 mm red LED
- 5 mm common-anode RGB LED

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch7

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:
wiley.com/go/exploringarduino2e

Perhaps the most important part of any Arduino is its capability to be programmed directly via a USB serial port. This feature enables you to program the Arduino without any special hardware, such as an AVRISP mkII. Ordinarily, microcontrollers rely on a dedicated piece of external hardware (such as the mkII) to serve as a *programmer* that connects your computer to the microcontroller you are trying to program. In the case of the Arduino, this programmer is essentially built into the board, instead of being a piece of external hardware. What's more, this gives you a direct connection to the ATmega integrated Universal Synchronous/Asynchronous Receiver and

Transmitter (USART). Using this interface, you can send information between your host computer and the Arduino, or between the Arduino and other serial-enabled components (including other Arduinos).

Both this chapter and the following chapter will cover just about everything you could want to know about connecting an Arduino to your computer via USB and transmitting data between the two components.

Understanding the Arduino's Serial Communication Capabilities

As already alluded to in the introduction to this chapter, different Arduino boards offer different serial implementations, both in terms of how the hardware implements the USB-to-serial adapters and in terms of the software support for various features. In this section, you will learn about the various serial communication hardware interfaces offered on different Arduino boards.

NOTE To learn all about serial communication, check out the tutorial video on this chapter's content web page at exploringarduino.com/content2/ch7.

To begin, you need to understand the differences between serial and USB ports. Depending on how old you are, you might not even remember serial (or technically, RS-232) ports, because they have been replaced primarily by USB ports. Figure 7-1 shows what a standard serial port looks like.



Figure 7-1: Serial port

Credit: Wikipedia (Public Domain)

The original Arduino boards came equipped with a serial port that you connected to your computer with a 9-pin serial cable. Nowadays, few computers still have these ports, although you can buy adapters that convert USB ports into DB-9 serial ports (the type of 9-pin connector used for serial ports). Microcontrollers like the ATmega328P that you find on the Arduino Uno have one hardware serial port. It includes a transmit (TX) and receive (RX) pin that can be accessed on digital pins 0 and 1.

As explained in Chapter 1, “Getting Started and Understanding the Arduino Landscape” (specifically in the sidebar “The Arduino Bootloader and Firmware Setup”), the Arduino is equipped with a bootloader that allows you to program it over this serial interface. To facilitate this, those pins are “multiplexed” (meaning that they are connected to more than one function); they connect, indirectly, to the transmit and receive lines of your USB cable. However, serial and USB are not directly compatible, so one of two methods is used to enable your Arduino to communicate over a modern USB interface. Option one is to use a secondary integrated circuit (IC) to facilitate the conversion between the serial interface from the Arduino and the USB interface that connects to your computer. This IC may be integrated onto the Arduino board, or integrated into a separate board or cable. This is the type of interface that is present on an Uno, where an intermediary IC facilitates USB-to-serial communication. Option two is to choose a microcontroller unit (MCU) that has a USB controller built in (such as the Arduino Leonardo’s 32U4 MCU).

Arduino Boards with an Internal or External FTDI or Silicon Labs USB-to-Serial Converter

As explained in the previous section, many Arduino boards (and Arduino clones) use a secondary integrated circuit to facilitate the USB-to-serial conversion. Popular serial UART “bridge” chips from FTDI and Silicon Labs have just one function: to convert between serial and USB. When your computer connects to an FTDI or Silicon Labs CP210x chip, it will be enumerated as a “Virtual Serial Port” that you can access as if it was a DB9 port wired right into your computer. Figure 7-2 shows the bottom of an Arduino Nano, which utilizes an integrated FTDI chip.

The Adafruit METRO 328 (which can be used in place of the Arduino Uno in this book) uses the CP2104 serial bridge chip.

NOTE Most modern operating systems now have built-in drivers that support FTDI and Silicon Labs chips. If yours doesn’t, you’ll need to install drivers to use boards with these chips. You can find the most recent FTDI drivers for Windows, OS X, and Linux at blum.fyi/ftdi-drivers. Adafruit also provides a Windows installer (Mac and Linux should just work with the Silicon Labs chips) that includes the driver for the Silicon Labs CP210x chips that are used on the Adafruit METRO 328 board. You can find that at blum.fyi/adafruit-windows-drivers. These downloads are also linked from the Second Edition Chapter 7 page on the Exploring Arduino website.

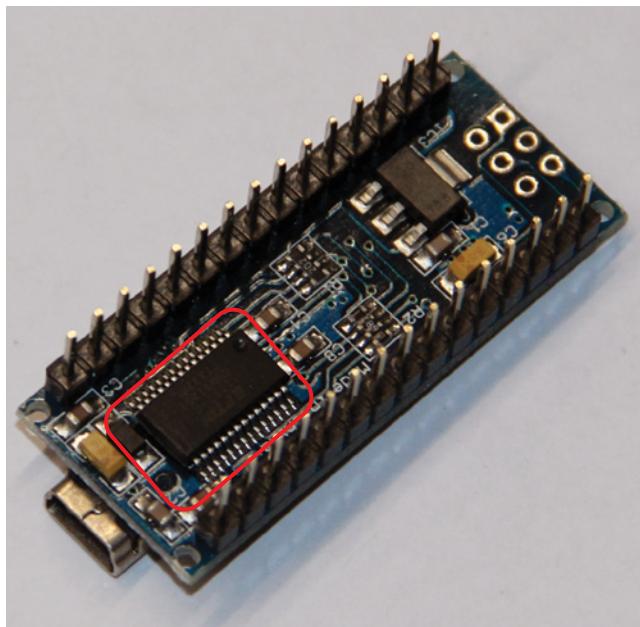


Figure 7-2: Arduino Nano, with integrated FTDI chip shown

On some boards, a USB bridge chip is external to the main board, usually to reduce their size. These boards have a standardized 6-pin “FTDI connector” left for connecting to either an FTDI cable (a USB cable with an FTDI chip built in to the end of the cable) or a small FTDI breakout board. Figure 7-3 and Figure 7-4 show these options.

Using a board with a removable FTDI programmer is great if you are designing a project that will not need to be connected to a computer via USB to run. This will reduce the cost if you are making several devices, and will reduce the overall size of the finished product.

Following is a list of some of the more common first-party Arduino boards that use an onboard FTDI chip. Note that new Arduino boards no longer use an FTDI chip (this is explained more in the following section), so most of these boards have been discontinued. However, many clones of these boards are still available for purchase, so they are listed here for completeness:

- Arduino Nano
- Arduino Extreme (Retired)
- Arduino NG (Retired)
- Arduino Diecimila (Retired)
- Arduino Duemilanove (Retired)
- Original Arduino Mega (Retired)



Figure 7-3: FTDI cable

Credit: Adafruit, adafruit.com

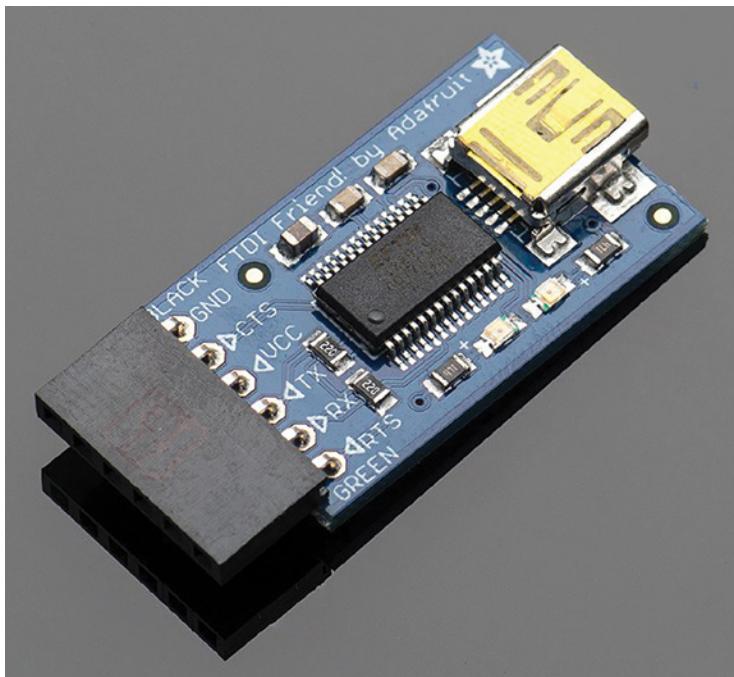


Figure 7-4: Adafruit "FTDI Friend" adapter board

Credit: Adafruit, adafruit.com

Following is a list of first-party Arduino boards that rely on an external FTDI cable or breakout board for programming and serial-to-USB communication:

- Arduino Mini
- Arduino Ethernet
- Original Arduino LilyPad
- Arduino Pro (Retired)
- Arduino Pro Mini (Retired)

Arduino Boards with a Secondary USB-Capable ATmega MCU Emulating a Serial Converter

The Arduino Uno was the first board to use an integrated circuit other than the FTDI chip to handle USB-to-serial conversion. Functionally, it works exactly the same way, with a few minor technical differences. Figure 7-5 shows the Uno's 16U2 serial converter. (This was an 8U2 on older revisions of the Uno.)

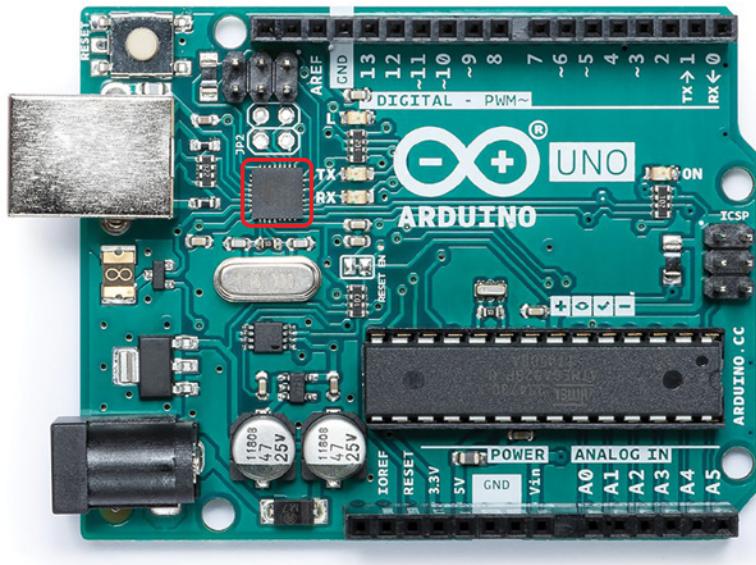


Figure 7-5: View of the Arduino Uno's 16U2 serial converter chip

Credit: Arduino, arduino.cc; emphasis by author

Following is a brief list of the differences:

- First, in Windows, boards with this USB-to-serial conversion solution require a custom driver to be installed. This driver comes bundled with the Arduino IDE when you download it, and it should be installed automatically when you install the IDE. (Drivers are not needed for OS X or Linux.)

- Second, the use of this second microcontroller unit (MCU) for the conversion allows a custom Arduino vendor ID and product ID to be reported to the host computer when the board is connected. When an FTDI-based board is connected to a computer, the computer will enumerate it as a generic USB-serial device. When an Arduino using a non-FTDI converter IC (an ATMega 16U2 in the case of the Uno) is connected, it is identified to the computer as an Arduino.
- Third, because the secondary MCU is fully programmable (it's running a firmware stack called LUFA that emulates a USB-to-serial converter), you can change its firmware to make the Arduino show up as something different from a virtual serial port, such as a joystick, keyboard, or MIDI device. If you were to make this sort of change, the USB-to-serial LUFA firmware would not be loaded, and you would have to program the Arduino directly using the in-circuit serial programmer with a device like the AVRISP mkII.

All modern first-party Arduino boards that aren't built around a USB-capable main MCU now use this approach for USB-to-serial conversion over the use of an FTDI chip. Most third-party boards use a single-function bridge IC, like the CP2104 or an FTDI chip.

Arduino Boards with a Single USB-Capable MCU

The Arduino Leonardo was the first board to have only one chip that acted as both the user-programmable MCU and the USB interface. The Leonardo (and similar Arduino and third-party boards) employs the ATmega32U4 microcontroller, a chip that has direct USB communication built in.

This feature has resulted in several improvements. First, board cost is reduced because fewer parts are required, and because one less factory-programming step is needed to produce the boards. Second, the board can more easily be used to emulate USB devices other than a serial port (such as a keyboard, mouse, or joystick). Third, the single ordinary USART port on the ATmega does not have to be multiplexed with the USB programmer, so communication with the host computer and a secondary serial device (such as a GPS unit) can happen simultaneously. The next chapter covers the usage of these devices as direct USB interfaces to your computer for doing things like emulating a keyboard or joystick.

Arduino Boards with USB-Host Capabilities

Some Arduino boards can connect to USB devices as a host, enabling you to connect traditional USB devices (keyboards, mice, or Android phones) to an Arduino. Naturally,

there must be appropriate drivers to support the device you are connecting to. For example, you cannot just connect a webcam to an Arduino Due and expect to be able to snap photos with no additional work. The Due, Zero, and MKR100 presently support a USB host class that enables you to plug a keyboard or mouse into their host-capable, on-the-go USB port to control it. The Arduino Mega ADK uses the Android Open Accessory (AOA) protocol to facilitate communication between the Arduino and an Android device. This is primarily used for controlling Arduino I/O from an application running on the Android device.

Listening to the Arduino

The most basic serial function that you can perform with an Arduino is to print to the computer's serial terminal. You've already done this in previous chapters. In this section, you will explore this functionality in more depth, and later in this chapter, you will build some desktop apps that respond to the data you send instead of just printing it to the terminal. This process is the same for all Arduinos.

Using *print* Statements

To print data to the terminal, you only need to utilize three functions:

- `Serial.begin(baud_rate)`
- `Serial.print("Message")`
- `Serial.println("Message")`

where `baud_rate` and "Message" are variables that you specify.

As you've already learned, `Serial.begin()` must be called once at the start of the program in `setup()` to prepare the serial port for communication. After you've done this, you can freely use `Serial.print()` and `Serial.println()` functions to write data to the serial port. The only difference between the two functions is that `Serial.println()` adds a line feed at the end of the line (so that the next item printed will appear on the following line). To experiment with this functionality, wire up a simple circuit with a potentiometer connected to pin A0 on the Arduino, as shown in Figure 7-6.

After wiring your potentiometer, load on the simple program, shown in Listing 7-1, that will read the value of the potentiometer and print it as both a raw value and a percentage value.

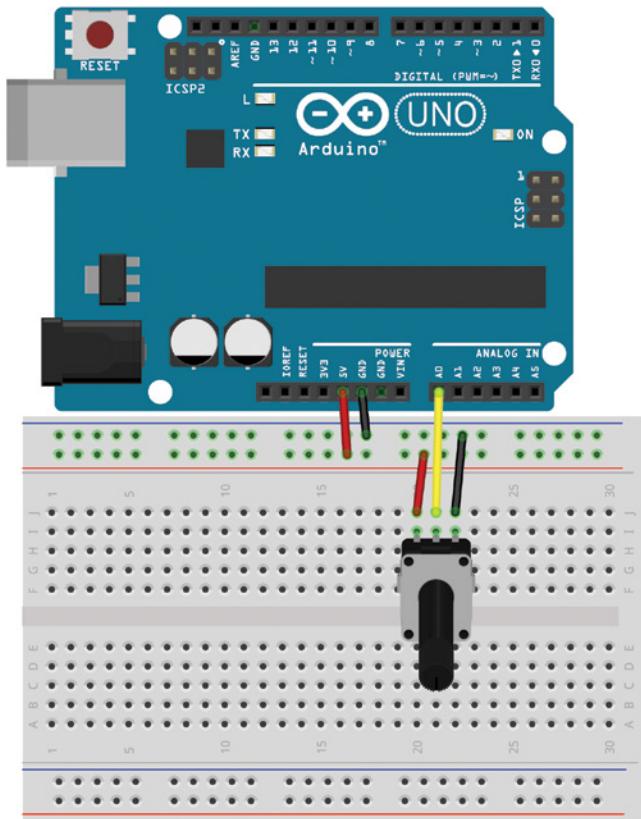


Figure 7-6: Potentiometer wiring diagram

Created with Fritzing

Listing 7-1

Potentiometer serial print test program—pot.ino

```
//Simple Serial Printing Test with a Potentiometer
```

```
const int POT=0; //Pot on analog pin 0

void setup()
{
    Serial.begin(9600); //Start serial port with baud = 9600
}
```

```

void loop()
{
    int val = analogRead(POT);           //Read potentiometer
    int per = map(val, 0, 1023, 0, 100); //Convert to percentage
    Serial.print("Analog Reading: ");
    Serial.print(val);                 //Print raw analog value
    Serial.print(" Percentage: ");
    Serial.print(per);                //Print percentage analog value
    Serial.println("%");              //Print % sign and newline
    delay(1000);                     //Wait 1 second, then repeat
}

```

Using a combination of `Serial.print()` and `Serial.println()` statements, this code prints both the raw and percentage values once per second. Note that by using `Serial.println()` only on the last line, you ensure that each previous transmission stays on the same line.

Open the serial monitor from the Arduino IDE and ensure that your baud rate is set to 9600 to match the value set in the Arduino sketch. You should see the values printing out once per second as you turn the potentiometer.

Using Special Characters

You can also transmit a variety of special characters over serial, which allow you to change the formatting of the serial data you are printing. You indicate these special characters with a backslash escape character (\) followed by a command character. There are a variety of these special characters, but the two of greatest interest are the tab and new-line characters. To insert a tab character, you add a \t to the string. To insert a new-line character, you add a \n to the string. This is particularly useful if you want a newline to be inserted at the beginning of a string, instead of at the end as the `Serial.println()` function does. If, for some reason, you actually want to print \n or \t in the string, you can do so by printing \\n or \\t, respectively. Listing 7-2 is a modification of the previous code, and allows you to use these special characters to show data in a tabular format.

Listing 7-2

Tabular printing using special characters—pot_tabular.ino

```

//Tabular serial printing test with a potentiometer

const int POT=0; //Pot on analog pin 0

void setup()
{
    Serial.begin(9600); //Start Serial Port with Baud = 9600
}

```

```
void loop()
{
    Serial.println("\nAnalog Pin\tRaw Value\tPercentage");
    Serial.println("-----");
    for (int i = 0; i < 10; i++)
    {
        int val = analogRead(POT);           //Read potentiometer
        int per = map(val, 0, 1023, 0, 100); //Convert to percentage

        Serial.print("A0\t");
        Serial.print(val);
        Serial.print("\t");
        Serial.print(per);                //Print percentage analog value
        Serial.println("%");             //Print % sign and newline
        delay(1000);                   //Wait 1 second, then repeat
    }
}
```

As you turn the potentiometer, the output from this program should look something like what's shown in Figure 7-7.

Analog Pin	Raw Value	Percentage
A0	87	8%
A0	0	0%
A0	0	0%
Analog Pin	Raw Value	Percentage
A0	2	0%
A0	43	4%
A0	90	8%
A0	141	13%
A0	163	15%
A0	238	23%
A0	311	30%
A0	376	36%
A0	422	41%
A0	470	45%
Analog Pin	Raw Value	Percentage
A0	514	50%
A0	572	55%
A0	720	70%
A0	1021	99%
A0	1023	100%

Figure 7-7: Screenshot of the serial terminal with tabular data

Changing Data Type Representations

The `Serial.print()` and `Serial.println()` functions are fairly intelligent when it comes to printing out data in the format you are expecting. However, you have options for outputting data in various formats, including hexadecimal, octal, and binary. Decimal-coded ASCII is the default format. The `Serial.print()` and `Serial.println()` functions have an optional second argument that specifies the print format. Table 7-1 includes examples of how you would print the same data in various formats and how it would appear in your serial terminal.

Table 7-1: Serial Data Type Options

Data Type	Example Code	Serial Output
Decimal	<code>Serial.println(23);</code>	23
Hexadecimal	<code>Serial.println(23, HEX);</code>	17
Octal	<code>Serial.println(23, OCT);</code>	27
Binary	<code>Serial.println(23, BIN);</code>	00010111

Talking to the Arduino

What good is a conversation with your Arduino if it's only going in one direction? Now that you understand how the Arduino sends data to your computer, let's spend some time discussing how to send commands from your computer to the Arduino.

Configuring the Arduino IDE's Serial Monitor to Send Command Strings

You've probably already noticed that the Arduino IDE serial monitor has a text entry field at the top, and a drop-down menu at the bottom of the window. Figure 7-8 highlights both of these features.

First, make sure that the line ending drop-down menu is set to Newline. This drop-down menu determines what, if anything, is appended to the end of your commands when you send them to the Arduino. The examples in the following sections assume that you have Newline selected, which just appends a `\n` to the end of any line of text that you send from the text entry field at the top of the serial monitor window.

Unlike with some other terminal programs, the Arduino IDE serial monitor sends your entire command string at one time (at the baud rate you specify) when you press the Enter key or click the Send button. This is in contrast to other serial terminals like PuTTY (an application whose download link is available on this chapter's digital content page at exploringarduino.com/content2/ch7) that send characters as you type them.

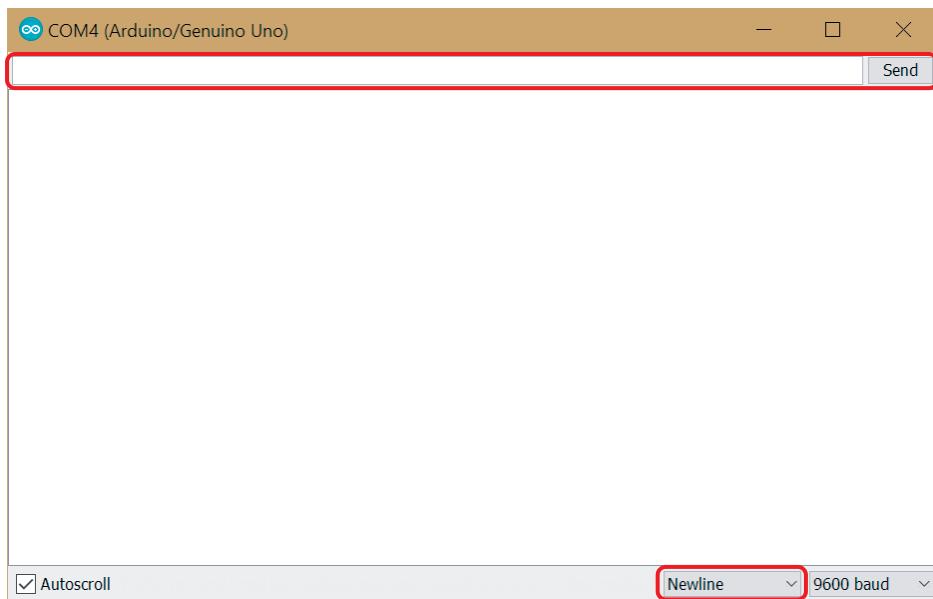


Figure 7-8: Screenshot of the serial terminal, highlighting the text entry field and the drop-down menu for selecting Line Ending Options

Reading Incoming Data from a Computer or Other Serial Device

You will start by using the Arduino IDE’s serial monitor to send commands manually to the Arduino. Once that’s working, you’ll learn how to send multiple command values at once and how to build a simple graphical interface for sending commands.

It’s important to recall that the Arduino’s serial port has a buffer. In other words, you can send several bytes of data at once and the Arduino will queue them up and process them in order, based on the content of your sketch. You do not need to worry about sending data faster than your loop time, but you do need to worry about sending so much data that it overflows the buffer and is lost.

Telling the Arduino to Echo Incoming Data

The simplest thing you can do is to have the Arduino echo back everything that you send it. To accomplish this, the Arduino needs to monitor its serial input buffer and print any character that it receives. To make this happen, you need to implement two new commands from the `Serial` object:

- `Serial.available()` returns the number of characters (or bytes) that are currently stored in the Arduino’s incoming serial buffer. Whenever it’s more than zero, you will read the characters and echo them back to the computer.

- `Serial.read()` reads and returns the next character that is available in the buffer.

Note that each call to `Serial.read()` will only return 1 byte, so you need to run it for as long as `Serial.available()` is returning a value greater than zero. Each time `Serial.read()` grabs a byte, that byte is also removed from the buffer, so the next byte is ready to be read. With this knowledge, you can now write and load the echo program in Listing 7-3 onto your Arduino.

Listing 7-3

Arduino serial echo test-echo.ino

```
//Echo every character

char data; //Holds incoming character

void setup()
{
    Serial.begin(9600); //Serial Port at 9600 baud
}

void loop()
{
    //Only print when data is received
    if (Serial.available() > 0)
    {
        data = Serial.read(); //Read byte of data
        Serial.print(data);   //Print byte of data
    }
}
```

Launch the serial monitor and type anything you want into the text entry field. As soon as you click Send, whatever you typed is echoed back and displayed in the serial monitor. You have already selected to append a “newline” to the end of each command, which will ensure that each response is on a new line. That is why `Serial.print()` is used instead of `Serial.println()` in the preceding sketch; the newline command byte is already included as one of the characters that is being echoed back.

Understanding the Differences between Chars and Ints

When you send an alphanumeric character via the serial monitor, you aren’t actually passing a “5” or an “A”; you’re sending a byte that the computer interprets as a character. In the case of serial communication, the ASCII character set is used to represent all the letters, numbers, symbols, and special commands that you might want to send. The base ASCII character set, shown in Figure 7-9, is a 7-bit set and contains a total of 128 unique characters or commands.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	:	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	-					

Figure 7-9: ASCII table

Credit: Wikipedia (Public Domain)

When reading a value that you've sent from the computer, as you did in Listing 7-3, the Arduino assumes that the data is a `char` type by default. For example, if you were to modify the code to declare `data` as type `int`, sending a value of 5 would return 53 to the serial monitor because the decimal representation of the character 5 is the number 53. You can confirm this by looking at the ASCII reference table in Figure 7-9.

Given that information, you need to take one of three approaches when sending data that is known to be of a particular type (integer, floating point number, and so on). First, you can simply compare the characters directly. If you want to turn an LED on when you send a 1, you can compare the character values like this: `if (Serial.read() == '1')`. Note that the single quotes around the '1' indicate that it should be treated like a character.

A second approach is to convert each incoming byte to an integer by subtracting the zero-valued character, like this: `int val = Serial.read() - '0'`. However, this doesn't work very well if you intend to send numbers that are greater than 9, because they will be multiple digits.

The third, and most versatile, approach is to use a handy function called `parseInt()` that attempts to extract integers from a serial data stream. The examples that follow elaborate on all of these techniques.

Sending Single Characters to Control an LED

Before you dive into parsing larger strings of multiple-digit numbers, you will start by writing a sketch that uses a simple character comparison to control an LED. You'll send a '1' to turn an LED on, and a '0' to turn it off. You first need to wire an LED up to pin 9 of your Arduino as shown in Figure 7-10.

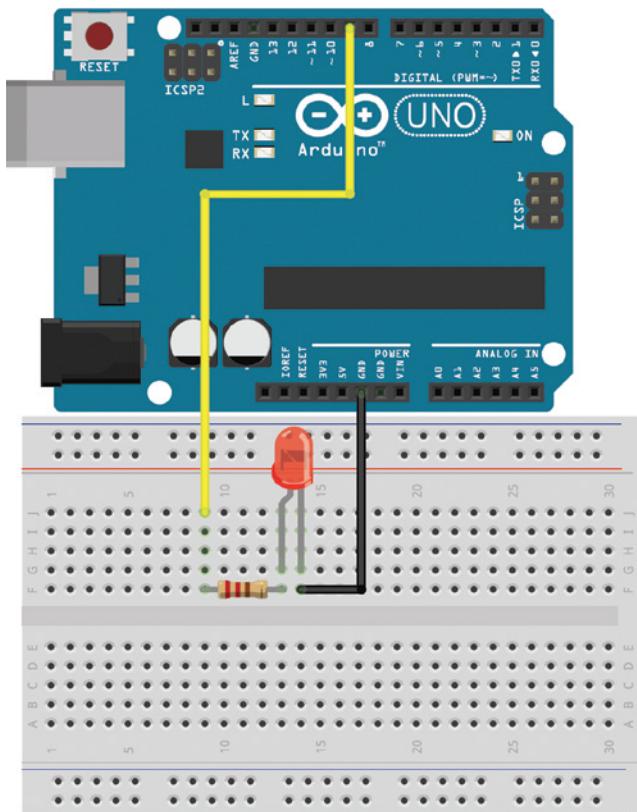


Figure 7-10: Single LED connected to the Arduino on pin 9

Created with Fritzing

As explained in the previous section, when you are only sending a single character, the easiest approach is to do a simple character comparison in an if statement. Each time a character is added to the buffer, it is compared to a '0' or a '1', and the appropriate action is taken. Load up the code in Listing 7-4 and experiment with sending a 0 or a 1 from the serial terminal.

Listing 7-4

Single LED control using characters-single_char_control.ino

```
//Single Character Control of an LED

const int LED=9;

char data; //Holds incoming character

void setup()
{
    Serial.begin(9600); //Serial Port at 9600 baud
    pinMode(LED, OUTPUT);
}

void loop()
{
    //Only act when data is available in the buffer
    if (Serial.available() > 0)
    {
        data = Serial.read(); //Read byte of data
        //Turn LED on
        if (data == '1')
        {
            digitalWrite(LED, HIGH);
            Serial.println("LED ON");
        }
        //Turn LED off
        else if (data == '0')
        {
            digitalWrite(LED, LOW);
            Serial.println("LED OFF");
        }
    }
}
```

Note that an else if statement is used instead of a simple else statement. Because your terminal is also set to send a newline character with each transmission, it's

critical to clear these newline characters from the buffer. `Serial.read()` will read in the newline character, the character will be seen as not equivalent to a '`'0'`' or a '`'1'`', and it will be overwritten the next time `Serial.read()` is called. If just an `else` statement were used, both '`'0'`' and '`'\n'`' would trigger turning the LED off. Even when sending a '`'1'`', the LED would immediately turn off again when the '`'\n'`' was received!

Sending Lists of Values to Control an RGB LED

Sending a single command character is fine for controlling a single digital pin, but what if you want to accomplish some more complex control schemes? This section explores sending multiple comma-separated values to simultaneously command multiple devices. To run this test, you need to wire up a common anode RGB LED as shown in Figure 7-11.

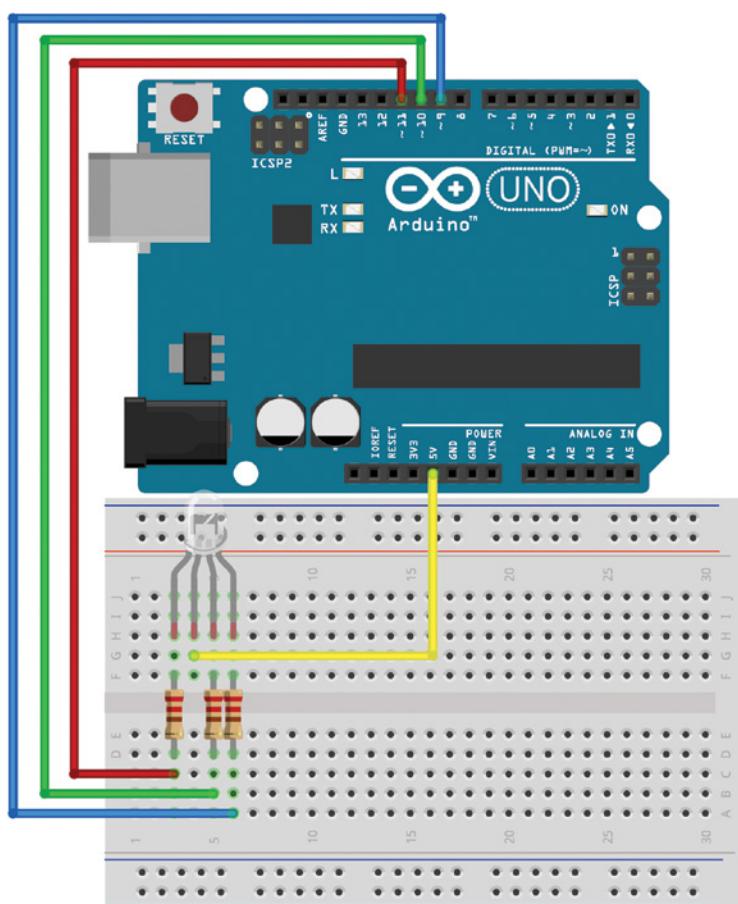


Figure 7-11: RGB LED connected to the Arduino

Created with Fritzing

To control this RGB LED, you send three separate percentage values (0–100) to set the brightness of each LED color. For example, to set all the colors to full brightness, you send 100,100,100. This presents a few challenges:

- You need to differentiate between numbers and commas.
- You need to turn this sequence of characters into integers and map them to 0–255 for controlling the LED with `analogWrite()` functions.
- You need to accommodate the fact that this is a common anode LED, and you are controlling the cathodes (so 255 would turn the LED off, and 0 would turn it to full brightness).
- You need to allow for the possibility that values could be one, two, or three digits.
- Your code should be robust enough to handle receiving poorly formatted data without it corrupting future transmissions (within reason).

Thankfully, the Arduino IDE implements a very handy function for identifying and extracting integers: `Serial.parseInt()`. Each call to this function waits until a non-numeric value enters the serial buffer, and converts the previous digits into an integer. The first two values are read when the commas are detected, and the last value is read when the newline is detected.

To test this function for yourself, load the program shown in Listing 7-5 onto your Arduino.

Listing 7-5

RGB LED control via `serial-list_control.ino`

```
//Sending Multiple Variables at Once
//Send Data in this format: <0-100>,<0-100>,<0,100>\n
//Where the three numbers represent percentage brightness of R, G, B.

//Define LED Pins
const int RED    = 11;
const int GREEN  = 10;
const int BLUE   = 9;

void setup()
{
    Serial.begin(9600); //Serial Port at 9600 baud
    Serial.setTimeout(10); //Serial timeout to wait for for int

    //Set pins as outputs
    pinMode(RED, OUTPUT);
    pinMode(GREEN, OUTPUT);
    pinMode(BLUE, OUTPUT);
```

```
//Turn off the LED
//It is common-anode, so setting the cathode pins to HIGH turns the LED off
digitalWrite(RED, HIGH);
digitalWrite(GREEN, HIGH);
digitalWrite(BLUE, HIGH);
}

void loop()
{
    //Read data when it's available in the buffer
    if (Serial.available() > 0)
    {
        //Expect to receive 3 integers over serial
        //parseInt will "block" until a valid integer is received
        //parseInt knows full integer was received once a comma or newline is seen
        //parseInt only removes invalid characters before the found int, not after
        int val1 = Serial.parseInt();
        int val2 = Serial.parseInt();
        int val3 = Serial.parseInt();

        //Throw out anything that remains in the buffer after integers are read
        while (Serial.available())
        {
            Serial.read();
        }

        //Constrain the received values to be only from 0 to 100%
        int val1c = constrain(val1,0,100);
        int val2c = constrain(val2,0,100);
        int val3c = constrain(val3,0,100);

        //Map the values from percentages to analog values
        int rval = map(val1c,0,100,255,0); //first valid integer
        int gval = map(val2c,0,100,255,0); //second valid integer
        int bval = map(val3c,0,100,255,0); //third valid integer

        //set LED brightness
        analogWrite(RED, rval);
        analogWrite(GREEN, gval);
        analogWrite(BLUE, bval);

        //Report Values that were used to set the LED
        Serial.println("Red: " + String(val1c) + "%");
        Serial.println("Green: " + String(val2c) + "%");
        Serial.println("Blue: " + String(val3c) + "%\n");
    }
}
```

In the `setup()`, you start the serial interface, and use `setTimeout()` to set the timeout to 10 milliseconds. This timeout is used by the `parseInt()` function later in the program. If more than 10 milliseconds pass without another character being received on the serial bus, that function will assume that the current integer that it is parsing is complete; this is just to keep the program from hanging if you send it an incomplete message.

Recall that because the RGB LED is wired with a common anode, you are controlling the connection from the LED cathode to ground. Thus, setting the cathode pin HIGH prevents the flow of current and turns the LED off. Similarly, it means that `analogWrite()` values must be inverted (255 turns the LED off, and 0 sets it to full brightness). The main program loop waits until serial data is available, and extracts the first three integers it can find. If any additional data was transmitted, it is discarded by running `Serial.read()` until the incoming serial buffer is empty. Next, the `constrain()` function is used to ensure that all values are between 0 and 100. Then, the `map()` function is used to map 0 percent to 255 and 100 percent to 0 for use with `analogWrite()`. Finally, the LED is set, and the color values are printed to the serial console as confirmation. The loop then waits for the next set of commands to be received.

To test this program, load it onto your Arduino and open the serial monitor. Enter three values between 0 and 100 separated by a comma, for example, "80,10,80", and hit Send. Try mixing all kinds of pretty colors!

Talking to a Desktop App

Eventually, you're bound to get tired of doing all your serial communication through the Arduino serial monitor. Fortunately, just about any desktop programming language you can think of has libraries that allow it to interface with the serial ports in your computer. You can use your favorite desktop programming language to write programs that send serial commands to your Arduino and that react to serial data being transmitted from the Arduino to the computer.

In this book, Processing is the desktop programming language of choice because it is very similar to the Arduino language with which you have already become familiar. In fact, the Arduino programming language is based on Processing! Other popular desktop languages (that have well-documented serial communication libraries) include Python, Node.js, C, Java, and more. First, you'll learn how to read transmitted serial data in Processing; then you'll learn how you can use Processing to create a simple graphical user interface (GUI) to send commands to your Arduino.

Processing has a fairly simple programming interface, similar to the one you've already been using for the Arduino. In this section, you will install Processing, and then write a simple graphical interface to generate a graphical output based on serial data transmitted from your Arduino. Once that's working, you will implement communication in the opposite direction to control your Arduino from a GUI on your computer.

Installing Processing

Before you begin, you need to install Processing on your machine. Visit processing.org/download (or find the download link on the digital content page for this chapter on exploringarduino.com/content2/ch7) and download the compressed package for your operating system. Simply unzip it to your preferred location, and you are ready to go! Run the Processing application; you should see an IDE that looks like the one shown in Figure 7-12.

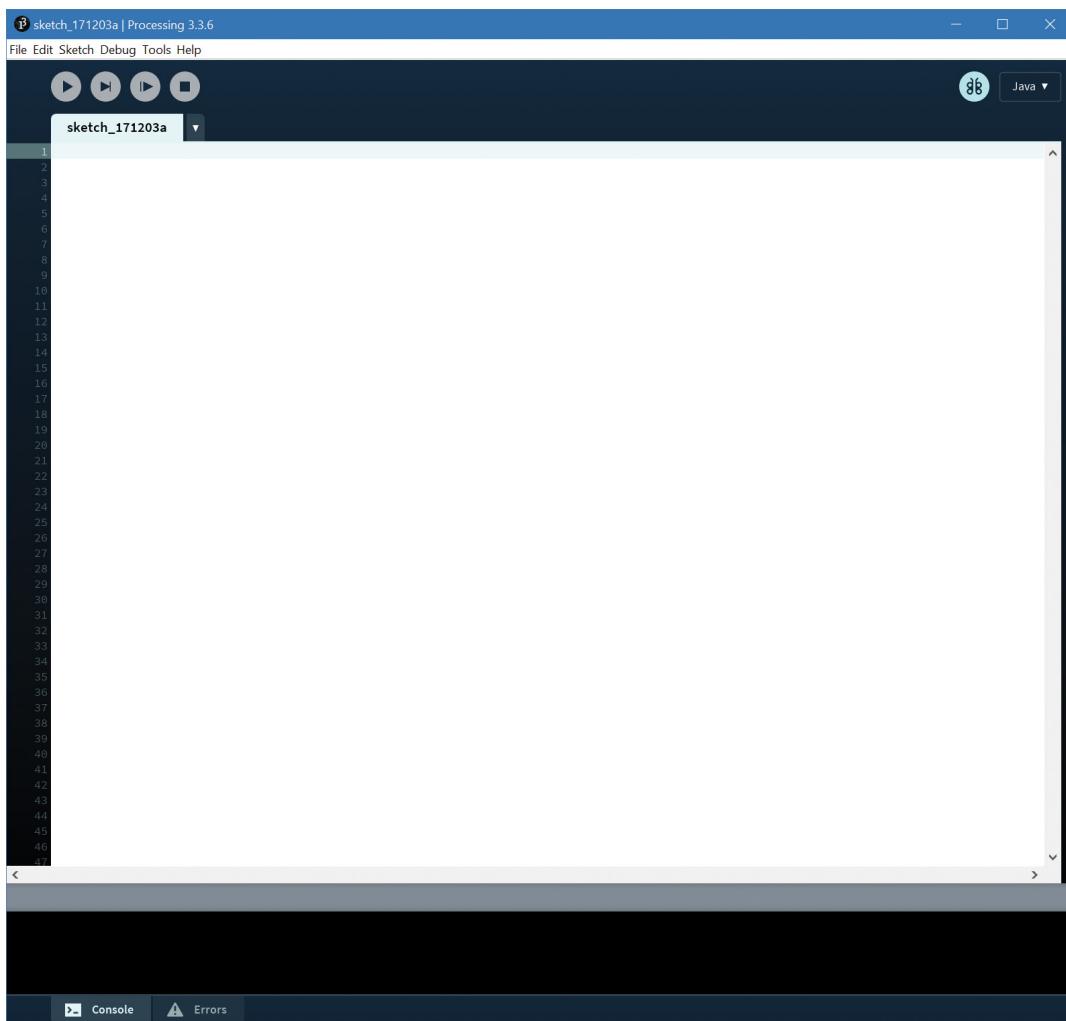


Figure 7-12: The Processing IDE

Controlling a Processing Sketch from Your Arduino

For your first experiment with Processing, you will use a potentiometer connected to your Arduino to control the color of a window on your computer. Wire up your Arduino with a potentiometer, referencing Figure 7-6 again. You already know the Arduino code necessary to send the analog values from the potentiometer to your computer. The fact that you are now feeding the serial data into Processing does not have any impact on the way you transmit it.

Reference the code in Listing 7-6 and load it on to your Arduino. This code sends an updated value of the potentiometer to the computer's serial port every 50 milliseconds. The 50 milliseconds is important; if you were to send these updated values as fast as possible, the Processing sketch wouldn't be able to handle them as quickly as you were sending them, and you would eventually overflow the serial input buffer on your computer.

Listing 7-6

Arduino code to send data to the computer—arduino_read_pot.ino

```
//Sending POT value to the computer

const int POT=0; //Pot on analog pin 0

int val; //For holding mapped pot value

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

void loop()
{
    val = map(analogRead(POT), 0, 1023, 0, 255); //Read and map POT
    Serial.println(val); //Send value
    delay(50); //Delay so we don't flood
                //the computer
}
```

Now comes the interesting part: writing a Processing sketch to do something interesting with this incoming data. The sketch in Listing 7-7 reads the data in the serial buffer and adjusts the brightness of a color on your computer screen based on the value it receives. First, copy the code from Listing 7-7 into a new Processing sketch. You need to change just one important part. The Processing sketch needs to know which serial

port to expect data to arrive on. This is the same port that you've been programming the Arduino from. In the following listing, replace "COM3" with your serial port number. (For example, on Linux and macOS it will look like /dev/ttyUSB0.) You can copy the exact name from within the Arduino IDE if you are unsure.

Listing 7-7

Processing code to read data and change color on the screen—`processing_display_color.pde`

```
//Processing Sketch to Read Value and Change Color on the Screen

//Import and initialize serial port library
import processing.serial.*;
Serial port;

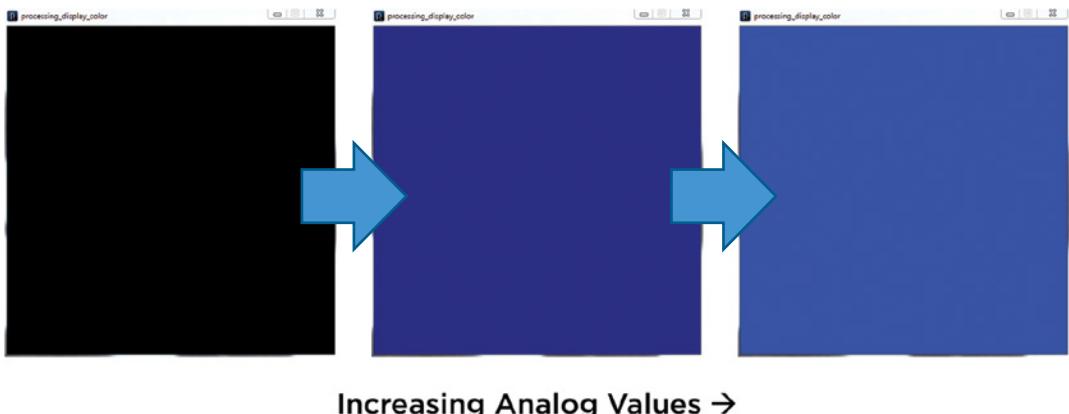
float brightness = 0; //For holding value from pot

void setup()
{
    size(500,500);                                //Window size
    port = new Serial(this, "COM3", 9600);          //Set up serial
    port.bufferUntil('\n');                         //Set up port to read until
                                                    //newline
}

void draw()
{
    background(0,0,brightness); //Updates the window
}

void serialEvent (Serial port)
{
    brightness = float(port.readStringUntil('\n'));//Gets value from Arduino
}
```

After you've loaded the code into your Processing IDE and set the serial port properly, make sure that the Arduino serial monitor isn't open. Only one program on your computer can have access to the serial port at a time. Click the Run button in the Processing IDE (the button with a triangle, located in the top-left corner of the window); when you do so, a small window pops up (see Figure 7-13). As you turn the potentiometer, you should see the color of the window change from black to blue.



Increasing Analog Values →

Figure 7-13: Example windows from the Processing sketch

Now that you've seen it working, let's walk through the code to better understand how the Processing sketch is working. Unlike in an Arduino sketch, the serial library is not automatically imported by Processing. By calling `import processing.serial.*;` and `Serial port;` you are importing the serial library and making a serial object called `port`.

Like the Arduino, Processing has a `setup()` function that runs once at the beginning of the sketch. In this sketch, it sets up the serial port and creates a window that is 500×500 pixels with the command `size(500,500)`. The command `port = new Serial(this, "COM3", 9600)` tells Processing everything it needs to know about creating the serial port. The instance (referred to as "port") will run in this sketch and communicate on COM3 (or whatever your serial port is) at 9600 baud. The Arduino and the program on your computer must agree on the speed at which they communicate; otherwise, you'll get garbage characters. The `port.bufferUntil('\n')` line tells Processing to buffer the serial input and not do anything with the information until it sees a new-line character.

Instead of `loop()`, Processing defines other special functions. This program uses `draw()` and `serialEvent()`. The `draw()` function is similar to Arduino's `loop()`; it runs continuously and updates the display. The `background()` function sets the color of the window by setting red, green, and blue values (the three arguments of the function). In this case, the value from the potentiometer is controlling the blue intensity, and red and green are set to 0. You can change what color your pot is adjusting by simply swapping which argument brightness is filling in. RGB color values are 8-bit values ranging from 0 to 255, which is why the potentiometer is mapped to those values before being transmitted.

The `serialEvent()` function is called whenever the `bufferUntil()` condition that you specified in the `setup()` is met. Whenever a newline character is received, the `serialEvent()` function is triggered. The incoming serial information is read as a string with `port.readStringUntil('\n')`. You can think of a string as an array of text. To use the string as a number, you must convert it to a floating-point number with `float()`. This sets the `brightness` variable, changing the background color of the application window.

To stop the application and close the serial port, click the Stop button in the Processing IDE (the square button, located to the right of the Run button).

Sending Data from Processing to Your Arduino

The obvious next step is to do the opposite of receiving data from your Arduino—send information from the computer to your Arduino. Wire up a common-anode RGB LED to your Arduino as shown in Figure 7-11, and load on the program from earlier that you used to receive a string of three comma-separated values for setting the red, green, and blue intensities (Listing 7-5). Now, instead of sending a string of three values from the serial monitor, you will select a color using a color picker built in Processing.

Load and run the code in Listing 7-8 in Processing. (Remember to adjust the serial port number accordingly, as you did with the previous sketch.) Processing sketches automatically load collateral files from a folder called “data” in the sketch folder. The `hsv.jpg` file is included in the code download for this chapter. Download it and place it in a folder named “data” in the same directory as your sketch. Processing defaults to saving sketches in your Documents folder. The structure will look similar to the one shown in Figure 7-14.

The image in the data folder will serve as the color selector.

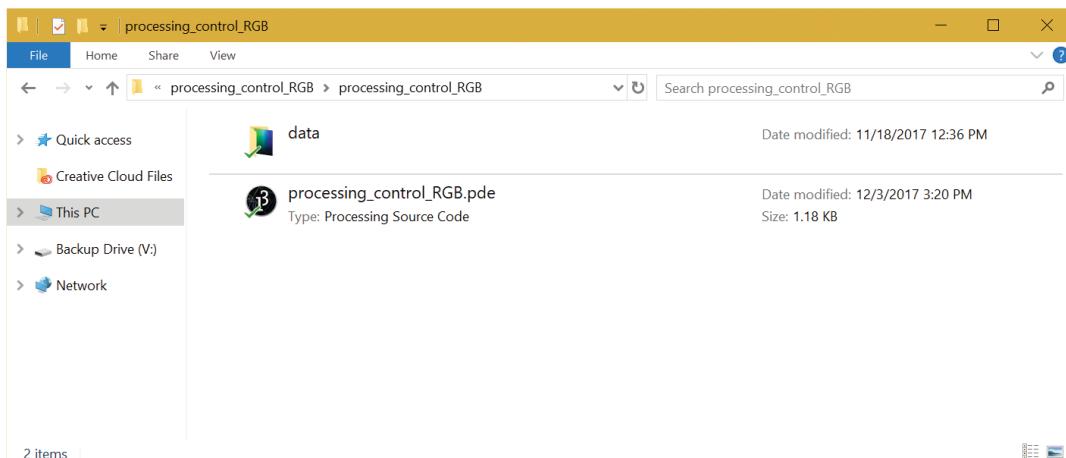


Figure 7-14: Folder structure

Listing 7-8

Processing sketch to set Arduino RGB colors— processing_control_RGB.pde

```
import processing.serial.*; //Import Serial Library
PImage img; //Image Object
Serial port; //Serial Port Object

void setup()
{
    size(640,256); //Size of HSV Image
    img = loadImage("hsv.jpg"); //Load in Background Image
    port = new Serial(this, "COM3", 9600); //Open Serial port
}

void draw()
{
    background(0); //Black Background
    image(img,0,0); //Overlay image
}

void mousePressed()
{
    color c = get(mouseX, mouseY); //Get the RGB color where mouse was pressed
    int r = int(map(red(c), 0, 255, 0, 100));
    int g = int(map(green(c), 0, 255, 0, 100));
    int b = int(map(blue(c), 0, 255, 0, 100));
    String colors = r+","+g+","+b+"\n"; //extract values from color
    print(colors); //print colors for debugging
    port.write(colors); //Send values to Arduino
}
```

When you execute the program, you should see a screen pop up, like the one shown in Figure 7-15. Click different colors; the RGB values will be transmitted to the Arduino to control the RGB LED's color. Note that the serial console also displays the commands being sent to assist you in any debugging.

After you've finished staring at all the pretty colors, look back at the code and consider how it's working. As before, the serial library is imported and a serial object called `port` is created. A `PImage` object called `img` is also created. This will hold the background image. In the `setup()`, the serial port is initialized, the display window is set to the size of the image, and the image is imported into the image object by calling `img = loadImage("hsv.jpg")`. This assumes that the `hsv.jpg` file is located in the data folder as described earlier.

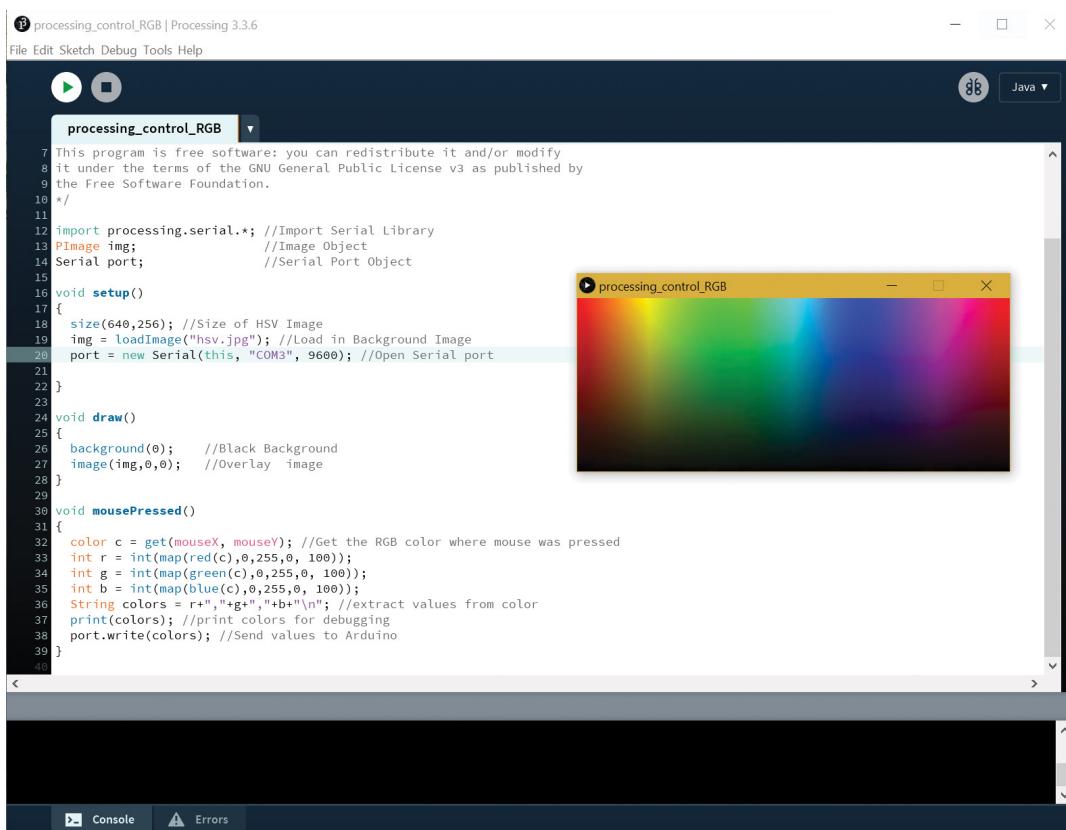


Figure 7-15: Processing color selection screen

In the draw() function, the image is loaded in to the window with image(img,0,0). Here, img is the image you want to draw in the window, and 0,0 are coordinates where the image will start to be drawn. The 0,0 coordinates represent the top-left corner of the application window.

Every time you press the mouse button, the mousePressed() function is called. The color of the pixel where you clicked is saved to a color object named c. The get() method tells the application where to get the color from (in this case, the location of the mouse's X and Y position in the sketch). The sketch then uses the map() function to map the color to the percentage values that the Arduino sketch is expecting. These values are then concatenated into a string that can be sent to the Arduino. These values are also printed to the Processing console so that you can see what is being sent.

Ensure that the Arduino is connected and programmed with the code from Listing 7-5. Run the Processing sketch (with the correct serial port specified) and click around the color map to adjust the color of the LED connected to your Arduino.

Summary

In this chapter, you learned the following:

- Arduinos connect to your computer via a USB-to-serial converter.
- Different Arduinos facilitate a USB-to-serial conversion using either dedicated ICs or built-in USB functionality.
- Your Arduino can print data to your computer via your USB serial connection.
- You can use special serial characters to format your serial printing with new-lines and tabs.
- All serial data is transmitted as characters that can be converted to integers in a variety of ways.
- You can send comma-separated integer lists and use integrated functions to parse them into commands for your sketch.
- You can send data from your Arduino to a Processing desktop application.
- You can receive data from a Processing application on your desktop to control peripherals connected to your Arduino.

8

Emulating USB Devices

Parts You'll Need for This Chapter

- Arduino Leonardo or Seeeduino Lite or Pololu A-Star 32U4 Prime LV
- USB cable (Type A to Micro-B)
- Half-size or full-size breadboard
- Assorted jumper wires
- Pushbuttons ($\times 3$)
- 220Ω resistor
- $10k\Omega$ resistors ($\times 3$)
- Photoresistor
- 5 mm red LED
- TMP36 analog temperature sensor
- Two-axis joystick

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch8

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:
wiley.com/go/exploringarduino2e

In the last chapter, you experimented with USB/serial communication between your computer and your Arduino. To accomplish this task, your computer connected to your Arduino's serial interface, allowing any software capable of interfacing with a serial port to talk to your Arduino. While this is really useful for basic data transfer, it doesn't even come close to using the full potential of what a native USB connection is capable of.

USB is the international de facto standard for connecting computer peripherals; its capabilities are ever expanding with USB SuperSpeed and USB-C connectors that can transport data, HD video, enough power to charge a laptop, and more. USB devices can be recognized by your computer as a variety of things. In this chapter, you'll move beyond USB/serial interfaces to learn how Arduinos with native USB support can act as human-interface devices (USB-HID).

NOTE The exercises in this chapter require an Arduino with native USB capabilities, like the Leonardo. They will not work on Arduino boards that use a secondary chip for the USB/serial interface, such as the Arduino Uno or Adafruit METRO 328. As noted in the parts list, there are a variety of boards from third-party manufacturers that clone the Leonardo's functionality. Specifically, the Seeeduino Lite (available from Adafruit) or the A-Star 32U4 Prime LV (available from Pololu) can be substituted for the Leonardo in all of this chapter's examples. The first-party Arduino boards can sometimes be challenging to keep in stock; you can trust third-party alternatives when they are sold by a reputable distributor such as Adafruit or Pololu. If you use the Seeeduino Lite, ensure its voltage selector switch is set to 5V. To program any of these boards, you can still select Arduino Leonardo in the board selection menu in the Arduino IDE.

The Leonardo, like other Arduinos that implement MCUs that connect directly to USB, has the unique ability to emulate non-serial devices such as a keyboard or mouse. In this chapter, you will learn about using a Leonardo to emulate these devices.

TIP You need to be careful to implement this chapter's examples in a way that does not make reprogramming difficult. For example, if you write a sketch that emulates a mouse and continuously moves your pointer around the screen, you might have trouble clicking the Upload button in the Arduino IDE! If you get stuck with a board that's too hard to program due to its keyboard or mouse input, hold down the Reset button and release it while clicking the Upload button in the Arduino IDE; this will keep the board in its bootloader mode until programming can start.

When you first connect a Leonardo (or equivalent clone) to your computer, the drivers should be installed automatically. On some Windows computers, you might run into issues. If you do, just follow the driver installation instructions from the Arduino website, at blum.fyi/installing-arduino-drivers. (These instructions are also linked from the digital content page for this chapter at exploringarduino.com/content2/ch8.)

Emulating a Keyboard

Using the Leonardo's unique capability to emulate USB devices, you can easily turn your Arduino into a keyboard. Emulating a keyboard allows you to easily send key-combination commands to your computer or type data directly into a file that is open on your computer.

Typing Data into the Computer

The Leonardo can emulate a USB keyboard, sending keystrokes and key combinations. This section explores how to use both capabilities. First, you will write a simple program that records data from a few analog sensors into a comma-separated-value (.csv) format that you can later open with Microsoft Excel or Google Sheets to generate a graph of the values.

Start by opening the text editor of your choice and saving a blank document with a .csv extension. To do this, you can generally choose the file type in the Save dialog box, select All Files, and manually type the filename with the extension, such as **data.csv**. The demo video from this chapter's web page also shows how to create a .csv file.

Next, create a simple circuit like the one shown in Figure 8-1. It will monitor both light and temperature levels using analog sensors that you have already seen in Chapter 3, “Interfacing with Analog Sensors.” In addition to the sensors, the circuit includes a button for turning the logging functionality on and off, and an LED that will indicate whether it is currently logging data.

Using the same debouncing function that you implemented in Chapter 2, “Digital Inputs, Outputs, and Pulse-Width Modulation,” you use the pushbutton to toggle the logging mode on and off. While in logging mode, the Arduino polls the sensors and “types” those values into your computer in a comma-separated format once every second. An indicator LED remains illuminated while you are logging data.

Because you want the Arduino to be constantly polling the state of the button, you cannot use a `delay()` function to wait 1000 milliseconds between each update. Instead, you use the `millis()` function, which returns the number of milliseconds since the Arduino was last reset. You can make the Arduino send data every time the `millis()` function returns a multiple of 1000 milliseconds, effectively creating a nonblocking 1-second delay between transmissions. To do this, you can use the modulo operator (%). Modulo returns the remainder of a division. If, for example, you executed `1000 % 1000`, you would find that the result was 0 because $1000/1000 = 1$, with a remainder of 0. On the other hand, `1500 % 1000` would return 500 because $1500/1000 = 1$, with a remainder of 500. If you take the modulus of `millis()` with 1000, the result is zero every time

`millis()` reaches a value that is a multiple of 1000. By checking this with an `if()` statement, you can execute code once every second.

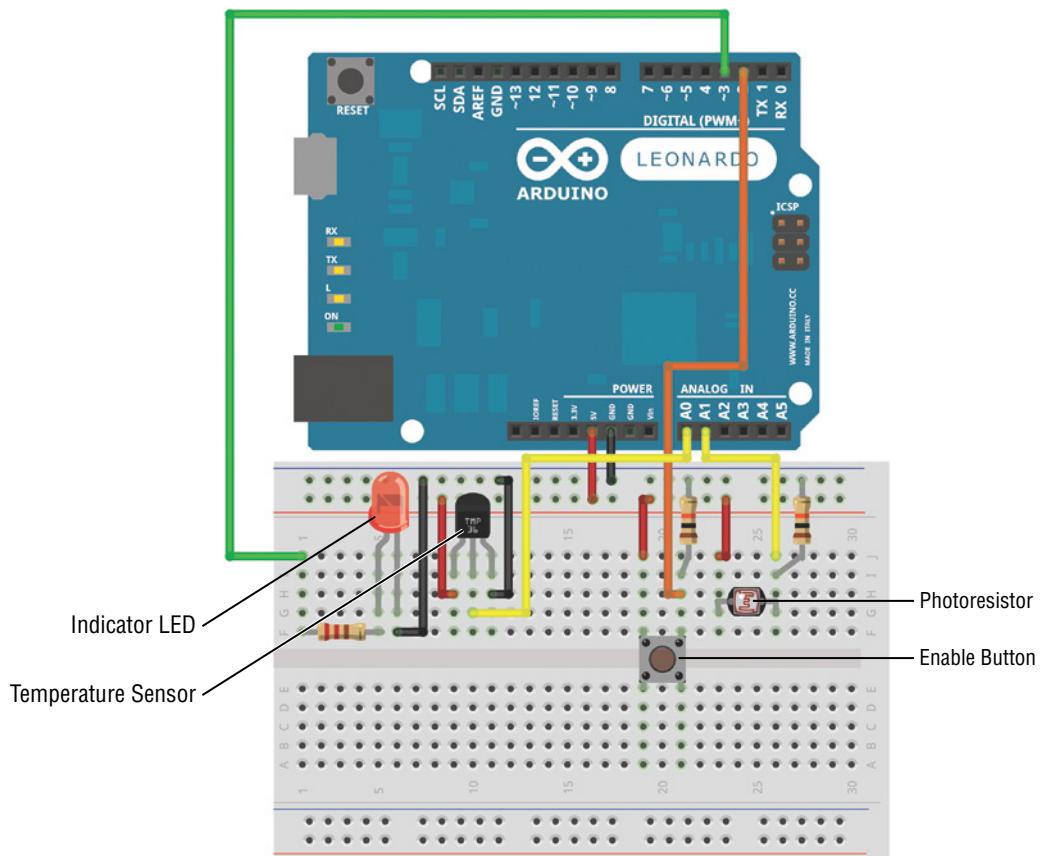


Figure 8-1: Temperature and light sensor circuit

Created with Fritzing

Examine the code in Listing 8-1 and load it onto your Arduino Leonardo. Ensure that you've selected Arduino Leonardo from the Tools > Board menu in the Arduino IDE (this option will work even if you are actually using the Seeeduino Lite).

Listing 8-1

Temperature and light data logger—`csv_logger.ino`

```
//Light and Temp Logger
```

```
#include <Keyboard.h>
```

```
const int TEMP    =0;      //Temp sensor on analog pin 0
const int LIGHT   =1;      //Light sensor on analog pin 1
const int LED     =3;      //Red LED on pin 3
const int BUTTON  =2;      //The button is connected to pin 2

boolean lastButton = LOW;   //Last button state
boolean currentButton = LOW; //Current button state
boolean running = false;    //Not running by default
int counter = 1;            //An index for logged data entries

void setup()
{
    pinMode (LED, OUTPUT); //Set red LED as output
    Keyboard.begin();      //Start keyboard emulation
}

void loop()
{
    currentButton = debounce(lastButton);           //Read debounced state

    if (lastButton == LOW && currentButton == HIGH) //If it was pressed...
        running = !running;                         //Toggle running state

    lastButton = currentButton;                     //Reset button value

    if (running)                                  //If the logger is running
    {
        digitalWrite(LED, HIGH);                  //Turn the LED on
        if (millis() % 1000 == 0)                 //If time is multiple of 1000ms
        {
            int temperature = analogRead(TEMP); //Read the temperature
            int brightness = analogRead(LIGHT); //Read the light level
            Keyboard.print(counter);          //Print the index number
            Keyboard.print(",");
            Keyboard.print(temperature);       //Print the temperature
            Keyboard.print(",");
            Keyboard.println(brightness);     //Print brightness (and a newline)
            counter++;                      //Increment the counter
        }
    }
    else
    {
        digitalWrite(LED, LOW); //If the logger not running, the LED off
    }
}

/*
 * Debouncing Function
```

```
* Pass it the previous button state,  
* and get back the current debounced button state.  
*/  
boolean debounce(boolean last)  
{  
    boolean current = digitalRead(BUTTON);           //Read the button state  
    if (last != current)                            //If it's different...  
    {  
        delay(5);                                 //Wait 5ms  
        current = digitalRead(BUTTON);             //Read it again  
    }  
    return current;                                //Return the current value  
}
```

Before you test the data logger, I'll highlight some of the new functionality that has been implemented in this sketch. Similar to how you initialized the serial communication, the keyboard communication is initialized by putting `Keyboard.begin()` in the `setup()`. Unlike the `Serial` library, which is always included by default, you must explicitly tell the compiler to load the keyboard library by adding `#include <Keyboard.h>` to the top of the file.

Each time through `loop()`, the Arduino checks the state of the button and runs the debouncing function that you are already familiar with. When the button is pressed, the value of the *running* variable is inverted. This is accomplished by setting it to its opposite with the `!` operator.

While the Arduino is in *running* mode, the logging step is only executed every 1000 milliseconds using the logic described previously. The keyboard functions work very similarly to the serial functions. `Keyboard.print()` “types” the given string into your computer. After reading the two analog sensors, the Arduino sends the values to your computer as keystrokes. The keyboard library also has a `Keyboard.println()` function that emulates pressing the Enter key after sending the provided text. An incrementing counter and both analog values are entered in a comma-separated format with a new line after each entry.

Follow the demo video from this chapter's web page to see this sketch in action. Make sure that your cursor is actively positioned in a text document, and then press the button to start logging. You should see the document begin to populate with data. Hold your hand over the light sensor to change the value, or squeeze the temperature sensor to see the value increase. When you have finished, press the button again to stop logging. After you save your file, you can import it into the spreadsheet application of your choice and graph it over time. This is shown in the demo video.

NOTE To watch the demo video of the live temperature and light logger, visit exploringarduino.com/content2/ch8.

Commanding Your Computer to Do Your Bidding

In addition to typing like a keyboard, you can also use the Leonardo to emulate key combinations. On Windows computers, pressing the Windows+L keys locks the computer screen. (On Linux, you can use Control+Alt+L.) Using that knowledge paired with a light sensor, you can have your computer lock automatically when you turn the lights off. OS X uses the Control+Shift+Eject or Control+Shift+Power keys to lock the machine, which can't be emulated by the Leonardo because it cannot send an Eject or Power simulated button press. In this example, you learn how to lock a Windows computer. You can continue to use the same circuit shown in Figure 8-1, although only the light sensor will be used in this example.

Run the previous sketch at a few different light levels and see how the light sensor reading changes. Using this information, you should pick a threshold value below which you want your computer to lock. (In my room, I found that with the lights off, the value was about 300, and it was about 700 with the lights on. So, I chose a threshold value of 500.) When the light sensor value drops below that value, the lock command will be sent to the computer. You might want to adjust this value for your environment.

Load the sketch in Listing 8-2 on to your Arduino. Just make sure you have your threshold set to a reasonable value first, by testing what light levels in your room correspond to various analog levels. If you pick a poorly calibrated value, it might lock your computer as soon as you upload it!

Listing 8-2

Light-based computer lock-lock_computer.ino

```
//Locks your computer when you turn off the lights

#include <Keyboard.h>

const int LIGHT      =1;      //Light sensor on analog pin 1
const int THRESHOLD =500;    //Brightness must drop below this level
                           //to lock computer

void setup()
{
  Keyboard.begin();
}

void loop()
{
  int brightness = analogRead(LIGHT);      //Read the light level
```

```
if (brightness < THRESHOLD)
{
    Keyboard.press(KEY_LEFT_GUI);
    Keyboard.press('1');
    delay(100);
    Keyboard.releaseAll();
}
```

After loading the program, try flipping the lights off. Your computer should lock immediately. The following video demo shows this program in action. Running `Keyboard.press()` is equivalent to starting to hold a key down. So, if you want to hold down the Windows key and the L key at the same time, you run `Keyboard.press()` on each key. Then, you delay for a short period of time and run the `Keyboard.releaseAll()` function to let go of, or release, the keys. Special keys are defined on the Arduino website, at blum.fyi/arduino-keyboard-modifiers. (This definition table is also linked from the content page for this chapter at exploringarduino.com/content2/ch8.)

NOTE To watch the demo video of the light-activated computer lock, visit exploringarduino.com/content2/ch8.

Emulating a Mouse

Using a two-axis joystick and some pushbuttons, you can use an Arduino Leonardo to make your own mouse! The joystick will control the mouse location, and the buttons will control the left, middle, and right buttons of the mouse. Just like with the keyboard functionality, the Arduino language has some great functions built in that make it easy to control mouse functionality.

First things first: get your circuit set up with a joystick and some buttons, as shown in Figure 8-2. Don't forget that your buttons need to have pull-down resistors! The joystick will connect to analog pins 0 and 1. (Joysticks are actually just two potentiometers hooked up to a knob.) When you move the joystick all the way in the x direction, it maxes out the x potentiometer, and the same goes for the y direction.

The diagram shows the Parallax 2-Axis joystick, which is available from Adafruit or Parallax. A variety of other vendors also make joysticks with similar interfaces. For details on the wiring of this joystick, check out the documentation links on the Parallax website, at blum.fyi/parallax-2-axis-joystick. Depending on the orientation of the joystick, you might need to adjust the bounds of the map function or swap the analog pin that the X_AXIS and Y_AXIS constants are set to in Listing 8-3.

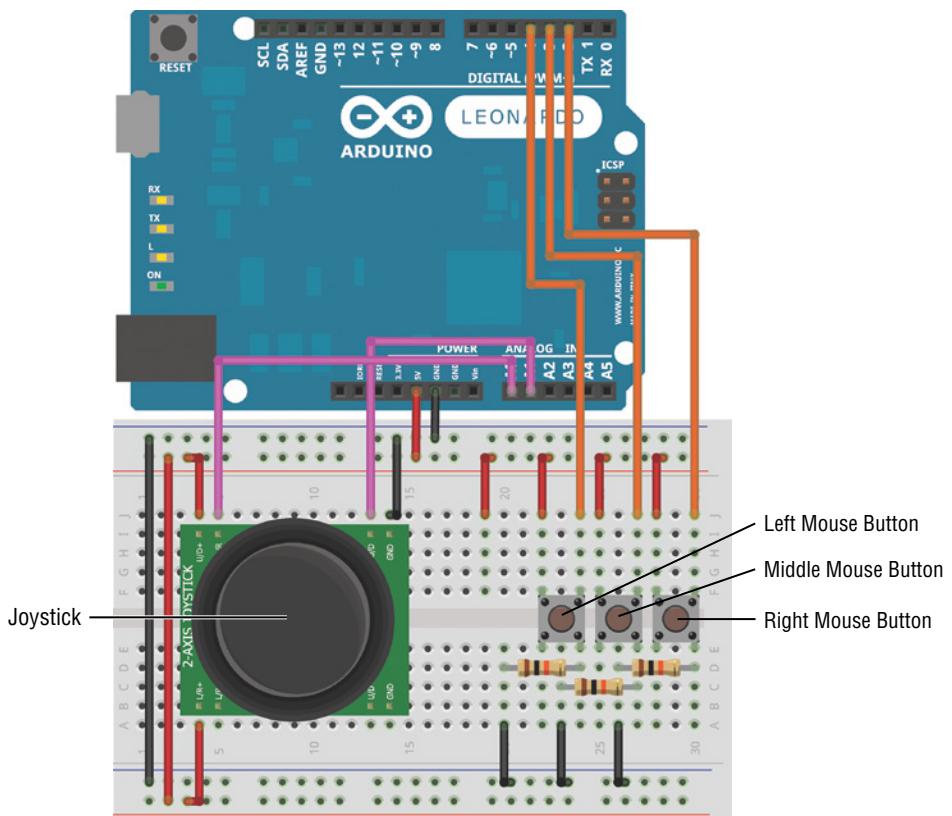


Figure 8-2: Joystick Leonardo mouse circuit

Created with Fritzing

After you've wired the circuit, it's time to load some code onto the Leonardo. Load up the code in Listing 8-3 and play with the joystick and buttons; the pointer on your screen should respond accordingly.

Listing 8-3

Mouse control code for the Leonardo-mouse.ino

```
// Make a Mouse!
```

```
#include <Mouse.h>

const int LEFT_BUTTON = 4; //Input pin for the left button
const int MIDDLE_BUTTON = 3; //Input pin for the middle button
```

```
const int RIGHT_BUTTON = 2; //Input pin for the right button
const int X_AXIS = 0; //Joystick x-axis analog pin
const int Y_AXIS = 1; //Joystick y-axis analog pin

void setup()
{
    Mouse.begin();
}

void loop()
{
    int xVal = readJoystick(X_AXIS); //Get x-axis movement
    int yVal = readJoystick(Y_AXIS); //Get y-axis movement

    Mouse.move(xVal, yVal, 0); //Move the mouse

    readButton(LEFT_BUTTON, MOUSE_LEFT); //Control left button
    readButton(MIDDLE_BUTTON, MOUSE_MIDDLE); //Control middle button
    readButton(RIGHT_BUTTON, MOUSE_RIGHT); //Control right button

    delay(5); //This controls responsiveness
}

//Reads joystick value, scales it, and adds dead range in middle
int readJoystick(int axis)
{
    int val = analogRead(axis); //Read analog value
    val = map(val, 0, 1023, -10, 10); //Map the reading

    if (val <= 2 && val >= -2) //Create dead zone to stop mouse drift
        return 0;

    else //Return scaled value
        return val;
}

//Read a button and issue a mouse command
void readButton(int pin, char mouseCommand)
{
    //If button is depressed, click if it hasn't already been clicked
    if (digitalRead(pin) == HIGH)
    {
        if (!Mouse.isPressed(mouseCommand))
        {
            Mouse.press(mouseCommand);
        }
    }
}
```

```
//Release the mouse if it has been clicked.  
else  
{  
    if (Mouse.isPressed(mouseCommand))  
    {  
        Mouse.release(mouseCommand);  
    }  
}  
}  
}
```

This is definitely one of the more complicated sketches that has been covered so far, so it's worth stepping through it to understand both the newly introduced functions and the program flow used to make the joystick mouse.

As with the keyboard functionality, it's necessary to include the mouse library with `#include <Mouse.h>`. Each of the button and joystick pins are defined at the top of the sketch, and the mouse library is started in the setup. Each time through the loop, the joystick values are read and mapped to movement values for the mouse. The mouse buttons are also monitored, and the button presses are transmitted if necessary.

A `readJoystick()` function was created to read the joystick values and map them. Each joystick axis has a range of 1024 values when read into the analog-to-digital converter (ADC). However, mouse motions are relative. In other words, passing a value of 0 to `Mouse.move()` for each axis will result in no movement on that axis. Passing a positive value for the x-axis will move the mouse to the right, and a negative value will move it to the left. The larger the value, the more the mouse will move. Hence, in the `readJoystick()` function, a value of 0 to 1023 is mapped to a value of -10 to 10. A small buffer value around 0 is added where the mouse will not move. This is because even while the joystick is in the middle position, the actual value may fluctuate around 512. By setting the desired distance back to 0 after being mapped within a certain range, you guarantee that the mouse will not move on its own while the joystick is not being actuated. Once the values are ascertained, `Mouse.move()` is given the x and y values to move the mouse. A third argument for `Mouse.move()` determines the movement of the scroll wheel.

To detect mouse clicks, the `readButton()` function was created so that it can be repeated for each of the three buttons to detect. The function detects the current state of the mouse with the `Mouse.isPressed()` command and controls the mouse accordingly using the `Mouse.press()` and `Mouse.release()` functions.

NOTE To watch a demo video of the joystick mouse controlling a computer pointer, check out exploringarduino.com/content2/ch8.

Summary

In this chapter, you learned the following:

- The Arduino Leonardo's native USB support enables it to emulate USB devices like keyboards and mice.
- By emulating key presses, an Arduino can be used to trigger special functions on an attached computer (such as locking the screen).
- A joystick is made by combining the signals from two orthogonally mounted potentiometers.

9

Shift Registers

Parts You'll Need for This Chapter

- Arduino Uno or Adafruit METRO 328
- USB cable (Type A to B for Uno, Type A to Micro-B for METRO)
- Half-size or full-size breadboard
- Assorted jumper wires
- 220 Ω resistors ($\times 8$)
- 5 mm red LEDs ($\times 8$)
- 5 mm green LEDs ($\times 4$)
- 5 mm yellow LEDs ($\times 3$)
- Sharp GP2Y0A21YK0F IR distance sensor with JST cable
- SN74HC595N shift register

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch9

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:
wiley.com/go/exploringarduino2e

As you plug away building exciting new projects with your Arduino, you might already be thinking: "What happens when I run out of pins?" Indeed, one of the most common uses for the Arduino platform is to put an enormous number of blinking LEDs on just about anything. Light up your room! Light up your computer! Light up your dog! Okay, maybe not that last one.

But there's a problem: What happens when you want to start blinking 50 LEDs (or controlling other digital outputs), but you've used up all of your I/O pins? That's where shift registers can come in handy. With shift registers, you can expand the I/O capabilities of your Arduino without having to pay a lot more for an expensive microcontroller with additional I/O pins. In this chapter, you'll learn how shift registers work, and you'll implement both the software and hardware necessary to interface your Arduino with shift registers for the purpose of expanding the digital output capabilities of your Arduino. Once you've completed the exercises in this chapter, you will be familiar with shift registers and will also be able to make more informed design decisions when developing a project with a large number of digital outputs.

CHOOSING THE RIGHT ARDUINO FOR THE JOB

This chapter, like most of the earlier chapters, uses the Arduino Uno (or the equivalent Adafruit METRO 328) as a development platform. Any other Arduino will work just as well to complete the exercises in this chapter, but it's worth considering why you might want to use one Arduino over another for a particular project. For example, you might already be wondering why you wouldn't just use an Arduino with more I/O pins, such as the Mega 2560 or the Due. Of course, that is a very reasonable way to complete projects that require more outputs. However, as an engineer, you should always be mindful of other considerations when designing a new project. If you only need the processing power of an Uno, but you need more digital outputs, for example, then adding a few shift registers will be considerably cheaper than upgrading your entire platform, as well as more compact. As a tradeoff, it will also require you to write slightly more complex code, and it might require more debugging time.

Understanding Shift Registers

A *shift register* is a device that accepts a stream of serial bits and simultaneously outputs the values of those bits onto parallel I/O pins. Most often, shift registers are used for controlling large numbers of LEDs, such as the configurations found in seven-segment displays or LED matrices. Before you dive into using a shift register with your Arduino,

consider the diagram in Figure 9-1, which shows the inputs and outputs to a serial-to-parallel shift register. Variations to this diagram throughout the chapter illustrate how different inputs affect the outputs.

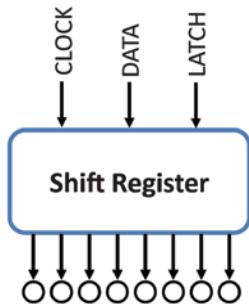


Figure 9-1: Shift register input/output diagram

The eight circles represent LEDs connected to the eight outputs of the shift register (through current-limiting resistors, of course). The three inputs are the serial communication lines that connect the shift register to the Arduino.

Sending Parallel and Serial Data

There are essentially two ways to send multiple bits of data. Recall that the Arduino, like all microcontrollers, is digital; it only understands 1s and 0s. So, if you want sufficient data to control eight LEDs digitally (each one on or off), you need to find a way to transmit a total of 8 bits of information. In previous chapters, you did this in parallel by using the `digitalWrite()` and `analogWrite()` commands to exert control over multiple I/O pins. For an example of parallel information transmission, suppose that you were to turn on eight LEDs with eight digital outputs; all the bits would be transmitted on independent I/O pins at the same time.

In Chapter 7, “USB Serial Communication,” you learned about serial transmission, which transmits 1 bit of data at a time. Shift registers allow you to easily convert between serial and parallel data transmission techniques. This chapter focuses on serial-to-parallel shift registers, sometimes called *serial in, parallel out* (SIPO) shift registers. With these handy devices, you can “clock in” multiple bytes of data serially,

and output them from the shift register in parallel. You can also chain together shift registers, and thus control hundreds of digital outputs with just three Arduino I/O pins.

Working with the 74HC595 Shift Register

For this project, you'll be using the 74HC595 shift register. Take a look at the pin-out diagram from the datasheet shown in Figure 9-2.

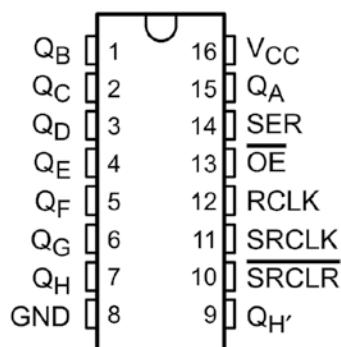


Figure 9-2: Shift register pin-out diagram

Credit: Courtesy of Texas Instruments Incorporated

Understanding the Shift Register pin Functions

Following is a breakdown of the shift register pin functions:

- pins Q_A through Q_H represent the eight parallel outputs from the shift register (connected to the circles shown in Figure 9-1).
- V_{CC} will connect to 5V.
- GND will connect to a shared ground with the Arduino.
- The SER pin is represented by the DATA input in Figure 9-1. This is the pin where you will feed in eight sequential binary bits to set the values of the parallel outputs.
- The SRCLK pin is represented by the CLOCK pin in Figure 9-1. Every time this pin goes high, the values in the register shift by 1 bit. It will be pulsed eight times to pull in all the data that you are sending on the data pin.

- The RCLK pin is represented by the LATCH pin in Figure 9-1. Also known as the *register clock pin*, the latch pin is used to “commit” your recently shifted serial values to the parallel outputs all at once. This pin allows you to sequentially shift data into the chip and have all the values show up on the parallel outputs at the same time.

You will not be using the SRCLR or \overline{OE} pins in these examples, but you might want to use them for your project, so it’s worth understanding what they do. \overline{OE} stands for “output enable.” The bar over the pin name indicates that it is *active low*. In other words, when the pin is held low, the output will be enabled. When it is held high, the output will be disabled. In these examples, this pin will be connected directly to ground, so that the parallel outputs are always enabled. You could alternatively connect this pin to an I/O pin of the Arduino to simultaneously turn all the LEDs on or off. The SRCLR pin is the *serial clear pin*. When pulled low, it empties the contents of the shift register. For your purposes in this chapter, you will tie it directly to 5V to prevent the shift register values from being cleared.

Understanding How the Shift Register Works

The shift register is a synchronous device; it only acts on the rising edge of the clock signal. Every time the clock signal transitions from low to high, all the values currently stored in the eight output registers are shifted over one position. (The last one is either discarded or output on the Q_H pin if you are cascading registers.) Simultaneously, the value currently on the DATA input is shifted into the first position. When this is done eight times, the present values are shifted out and the new values are shifted into the register. The LATCH pin is set high at the end of this cycle to make the newly shifted values appear on the outputs. The flowchart shown in Figure 9-3 further illustrates this program flow. Suppose, for example, that you want to set every other LED to the ON state (Q_A , Q_C , Q_E , Q_G). Represented in binary, you would want the output of the parallel pins on the shift register to look like this: 10101010.

Now, follow the previously described steps for writing to the shift register. First, the LATCH pin is set low so that the current LED states are not changed while new values are shifted in. Then, the LED states are shifted into the registers in order on the CLOCK edge from the DATA line. After all the values have been shifted in, the LATCH pin is set high again, and the values are output from the shift register.

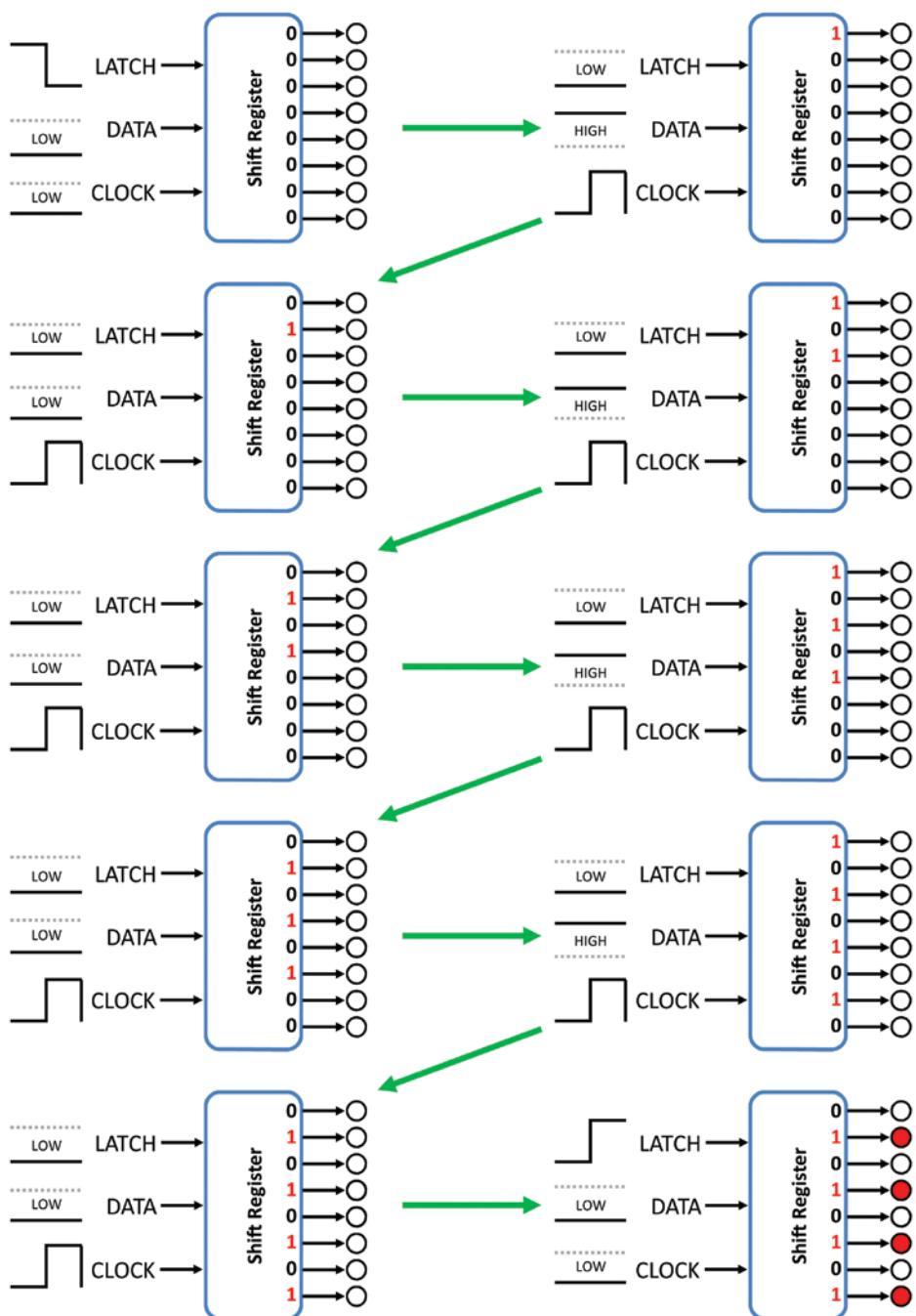


Figure 9-3: Shifting a value into a shift register

Shifting Serial Data from the Arduino

Now that you understand what's happening behind the scenes, you can write the Arduino code to control the shift register. As with all your previous experiments, you can use a convenient function that's built in to the Arduino IDE to shift data into the register IC. The `shiftOut()` function lets you easily shift out 8 bits of data onto an arbitrary I/O pin. It accepts four parameters:

- The data pin number
- The clock pin number
- The bit order
- The value to shift out. If, for example, you want to shift out the alternating pattern described in the previous section, you can use the `shiftOut()` function as follows:

```
shiftOut(DATA, CLOCK, MSBFIRST, B10101010);
```

The `DATA` and `CLOCK` constants are set to the pin numbers for those lines. `MSBFIRST` indicates that the most significant bit will be sent first (the leftmost bit when looking at the binary number to send). You could alternatively send the data with the `LSBFIRST` setting, which would start by transmitting the bits from the right side of the binary data. The final parameter is the number to be sent. By putting a capital `B` before the number, you are telling the Arduino IDE to interpret the following numbers as a binary value rather than as a decimal integer.

Next, you will build a physical version of the system that you learned about in the previous sections. First, you need to get the shift register wired up to your Arduino:

- The `DATA` pin will connect to pin 8.
- The `LATCH` pin will connect to pin 9.
- The `CLOCK` pin will connect to pin 10.

Don't forget to use current-limiting resistors with your LEDs. Reference the diagram shown in Figure 9-4 to set up the circuit.

Now, using your understanding of how shift registers work, and of the `shiftOut()` function, you can use the code in Listing 9-1 to write the alternating LED pattern to the attached LEDs.

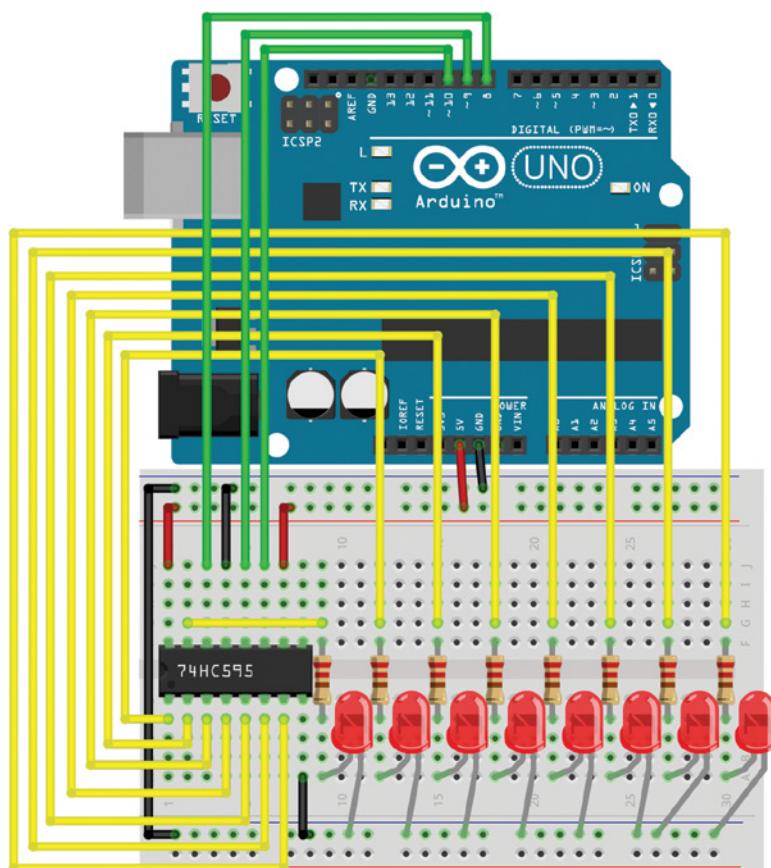


Figure 9-4: Eight-LED shift register circuit diagram

Created with Fritzing

Listing 9-1

Alternating LED pattern on a shift register—`alternate.ino`

```
//Alternating LED Pattern using a Shift Register
```

```
const int SER = 8; //Serial Output to Shift Register
const int LATCH = 9; //Shift Register Latch Pin
const int CLK = 10; //Shift Register Clock Pin
```

```
void setup()
{
```

```

//Set pins as outputs
pinMode(SER, OUTPUT);
pinMode(LATCH, OUTPUT);
pinMode(CLK, OUTPUT);

digitalWrite(LATCH, LOW);           //Latch Low
shiftOut(SER, CLK, MSBFIRST, B10101010); //Shift Most Sig. Bit First
digitalWrite(LATCH, HIGH);          //Latch High - Show pattern
}

void loop()
{
    //Do nothing
}

```

Because the shift register will latch the values, you need to send them only once in the setup; they will then stay at those values until you change them to something else. This program follows the same steps that were shown graphically in Figure 9-3. The LATCH pin is set low, the 8 bits of data are shifted in using the `shiftOut()` function, and then the LATCH pin is set high again so that the shifted values are output on the parallel output pins of the shift register IC.

CASCADED SHIFT REGISTERS

Getting eight digital outputs from three I/O pins is a pretty good tradeoff, but what if you could get even more? You can! By daisy chaining multiple shift registers together, you could theoretically add hundreds of digital outputs to your Arduino using just three pins. If you do this, you'll probably want to use a beefier power supply than just USB, as the current requirements of a few dozen LEDs can add up very quickly.

Recall from the pin-out in Figure 9-2 that there is an unused pin called $Q_{H'}$. When the oldest value is shifted out of the shift register, it isn't discarded; it's actually sent out on that pin. By connecting the $Q_{H'}$ to the DATA pin of another shift register, and sharing the LATCH and CLOCK pins with the first shift register, you can create a 16-bit shift register that controls twice as many pins.

You can keep adding more and more shift registers, each connected to the last one, to add a crazy number of outputs to your Arduino. You can try this out by hooking up another shift register as described, and simply executing the `shiftOut()` function in your code twice. (Each call to `shiftOut()` can handle only 8 bits of information.)

Converting Between Binary and Decimal Formats

In Listing 9-1, the LED state information was written as a binary string of digits. This string helps you visualize which LEDs will be turned on and off. However, you can also write the pattern as a decimal value by converting between base2 (binary) and base10 (decimal) systems. Each bit in a binary number (starting from the rightmost, or least significant, bit) represents an increasing power of 2. Converting binary representations to decimal representations is very straightforward. Consider the binary number from earlier in the chapter, now displayed in Figure 9-5 with the appropriate decimal conversion steps.

$$\begin{array}{cccccccc}
 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
 1 \times 128 + 0 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 170
 \end{array}$$

Figure 9-5: Binary to decimal conversion

The binary value of each bit represents an incrementing power of 2. In the number in this example, bits 7, 5, 3, and 1 are high. So, to find the decimal equivalent, you add 2^7 , 2^5 , 2^3 , and 2^1 . The resulting decimal value is 170. You can prove to yourself that this value is equivalent by substituting it into the code listed earlier. Replace the `shiftOut()` line with the following:

```
shiftOut(SER, CLK, MSBFIRST, 170);
```

You should see the same result as when you used the binary notation.

Controlling Light Animations with a Shift Register

In the previous example, you built a static display with a shift register. However, you'll probably want to display more dynamic information on your LEDs. In the next two examples, you will use a shift register to control a lighting effect and a physical bar graph.

Building a “Light Rider”

The light rider is a neat effect that makes it look like the LEDs are chasing each other back and forth. You will use the same circuit that you used previously. The

`shiftOut()` function is very fast, and you can use it to update the shift register several thousand times per second. Because of this, you can quickly update the shift register outputs to make dynamic lighting animations. Here, you light up each LED in turn, “bouncing” the light back and forth between the leftmost and rightmost LEDs. Watch a demo video of this project at exploringarduino.com/content2/ch9 if you want to see what the finished project will look like before you build it.

You first want to figure out each animation state so that you can easily cycle through them. For each time step, the LED that is currently illuminated turns off, and the next light turns on. When the lights reach the end, the same thing happens in reverse. The timing diagram in Figure 9-6 shows how the lights will look for each time step and the decimal value required to turn that specific LED on.

	MSB	LSB	
t=1	○ ○ ○ ○ ○ ○ ○ ○	●	1
t=2	○ ○ ○ ○ ○ ○ ○ ○	○ ● ○	2
t=3	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ● ○	4
t=4	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ● ○ ○ ○	8
t=5	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ● ○	16
t=6	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ●	32
t=7	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ●	64
t=8	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ○ ●	128
t=9	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ●	64
t=10	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ●	32
t=11	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ●	16
t=12	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ●	8
t=13	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ●	4
t=14	○ ○ ○ ○ ○ ○ ○ ○	○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ●	2

Figure 9-6: Light rider animation steps

Recalling what you learned earlier in the chapter, convert the binary values for each light step to decimal values that can easily be cycled through. Using a `for` loop, you can cycle through an array of each of these values and shift them out to the shift register one at a time. The code in Listing 9-2 does just that.

Listing 9-2

Light rider sequence code—lightrider.ino

```
//Make a light rider animation

const int SER    =8;      //Serial Output to Shift Register
const int LATCH =9;      //Shift Register Latch Pin
const int CLK    =10;     //Shift Register Clock Pin

//Sequence of LEDs
int seq[14] = {1,2,4,8,16,32,64,128,64,32,16,8,4,2};

void setup()
{
    //Set pins as outputs
    pinMode(SER, OUTPUT);
    pinMode(LATCH, OUTPUT);
    pinMode(CLK, OUTPUT);
}

void loop()
{
    for (int i = 0; i < 14; i++)
    {
        digitalWrite(LATCH, LOW);           //Latch Low - start sending
        shiftOut(SER, CLK, MSBFIRST, seq[i]); //Shift Most Sig. Bit First
        digitalWrite(LATCH, HIGH);          //Latch High - stop sending
        delay(100);                      //Animation Speed
    }
}
```

By adjusting the value within the `delay` function, you can change the speed of the animation. Try changing the values of the `seq` array to make different pattern sequences.

NOTE To watch a demo video of the light rider, check out exploringarduino.com/content2/ch9.

Responding to Inputs with an LED Bar Graph

Using the same circuit but adding an IR distance sensor, you can make a bar graph that responds to how close you get. To mix it up a bit more, try using multiple LED colors.

The circuit diagram in Figure 9-7 shows the circuit modified with different-colored LEDs and an IR distance sensor.

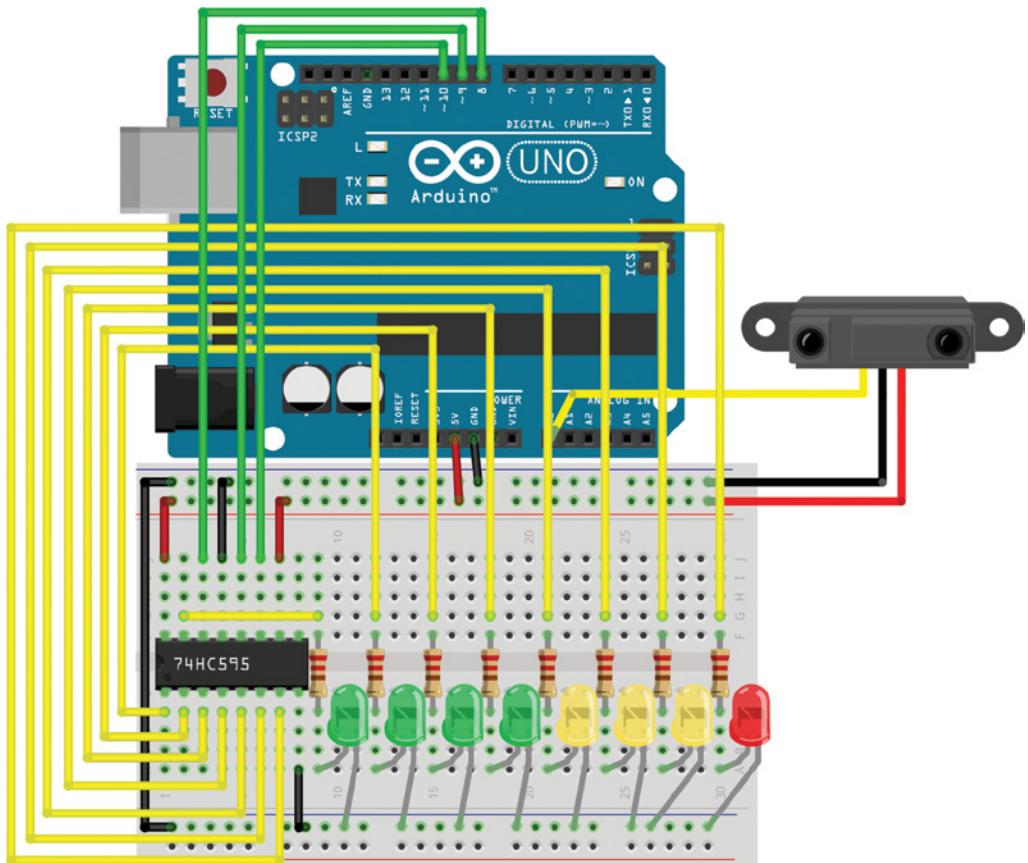


Figure 9-7: Distance-responsive bar graph

Created with Fritzing

Using the knowledge you already have from working with analog sensors and the shift register, you should be able to make thresholds and set the LEDs accordingly based on the distance reading. Figure 9-8 shows the decimal values that correspond to each binary representation of LEDs.

As you discovered in Chapter 3, “Interfacing with Analog Sensors,” the range of usable values for the IR distance sensor is not the full 10-bit range. (I found that a maximum value of around 500 worked for me, but your setup will probably differ.) Your minimum might not be zero either. It’s best to test the range of your sensor and

fill in the appropriate values. You can place all the bar graph decimal representations in an array of nine values. By mapping the IR distance sensor (and constraining it) from 0 to 500 down to 0 to 8, you can quickly and easily assign distances to bar graph configurations. The code in Listing 9-3 shows this method in action.

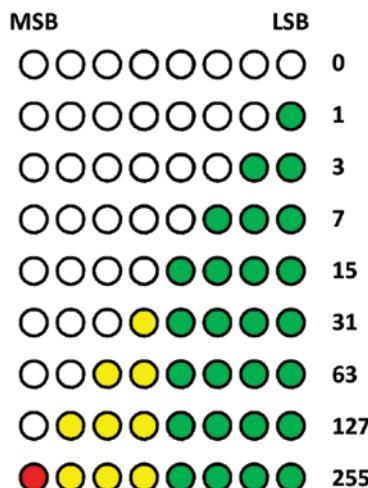


Figure 9-8: Bar graph decimal representations

Listing 9-3

Bar graph distance control-bargraph.ino

```
//A bar graph that responds to how close you are

const int SER = 8;      //Serial Output to Shift Register
const int LATCH = 9;    //Shift Register Latch Pin
const int CLK = 10;     //Shift Register Clock Pin
const int DIST = 0;     //Distance Sensor on Analog Pin 0

//Possible LED settings
int vals[9] = {0,1,3,7,15,31,63,127,255};

//Maximum value provided by sensor
int maxVal = 500;

//Minimum value provided by sensor
int minVal = 0;

void setup()
{
```

```
//Set pins as outputs
pinMode(SER, OUTPUT);
pinMode(LATCH, OUTPUT);
pinMode(CLK, OUTPUT);
}

void loop()
{
    int distance = analogRead(DIST);
    distance = map(distance, minVal, maxVal, 0, 8);
    distance = constrain(distance,0,8);

    digitalWrite(LATCH, LOW);           //Latch low - start sending
    shiftOut(SER, CLK, MSBFIRST, vals[distance]); //Send data, MSB first
    digitalWrite(LATCH, HIGH);          //Latch high - stop sending
    delay(10);                      //Animation speed
}
```

Load this program on to your Arduino and move your hand back and forth in front of the distance sensor—you should see the bar graph respond by going up and down in parallel with your hand. If you find that the graph hovers too much at “all on” or “all off,” try adjusting the `maxVal` and `minVal` values to better fit the readings from your distance sensor. To test the values you are getting at various distances, you can initialize a serial connection in the `setup()` command and call `Serial.println(distance);` right after you perform the `analogRead(DIST);` step.

NOTE To watch a demo video of the distance-responsive bar graph, visit exploringarduino.com/content2/ch9.

Summary

In this chapter, you learned the following:

- How a shift register works
- The differences between serial and parallel data transmission
- The differences between decimal and binary data representations
- How to create animations using a shift register



Communication Interfaces

Chapter 10: The I²C Bus

Chapter 11: The SPI Bus and Third-Party Libraries

Chapter 12: Interfacing with Liquid Crystal Displays

10

The I²C Bus

Parts You'll Need for This Chapter

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

Half-size or full-size breadboard

Assorted jumper wires

220Ω resistors (×8)

4.7kΩ resistors (×2)

5 mm red LED

5 mm green LEDs (×4)

5 mm yellow LEDs (×3)

SN74HC595N shift register

TC74A0-5.0VAT I²C temperature sensor

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch10

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

You've already learned how to connect both analog and digital inputs and outputs, but what about more complicated devices? The Arduino (or any microcontroller, for that matter) can expand its capabilities by interfacing with a variety of external components. Many integrated circuits (ICs) can implement standardized digital

communication protocols to facilitate communication between your microcontroller and a wide array of possible modules. This chapter explores the I²C bus (pronounced “eye squared see” or “eye two see”).

The I²C bus enables robust, high-speed, two-way communication between devices while using a minimal number of I/O pins to facilitate communication. Usual maximum speeds range from 100 kilobits per second (Kbps) up to a few megabits per second (Mbps), depending on the components and system design. An I²C bus is controlled by a master device (usually a microcontroller or microprocessor), and contains one or more slave devices that receive information from the master. In this chapter, you will learn about the I²C protocol, and you will implement it to communicate with a digital I²C temperature sensor capable of returning measurements as degree values rather than arbitrary analog values. You will build upon knowledge obtained from previous chapters by combining what you learn in this chapter to expand on earlier projects.

NOTE To watch a video tutorial about the I²C bus, visit this chapter’s content web page at exploringarduino.com/content2/ch10.

History of the I²C Bus

Understanding how a communication protocol evolved over time makes it a lot easier to understand why it works the way it does. The I²C protocol was developed by Philips Semiconductors in the early 1980s to allow for relatively low-speed communication between various integrated circuits. The protocol was standardized by the 1990s, and other companies quickly began to adopt the protocol, releasing their own compatible chips. Generically, it is known as the “two-wire” protocol because two lines are used for communication: a clock and data line. Although not all two-wire protocol devices have paid the license fee to be called I²C devices, they are all commonly referred to as I²C. This is similar to how Kleenex® is often used to refer to all tissues, even those that aren’t manufactured by Kimberly-Clark. If you find a device that says it uses the “two-wire” communication protocol, you can be fairly certain that it will work in the ways described in this chapter.

You might also find devices that utilize the SMBus (System Management Bus). Derived from the I²C standard by Intel and Duracell, the SMBus standard is very similar to I²C, but implements some slightly different electrical limits, defines a protocol for error checking, and explicitly supports an optional interrupt signal line to enable slaves to notify the master of certain events. It is usually possible to successfully mix SMBus devices and I²C devices on the same bus if you are careful to follow the requirements in all of the devices’ datasheets.

I²C Hardware Design

Figure 10-1 shows a common reference setup for an I²C communication system. Unlike digital communication systems that you've seen earlier in this book, I²C is unique in that multiple devices all share the same communication lines: a clock signal (SCL) and a bidirectional data line used for sending information back and forth between the master and the slaves (SDA). Notice, as well, that the I²C bus requires pull-up resistors on both data lines.

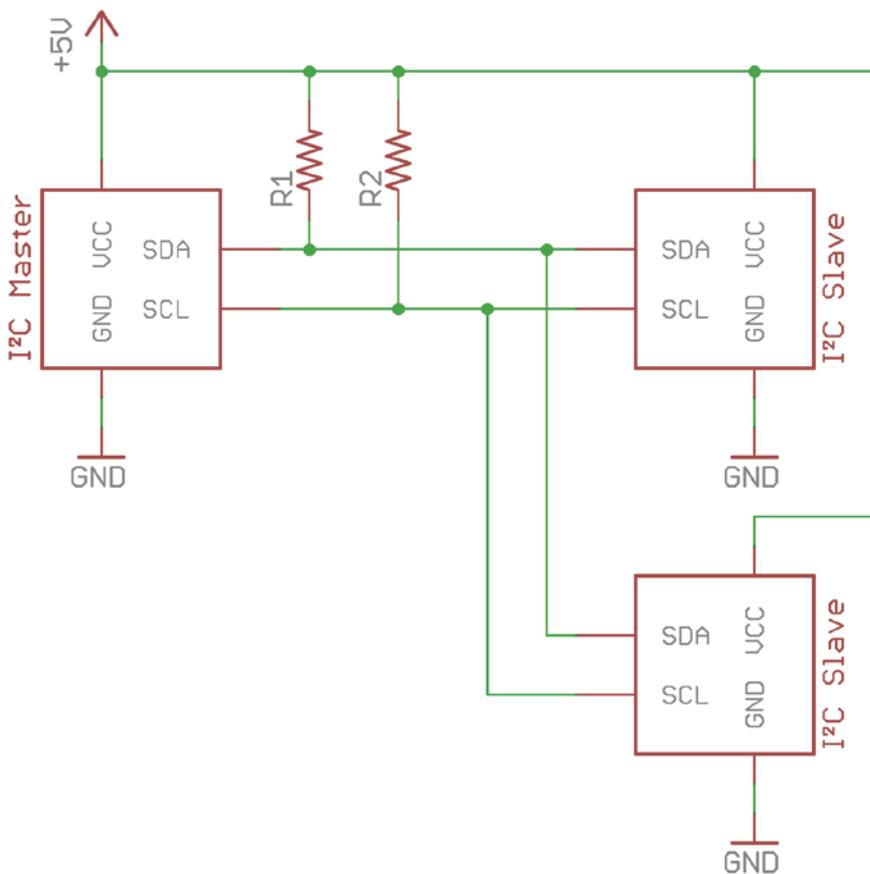


Figure 10-1: I²C reference hardware configuration

Created with EAGLE

Communication Scheme and ID Numbers

The I²C bus allows multiple slave devices to share communication lines with a single master device. In this chapter, the Arduino acts as the master device. The bus master is

responsible for initiating all communications. Slave devices cannot initiate communications; they can only respond to requests that are sent by the master device. Because multiple slave devices share the same communication lines, it's very important that only the master device be able to initiate communication. Otherwise, multiple devices may try to talk at the same time and the data will be garbled.

All commands and requests sent from the master are received by all devices on the bus. Each I²C slave device has a unique 7-bit address, or ID number. When communication is initiated by the master device, a device ID is transmitted. I²C slave devices react to data on the bus only when it is directed at their ID number. Because all the devices are receiving all the messages, each device on the I²C bus must have a unique address. Some I²C devices have selectable addresses, whereas others come from the manufacturer with a fixed address. If you want to have multiple numbers of the same device on one bus, you need to identify components that are available with different IDs.

Temperature sensors, for example, are commonly available with various pre-programmed I²C addresses because it is common to want more than one temperature sensor on a single I²C bus. In this chapter, you will use the TC74 temperature sensor. A peek at the TC74 datasheet reveals that it is available with a variety of different addresses. Figure 10-2 shows an excerpt of the datasheet. In this chapter, you will use TC74A0-5.0VAT, which is the 5V, T0-220 version of the IC with an address of 1001000.

<u>PART NO.</u>	<u>XX</u>	<u>-XX</u>	<u>X</u>	<u>XX</u>	Examples:
Device	Address Options	Supply Voltage	Operating Temperature	Package	a) TC74A0-3.3VAT: TO-220 Serial Digital Thermal Sensor
Device:	TC74: Serial Digital Thermal Sensor				b) TC74A1-3.3VAT: TO-220 Serial Digital Thermal Sensor
Address Options:	A0 = 1001 000 A1 = 1001 001 A2 = 1001 010 A3 = 1001 011 A4 = 1001 100 A5 = 1001 101 * A6 = 1001 110 A7 = 1001 111				c) TC74A2-3.3VAT: TO-220 Serial Digital Thermal Sensor
	* Default Address				d) TC74A3-3.3VAT: TO-220 Serial Digital Thermal Sensor
Output Voltage:	3.3 = Accuracy optimized for 3.3V 5.0 = Accuracy optimized for 5.0V				e) TC74A4-3.3VAT: TO-220 Serial Digital Thermal Sensor
Operating Temperature:	V = -40°C ≤ T _A ≤ +125°C				f) TC74A5-3.3VAT: TO-220 Serial Digital Thermal Sensor *
Package:	AT = TO-220-5				g) TC74A6-3.3VAT: TO-220 Serial Digital Thermal Sensor
					h) TC74A7-3.3VAT: TO-220 Serial Digital Thermal Sensor
					a) TC74A0-5.0VAT: TO-220 Serial Digital Thermal Sensor
					b) TC74A1-5.0VAT: TO-220 Serial Digital Thermal Sensor
					c) TC74A2-5.0VAT: TO-220 Serial Digital Thermal Sensor
					d) TC74A3-5.0VAT: TO-220 Serial Digital Thermal Sensor
					e) TC74A4-5.0VAT: TO-220 Serial Digital Thermal Sensor
					f) TC74A5-5.0VAT: TO-220 Serial Digital Thermal Sensor *
					g) TC74A6-5.0VAT: TO-220 Serial Digital Thermal Sensor
					h) TC74A7-5.0VAT: TO-220 Serial Digital Thermal Sensor
					* Default Address

Figure 10-2: TC74 address options

Credit: © Microchip Technology Incorporated. Used with permission.

You can purchase this particular IC with eight different ID numbers; hence, you could put up to eight of them on one I²C bus and read each of them independently. When you're writing programs to interface with this temperature sensor later in this chapter, be sure to note the ID of the device you ordered so that you send the right commands!

Other I²C chips, such as the AD7414 and AD7415 (another I²C digital temperature sensor manufactured by Analog Devices), have "address select" (AS) pins that allow you to configure the I²C address of the device. Take a look at the excerpt from the AD7414 datasheet in Figure 10-3.

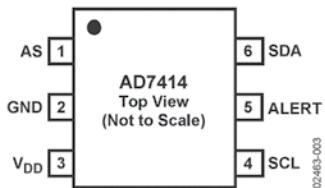


Figure 3. AD7414 Pin Configuration (SOT-23)

Table 4. I²C Address Selection

Part Number	AS Pin	I ² C Address
AD7414-0	Float	1001 000
AD7414-0	GND	1001 001
AD7414-0	V _{DD}	1001 010
AD7414-1	Float	1001 100
AD7414-1	GND	1001 101
AD7414-1	V _{DD}	1001 110
AD7414-2	N/A	1001 011
AD7414-3	N/A	1001 111

Figure 10-3: AD7414 addressing

Credit: Copyright © 2019, Analog Devices, Inc. All Rights Reserved.

As shown in Figure 10-3, the AD7414 is available in four versions: two with an AS pin and two without. The versions with AS pins can each have three possible ID numbers, depending on whether the AS pin is left disconnected, tied to VCC, or tied to GND.

PART SELECTION FROM THE PERSPECTIVE OF A PRODUCT DESIGN ENGINEER

Suppose you were designing a product that needed to leverage several temperature sensors. For instance, a circuit board with three stepper-motor driver ICs for controlling the three motors on a gantry robot might include three strategically placed temperature sensor ICs for keeping an eye on each of the stepper driver heatsink temperatures. In this scenario, does it make more sense to select three sensors that are available with three different pre-programmed addresses (like the variations of the TC74, for instance)? Or, does it make more sense to pick a single AD7414 device that can be set to three different addresses based on the state of the AS pin? Well, it depends on your design constraints.

(Continued)

(Continued)

If you're planning to manufacture a lot of these circuit boards, then economies of scale can be really important when selecting components. In the case of the AD7414, it costs \$2.59 to buy one, but it costs \$1.16 each if you buy 3,000 units at a time. If you're planning to make a few thousand of these circuit boards, you can save a lot of money by using the same component three times, instead of using three different components per board (assuming the address can be set without the use of external components).

On the other hand, what if you're making an implantable medical device that needs those three temperature sensors? In that case, product cost might not be as important to you, and you might care more about making the smallest possible circuit board. A temperature sensor with a preset address that doesn't require a dedicated address select pin might be a fraction of a square millimeter smaller, which can make all the difference when designing an ultra-small implantable product.

Hardware Requirements and Pull-Up Resistors

You may have noticed in Figure 10-1 that the standard I²C bus configuration requires pull-up resistors on both the clock and data lines. The value for these resistors depends on the slave devices and how many of them are attached. In this chapter, you will use 4.7kΩ resistors for both pull-ups; this is a fairly standard value that is specified on many datasheets.

HOW TO PICK THE RIGHT VALUE FOR A PULL-UP RESISTOR

The I²C bus uses I/O pins that are configured as “open drains” or “open collectors.” This means that the I/O pins are configured such that there is a transistor to pull the signal down to ground, but there is not a transistor to pull the signal up to the high logic level. Data on the I²C is active low, meaning that the clock and data pins are held at the high logic level by the resistor by default, and only go low (ground) when the transistor switches on and connects the line to ground. This approach offers a few advantages that make it beneficial for I²C:

- Because the data line (and sometimes the clock line) is bidirectional (either the master or any of the slaves can control it by pulling it low), the open drain guarantees that a communications glitch can never result in one bus device

enabling its high transistor while another bus device's low transistor is enabled. If this were to happen, it would create a short circuit from the high voltage to ground, potentially damaging the circuit. For this reason, you'll find that open-drain topologies are often employed when multiple drives are connected to the same signal line.

- It may enable devices with different operating voltages to communicate without any additional logic level conversion. If you have a 3.3V microcontroller that is talking to a 5.0V sensor, you can pull the voltage up to 5.0V (assuming the microcontroller is rated to handle this) with the pull-up resistor. These two devices will then be able to talk because the logic high-voltage threshold will be detectable by both. With a traditional "push-pull" I/O pin configuration, the microcontroller would need to use a transistor to drive the logic high, but it would only be able to drive it up to 3.3V because that is its operating voltage. That voltage might not be high enough to be registered as a logic high by the 5.0V sensor.

The value of the resistor depends on a number of factors. As a designer, you have to find the best trade-off between all of these things:

- If the pull-up resistor value is too low (strong pull-up), then the open-drain drivers might not be strong enough to pull the bus low. (This is a function of the current-sinking capabilities of the I/O transistors.)
- If the pull-up resistor value is too high (weak pull-up), then very little current will flow through the resistors and it will take a long time for the bus voltage to return to the high logic state after being pulled low. This will limit the maximum speed at which the I²C can be operated. This is also impacted by the capacitance of the bus. (Longer wires and more devices on the bus will increase the capacitance.) In Chapter 17, "Wi-Fi Connectivity and 'The Cloud,'" you will see an example of a setup where weak (10KΩ) resistors on an I²C bus with long wires can cause issues due to slow signal rise time.
- Lower-value resistors will allow the bus to operate faster, but will also consume more power when the bus is pulled low. (This is important for mobile, battery-powered devices where every microamp of power consumption can make a difference.)

Seems like a lot to keep track of, right? That's why following a simple rule of thumb for basic Arduino prototyping is okay. For most Arduino applications, a pull-up between 2KΩ and 10KΩ will work just fine. When in doubt, consult the datasheet of the slave device. Once you start applying your skills towards designing a mass-produced

(Continued)

(Continued)

product, you can consider using one of many I²C resistor calculators that you can find on the web to help you pick the perfect value. Then, you can use an oscilloscope to check the performance of the bus directly, and check for bus signals not making it all the way to the high logic level before the next low transition. (This indicates that the resistor value needs to be decreased, the bus capacitance needs to be reduced, or the operating speed of the bus needs to be reduced.)

Communicating with an I²C Temperature Probe

The steps for communicating with different I²C devices vary, based on the requirements of the specific device. Thankfully, you can use the Arduino I²C library to abstract away most of the difficult timing work. In this section, you will talk to the I²C temperature sensor described earlier in the chapter. You will learn how to interpret the datasheet information as you progress so that you can apply these concepts to other I²C devices with relative ease.

The basic steps for controlling any ICC device are as follows:

1. The master sends a start bit.
2. The master sends a 7-bit slave address of the device it wants to talk to.
3. The master sends a read (1) or write (0) bit, depending on whether it wants to write data into an I²C device's registers or it wants to read from one of the I²C device's registers.
4. The slave responds with an “acknowledge” or ACK bit (a logic low).
5. In write mode, the master sends 1 byte of information at a time, and the slave responds with ACKs. In read mode, the master receives 1 byte of information at a time and sends an ACK to the slave after each byte.
6. When communication has been completed, the master sends a stop bit.

Setting Up the Hardware

To confirm that your first program works as expected, you can use the serial monitor to print out temperature readings from an I²C temperature sensor to your computer. Because this is a digital sensor, it prints the temperature in degrees. Unlike the temperature sensors that you used in previous chapters, you do not have to worry about converting an analog reading to an actual temperature. How convenient! Now, wire a temperature sensor to the Arduino as shown in Figure 10-4.

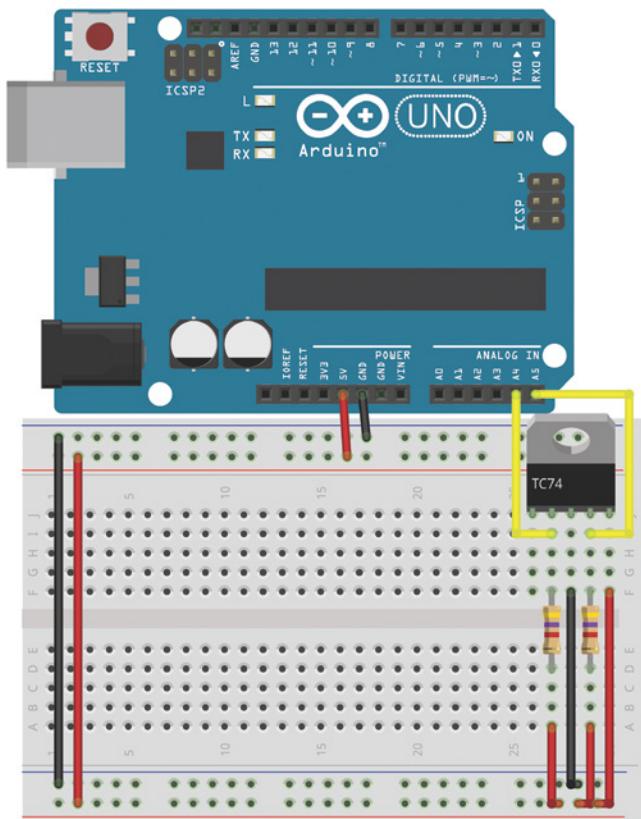


Figure 10-4: Temperature sensor

Created with Fritzing

Note that the SDA and SCL pins are wired to pins A4 and A5, respectively. Recall from earlier in the chapter that the SDA and SCL are the two pins used for communicating with I²C devices—they carry data and clock signals, respectively. You've already learned about multiplexed pins in previous chapters. On the Arduino Uno (and other ATmega 328-based Arduinos), pins A4 and A5 are multiplexed between the analog-to-digital converter (ADC) and the hardware I²C interface. When you initialize the Wire library in your code, those pins connect to the ATmega's internal I²C controller, enabling you to use the wire library to talk to I²C devices via those pins. When using the Wire library, you cannot use pins A4 and A5 as analog inputs because they are reserved for communication with I²C devices. On the latest revisions of the Arduino boards, there are also dedicated I²C pins above the AREF pin (they are electrically connected to the A4/A5 pins and are functionally identical). You can connect to those pins if you prefer.

Referencing the Datasheet

Next, you need to write the software that instructs the Arduino to request data from the I²C temperature sensor. The Arduino Wire library makes this fairly easy. To use it properly, you need to know how to read the datasheet to determine the communication scheme that this particular chip uses. Let's dissect the communication scheme presented in the datasheet, using what you already know about how I²C works. Consider the diagrams from the datasheet shown in Figure 10-5 and Figure 10-6.

You can both read from and write to this IC, as shown in the datasheet in Figure 10-5. The TC74 has two registers: one that contains the current temperature in Celsius and one that contains configuration information about the chip (including standby state and data-ready state). Table 4-1 of the datasheet shows this. You don't need to mess with the configuration information; you only want to read the temperature from the device. Tables 4-3 and 4-4 in Figure 10-6 show how the temperature information is stored within the 8-bit data register.

Write Byte Format

S	Address	WR	ACK	Command	ACK	Data	ACK	P
	7 Bits			8 Bits		8 Bits		

Slave Address

Command Byte: selects which register you are writing to

Data Byte: data goes into the register set by the command byte

Read Byte Format

S	Address	WR	ACK	Command	ACK	S	Address	RD	ACK	Data	NACK	P
	7 Bits			8 Bits			7 Bits			8 Bits		

Slave Address

Command Byte: selects which register you are reading from

Slave Address: repeated
due to change in data-
flow direction

Data Byte: reads from the register set by the command byte

Receive Byte Format

Receive Byte Format						
S	Address	RD	ACK	Data	NACK	P
	7 Bits			8 Bits		

S ≡ START Condition

P = STOP Condition

Shaded = Slave Transmission

Data Byte: reads data from the register commanded by the last Read Byte or Write Byte transmission

Figure 10-5: TC74 sensor communication scheme

Credit: © Microchip Technology Incorporated. Used with permission.

TC74

4.0 REGISTER SET AND PROGRAMMER'S MODEL

TABLE 4-1: COMMAND BYTE DESCRIPTION (SMBUS/I²C READ_BYTE AND WRITE_BYTE)

Command	Code	Function
RTR	00h	Read Temperature (TEMP)
RWCR	01h	Read/Write Configuration (CONFIG)

TABLE 4-2: CONFIGURATION REGISTER (CONFIG); 8 BITS, READ/ WRITE)

Bit	POR	Function	Type	Operation
D[7]	0	STANDBY Switch	Read/ Write	1 = standby, 0 = normal
D[6]	0	Data Ready *	Read Only	1 = ready 0 = not ready
D[5]-D[0]	0	Reserved - Always returns zero when read	N/A	N/A

Note 1: *DATA_RDY bit RESET at power-up and SHDN enable.

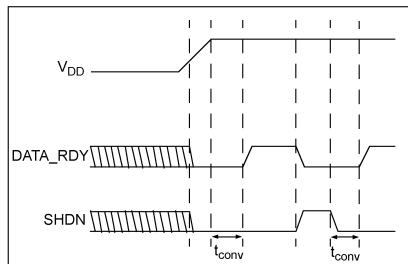


FIGURE 4-1: DATA_RDY, SHDN Operation Logic Diagram.

4.1 Temperature Register (TEMP), 8 Bits, READ ONLY

The binary value (2's complement format) in this register represents temperature of the onboard sensor following a conversion cycle. The registers are automatically updated in an alternating manner.

TABLE 4-3: TEMPERATURE REGISTER (TEMP)

D[7]	D[6]	D[5]	D[4]	D[3]	D[2]	D[1]	D[0]
MSB	X	X	X	X	X	X	LSB

In temperature data registers, each unit value represents one degree (Celsius). The value is in 2's complement binary format such that a reading of 0000 0000b corresponds to 0°C. Examples of this temperature to binary value relationship are shown in Table 4-4.

TABLE 4-4: TEMPERATURE-TO-DIGITAL VALUE CONVERSION (TEMP)

Actual Temperature	Registered Temperature	Binary Hex
+130.00°C	+127°C	0111 1111
+127.00°C	+127°C	0111 1111
+126.50°C	+126°C	0111 1110
+25.25°C	+25°C	0001 1001
+0.50°C	0°C	0000 0000
+0.25°C	0°C	0000 0000
0.00°C	0°C	0000 0000
-0.25°C	-1°C	1111 1111
-0.50°C	-1°C	1111 1111
-0.75°C	-1°C	1111 1111
-1.00°C	-1°C	1111 1111
-25.00°C	-25°C	1110 0111
-25.25°C	-26°C	1110 0110
-54.75°C	-55°C	1100 1001
-55.00°C	-55°C	1100 1001
-65.00°C	-65°C	1011 1111

4.2 Register Set Summary

The TC74 register set is summarized in Table 4-5. All registers are 8 bits wide.

TABLE 4-5: TC74 REGISTER SET SUMMARY

Name	Description	POR State	Read	Write
TEMP	Internal Sensor Temperature (2's Complement)	0000 0000b ⁽¹⁾	✓	N/A
CONFIG	CONFIG Register	0000 0000b	✓	✓

Note 1: The TEMP register will be immediately updated by the A/D converter after the DATA_RDY Bit goes high.

The “Read Byte Format” section of Figure 10-5 outlines the process of reading the temperature from the TC74:

1. Send to the device’s address in write mode and write a 0 to indicate that you want to read from the data register.
2. Send to the device’s address in read mode and request 8 bits (1 byte) of information from the device.
3. Confirm that all 8 bits (1 byte) of temperature information were received.

Now that you understand the steps necessary to request information from this device, you should better understand how similar I²C devices would also work. When in doubt, search the web for code examples that show how to connect your Arduino to various I²C devices. Next, you will write the code that executes the three steps outlined earlier.

Writing the Software

Arduino’s I²C communication library is called the Wire library. After you insert it at the top of your sketch, you can easily write to and read from I²C devices. As a first step for your I²C temperature sensor system, load up the code in Listing 10-1, which takes advantage of the functions built in to the Wire library. See whether you can match up various `Wire` commands in the code with the steps outlined in the previous section.

Listing 10-1

I²C temperature sensor printing code—`read_temp.ino`

```
//Reads Temp from I2C temperature sensor
//and prints it on the serial port

//Include Wire I2C library
#include <Wire.h>
int temp_address = 72; //1001000 written as decimal number

void setup()
{
    //Start serial communication at 9600 baud
    Serial.begin(9600);

    //Create a Wire object
    Wire.begin();
}
```

```
void loop()
{
    //Send a request
    //Start talking to the device at the specified address
    Wire.beginTransmission(temp_address);
    //Send a bit asking for register zero, the data register
    Wire.write(0);
    //Complete Transmission
    Wire.endTransmission();

    //Read the temperature from the device
    //Request 1 Byte from the specified address
    int returned_bytes = Wire.requestFrom(temp_address, 1);

    //If no data was returned, then something is wrong.
    if (returned_bytes == 0)
    {
        Serial.println("I2C Error"); //Print an error
        while(1); //Halt the program
    }

    // Get the temp and read it into a variable
    int c = Wire.read();

    //Do some math to convert the Celsius to Fahrenheit
    int f = round(c*9.0/5.0 +32.0);

    //Send the temperature in degrees C and F to the serial monitor
    Serial.print(c);
    Serial.print("C ");
    Serial.print(f);
    Serial.println("F");

    delay(500);
}
```

Consider how the commands in this program relate to the previously mentioned steps. `Wire.beginTransmission()` starts the communication with a slave device with the given ID. Next, the `Wire.write()` command sends a 0, indicating that you want to be reading from the temperature register. You then send a stop bit with the `Wire.endTransmission()` command to indicate that you have finished writing to the device. Next, the master reads from the slave I²C device. The `Wire.requestFrom()` command asks for a certain amount of data (1 byte) and then returns the number of bytes that were actually received. This is stored into a variable called `returned_bytes`. This value is then checked; if it is zero, then the I²C devices did not return any data. This generally implies a hardware problem, such as the sensor not being wired up properly. Thus, an error is printed to the serial monitor and the

program enters an endless wait loop if this condition is triggered. Assuming that data was received back from the I²C device, the 8-bit value is read into an integer variable with a `Wire.read()` command.

The program in Listing 10-1 also handles converting the Celsius temperature to Fahrenheit, for those who are not metrically inclined. You can find the formula for this conversion with a simple web search. I've chosen to round the result to a whole number.

Now, run the preceding code on your Arduino and open up the serial monitor on your computer. You should see output similar to Figure 10-7. If you receive an error, then your sensor is not properly wired to your Arduino.

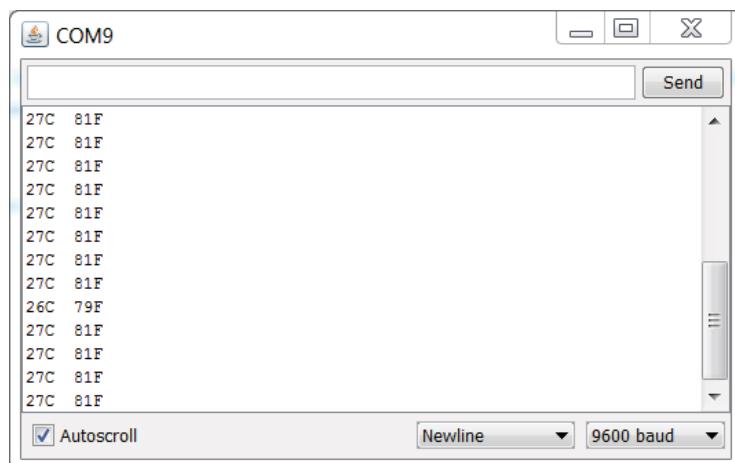


Figure 10-7: I²C temperature sensor serial output

Combining Shift Registers, Serial Communication, and I²C Communications

Now that you have a simple I²C communication scheme set up with serial printing, you can apply some of your knowledge from previous chapters to do something more interesting. You will use the shift register graph circuit from Chapter 9, “Shift Registers,” along with a Processing desktop sketch to visualize temperature in the real world and on your computer screen.

Building the Hardware for a Temperature Monitoring System

First things first: get the system wired up. You’re essentially just combining the shift register circuit from the previous chapter with the I²C circuit from this chapter. Your setup should look like Figure 10-8.

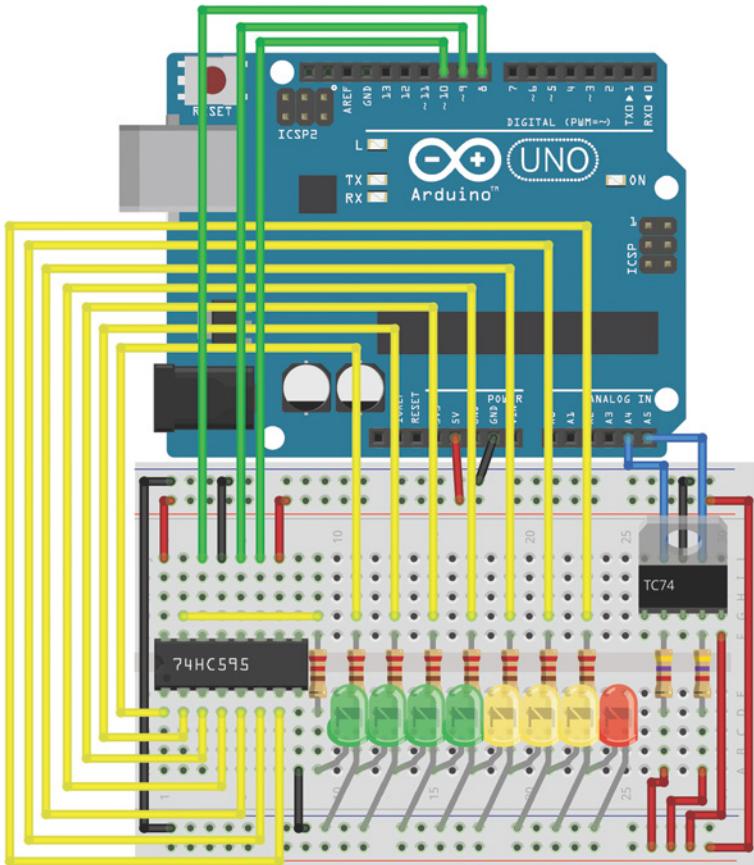


Figure 10-8: I²C temperature sensor with a shift register bar graph

Created with Fritzing

Modifying the Embedded Program

You need to make two adjustments to the previous Arduino program to make serial communication with Processing easier, and to implement the shift register functionality. First, modify the temperature printing statements in the program you just wrote to look like this:

```
Serial.print(c);
Serial.print("C,");
Serial.print(f);
Serial.print("F.");
```

Modify the `Serial.println("I2C Error");` error condition to also print a similarly formatted message (two values separated by a comma, and ending with a period):

```
Serial.print("Err,Err.");
```

Processing needs to parse the Celsius and Fahrenheit temperature data. By replacing the spaces and carriage returns with commas and periods, you can easily look for these delimiting characters and use them to parse the data.

Next, you need to add the shift register code from the previous chapter, and map the LED levels appropriately to the temperature range that you prefer. If you a need a refresher on the shift register code that you previously wrote, take another look at Listing 9-3; much of the code from that program will be reused here, with a few small tweaks. To begin, change the total number of light variables from nine to eight. With this change, you always leave one LED on as an indication that the system is working. (The 0 value is eliminated from the array.) You need to accommodate for that change in the variable value mapping, and you need to map a range of temperatures to LED states. Check out the complete code sample in Listing 10-2 to see how that is done. I chose to make my range from 24°C to 31°C (75°F to 88°F), but you can choose any range.

Listing 10-2

I²C temperature sensor code with shift register LEDs and serial communication-temp_unit.ino

```
//Reads temp from I2C temperature sensor
//show it on the LED bar graph, and show it in Processing

//Include Wire I2C library
#include <Wire.h>

const int SER    =8; //Serial Output to Shift Register
const int LATCH  =9; //Shift Register Latch Pin
const int CLK    =10; //Shift Register Clock Pin

int temp_address = 72;

//Possible LED settings
int vals[8] = {1,3,7,15,31,63,127,255};

void setup()
{
    //Instantiate serial communication at 9600 bps
    Serial.begin(9600);

    //Create a Wire Object
    Wire.begin();
```

```
//Set shift register pins as outputs
pinMode(SER, OUTPUT);
pinMode(LATCH, OUTPUT);
pinMode(CLK, OUTPUT);
}

void loop()
{
    //Send a request
    //Start talking to the device at the specified address
    Wire.beginTransmission(temp_address);
    //Send a bit asking for register zero, the data register
    Wire.write(0);
    //Complete Transmission
    Wire.endTransmission();

    //Read the temperature from the device
    //Request 1 Byte from the specified address
    int returned_bytes = Wire.requestFrom(temp_address, 1);

    //If no data was returned, then something is wrong.
    if (returned_bytes == 0)
    {
        Serial.print("Err, Err.");
        //Print an error
        while(1);
    }

    // Get the temp and read it into a variable
    int c = Wire.read();

    //Map the temperatures to LED settings
    int graph = map(c, 24, 31, 0, 7);
    graph = constrain(graph,0,7);

    digitalWrite(LATCH, LOW);           //Latch low - start sending data
    shiftOut(SER, CLK, MSBFIRST, vals[graph]); //Send data, MSB first
    digitalWrite(LATCH, HIGH);          //Latch high - stop sending data

    //Do some math to convert the Celsius to Fahrenheit
    int f = round(c*9.0/5.0 +32.0);

    Serial.print(c);
    Serial.print("C,");
    Serial.print(f);
    Serial.print("F.");

    delay(500);
}
```

After loading this code on to your Arduino, you can see the LEDs changing color with the temperature. Try squeezing the temperature sensor with your fingertips to make the temperature go up; you should see a response in the LEDs. Next, you will write a Processing sketch that displays the temperature value on the computer in an easy-to-read format.

Writing the Processing Sketch

At this point, your Arduino is already transmitting easy-to-parse data to your computer. All you need to do is write a Processing program that can interpret it and display it in an attractive way.

Because you'll be updating text in real time, you need to first learn how to load fonts into Processing. Open Processing to create a new, blank sketch. Save the sketch before continuing. Then, navigate to Tools ➤ Create Font. You see a screen that looks like Figure 10-9.

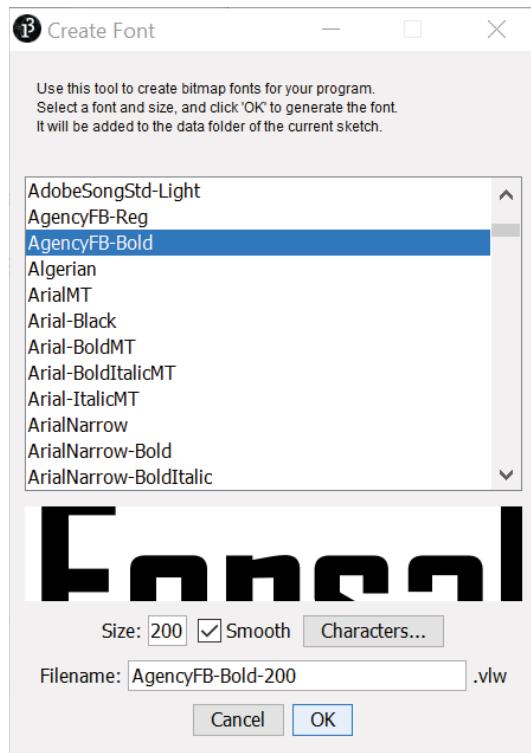


Figure 10-9: Processing font creator

Pick your favorite font and specify a size. (I recommend a size of around 200 for this exercise.) When you're done, click OK. The font is automatically generated and added to the *data* subfolder of your Processing sketch folder. The Processing sketch needs to accomplish a few things:

- Generate a graphical window on your computer showing the temperature in both Celsius and Fahrenheit.
- Read the incoming data from the serial port, parse it, and save the values to local variables that can be displayed on the computer.
- Continually update the display with the new values that are received over the serial link.

Copy the code from Listing 10-3 into your Processing sketch and adjust the serial port name to the right value for your computer and the name of the font you created. Then, ensure your Arduino is connected and click the Run icon to watch the magic! Don't forget to ensure that the serial monitor in the Arduino IDE is closed, first—only one program can access the serial port at a time.

Listing 10-3

Processing sketch for displaying temperature values–display_temp.pde

```
//Displays the temperature recorded by an I2C temp sensor

import processing.serial.*;
Serial port;
String temp_c = "";
String temp_f = "";
String data = "";
int index = 0;
PFont font;

void setup()
{
    size(400,400);
    //Change "COM9" to the name of the serial port on your computer
    port = new Serial(this, "COM9", 9600);
    port.bufferUntil('.');
    //Change the font name to reflect the name of the font you created
    font = loadFont("AgencyFB-Bold-200.vlw");
    textAlign(CENTER);
    textFont(font, 200);
}

void draw()
{
```

```
background(0,0,0);
fill(46, 209, 2);
text(temp_c, 70, 175);
fill(0, 102, 153);
text(temp_f, 70, 370);
}

void serialEvent (Serial port)
{
  data = port.readStringUntil('.');
  data = data.substring(0, data.length() - 1);

  //Look for the comma between Celsius and Fahrenheit
  index = data.indexOf(",");
  // fetch the C Temp
  temp_c = data.substring(0, index);
  //Fetch the F temp
  temp_f = data.substring(index+1, data.length());
}
```

As in previous Processing examples that you've run, the sketch starts by importing the serial library and setting up the serial port. In `setup()`, you are defining the size of the display window, loading the font you just created, and setting up the serial port to buffer until it receives a period. `draw()` fills the background in black and prints out the Celsius and Fahrenheit values in two colors. With the `fill()` command, you are telling Processing to make the next element it adds to the screen that color (in RGB values). `serialEvent()` is called whenever the `bufferUntil()` event is triggered. It reads the buffer into a string, and then breaks it up based on the location of the comma. The two temperature values are stored in variables that are printed out in the application window.

When you execute the program, the output should look similar to Figure 10-10.

When you squeeze the sensor, the Processing display should update, and the lights on your board should illuminate. If you see “Err Err” on the processing display, that means that your temperature sensor is not returning a value when queried over I²C—check the wiring.

NOTE To watch a demo video of the temperature monitoring hardware and Processing system, check out exploringarduino.com/content2/ch10.

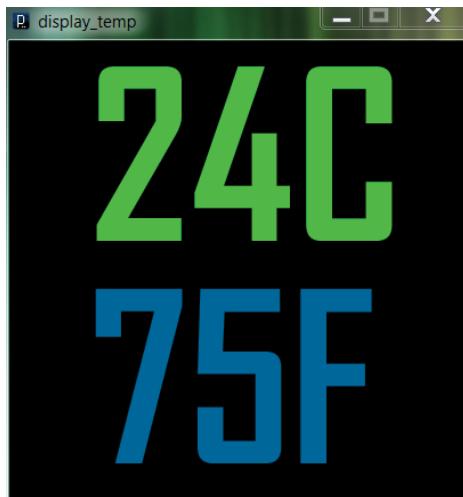


Figure 10-10: Processing temperature display

Summary

In this chapter, you learned the following:

- I²C uses two data lines to enable digital communication between the Arduino and multiple slave devices (as long as they have different addresses).
- The Arduino Wire library can be used to facilitate communication with I²C devices connected to the Arduino's SCL and SDA pins.
- I²C communication can be employed alongside shift registers and serial communication to create more complex systems.
- You can create fonts in Processing to generate dynamically updating onscreen displays.
- Processing can be used to display parsed serial data obtained from I²C devices connected to the Arduino.

11

The SPI Bus and Third-Party Libraries

Parts You'll Need for This Chapter

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

Half-size or full-size breadboard

Assorted jumper wires

220 Ω resistors (x4)

5mm common-anode RGB LED

Piezo buzzer

Adafruit LIS3DH breakout board

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch11

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

You've already learned about two important digital communication methods that are available to you on the Arduino: the I²C bus and the serial UART bus. In this chapter, you will learn about a third digital communication method supported by the Arduino hardware: the Serial Peripheral Interface bus, or SPI (often pronounced "spy") bus for short.

Unlike the I²C bus, the SPI bus uses separate lines for sending and receiving data, and it employs an additional line for selecting which slave device you are talking to. This adds extra wires, but also eliminates the problem of needing different slave device addresses. The SPI bus is generally easier to get running than I²C and can run at a faster

Exploring Arduino®: Tools and Techniques for Engineering Wizardry, Second Edition.

Jeremy Blum.

© 2020 John Wiley & Sons, Inc. Published 2020 by John Wiley & Sons, Inc.

speed. In this chapter, you will learn about using the Arduino's built-in SPI hardware to communicate with a digital accelerometer. You will learn how to find and install third-party libraries that make it easier to interface with complex hardware, and you will use the accelerometer to control both LED brightness and sound effects, allowing you to make a motion-responsive audiovisual instrument.

NOTE The first edition of this book leveraged an MCP4231 digital SPI potentiometer to explain how the SPI bus works. These chips are getting harder to find, and don't lend themselves to making mentally stimulating projects in the ways that an accelerometer can. If you want to learn how to interface with a digital potentiometer, check out a tutorial video on this topic at blum.fyi/spi-digipot-tutorial.

Overview of the SPI Bus

Originally created by Motorola, the SPI bus is a full-duplex serial communication standard that enables simultaneous bidirectional communication between a master device and one or more slave devices. Because the SPI protocol does not follow a formal standard, it is common to find SPI devices that operate in a slightly different way; for example, the number of transmitted bits may differ, or the slave select line might be omitted, among other things. This chapter focuses on interfacing with devices that implement the most common SPI interfaces (which are the ones that are supported by the Arduino IDE, and the third-party libraries that you'll use).

WARNING Bear in mind that SPI implementations can vary, so reading the data-sheet of the device you plan to use is extremely important.

SPI buses can act in four main ways, which depend on the requirements of your device. SPI devices are often referred to as *slave devices*. SPI devices are synchronous, meaning that data is transmitted in sync with a serial clock line (SCLK). Data can be shifted into the slave device on either the rising or falling edge of the clock signal (called the *clock phase*), and the SCLK default state can be set to either high or low (called the *clock polarity*).

Because there are two options for the clock phase and two options for the clock polarity, you can configure the SPI bus in a total of four ways. Table 11-1 shows each of the possibilities and the modes that they correspond to in the Arduino SPI library. When you use a library that is explicitly designed to interface with a particular device, the library will

Table 11-1: SPI Communication Modes

SPI Mode	Clock Polarity	Clock Phase
Mode 0	Low at Idle	Data Capture on Clock Rising Edge
Mode 1	Low at Idle	Data Capture on Clock Falling Edge
Mode 2	High at Idle	Data Capture on Clock Falling Edge
Mode 3	High at Idle	Data Capture on Clock Rising Edge

generally operate automatically in the correct mode. Still, understanding the low-level communication hurdles will help you troubleshoot potential issues down the road.

SPI Hardware and Communication Design

The SPI system setup is relatively simple. Three pins are used for communicating between a master and all slave devices:

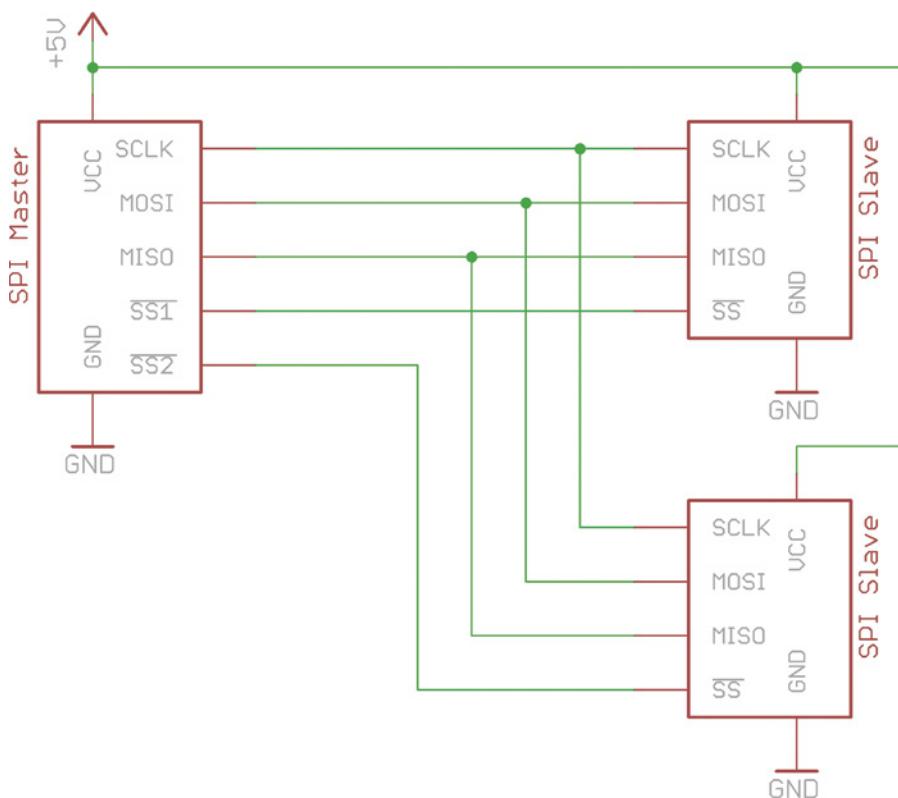
- Shared/Serial Clock (SCLK)
- Master Out/Slave In (MOSI)
- Master In/Slave Out (MISO)

Each slave device also requires an additional slave select (SS) pin. Hence, the total number of I/O pins required on the master device will always be $3 + n$, where n is the number of slave devices. Figure 11-1 shows an example SPI system with two slave devices.

Hardware Configuration

Four data lines, at a minimum, are present in any SPI system. Additional SS lines are added for each slave device appended to the network. Before you learn how to actually send and receive data to and from an SPI device, you need to understand what these I/O lines do and how they should be wired. Table 11-2 describes these lines.

Unlike with the I²C bus, pull-up resistors are not required (the I/O pins are all push/pull instead of open-drain), and communication is fully bidirectional. To wire an SPI device to the Arduino, all you have to do is connect the MOSI, MISO, SCLK, and SS. Don't forget to be mindful of voltage and logic levels; if you are using a 5V Arduino (like the Uno), then you should ensure that your SPI slave device can also talk at 5V logic levels.

**Figure 11-1:** SPI reference hardware configuration

Created with Eagle

Table 11-2: SPI Communication Lines

SPI Communication Line	Description
MOSI	Used for sending serial data from the master device to a slave device.
MISO	Used for sending serial data from a slave device to the master device.
SCLK	The signal by which the serial data is synchronized with the receiving device, so it knows when to read the input.
SS	A line indicating slave device selection. Pulling it low means you are speaking with that slave device. Generally, only one SS line on a bus should be pulled low at a time.

NAMING CONVENTIONS

Because SPI is not a universal standard, some devices and manufacturers may use different names for the SPI communication lines. Slave select is sometimes referred to as *chip select* (CS); serial clock is sometimes just called *clock* (CLK or SCK); and MOSI and MISO pins on slave devices are sometimes abbreviated to *serial data in* (SDI) and *serial data out* (SDO). You will see variations of all these conventions used throughout this chapter, as different manufacturers denote these differently.

Communication Scheme

The SPI communication scheme is synced with the clock signal and depends on the state of the SS line. Because all devices on the bus share the MOSI, MISO, and SCLK lines, all commands sent from the master arrive at each slave. The SS pin tells the slave whether it should ignore this data or respond to it. Importantly, this means that you must make sure to only have one SS pin set low (the active mode) at a time in any program that you write.

The basic process for communicating with an SPI device is as follows:

1. Set the SS pin low for the device you want to communicate with.
2. Toggle the clock line up and down at a speed less than or equal to the transmission speed supported by the slave device.
3. For each clock cycle, send 1 bit on the MOSI line, and receive 1 bit on the MISO line.
4. Continue until transmitting or receiving is complete, and stop toggling the clock line.
5. Return the SS pin to high state.

Comparing SPI to I²C and UART

Many kinds of devices, including accelerometers, digital potentiometers, and displays, are available in both SPI and I²C versions. (The accelerometer that you'll use later in this chapter supports both SPI and I²C connections.) If you want to have multiple Arduinos talk to each other, you can devise a protocol that will work over SPI, I²C, or UART. So, how do you decide? Table 11-3 lists some of the trade-offs between SPI, I²C, and UART. Ultimately, the one you choose to use will depend on what you believe is easier to implement, and best suited for your situation.

Table 11-3: SPI, I²C, and UART Comparison

SPI	I ² C	UART
Can operate at the highest speeds.	Maximum speed is highly dependent upon physical characteristics of the bus (length, number of devices, pull-up strength, and so on).	Requires baud rate to be agreed upon by both devices in advance of starting communication.
Is generally easier to work with than I ² C.	Requires only two communication lines.	Effectively uses zero protocol overhead—very simple to implement.
No pull-up resistors needed.	Can support communication between devices operating from different voltage rails.	No predefined master/slave—it is up to you to define the protocol.
Number of slave devices is limited only by number of available SS pins on master.	Number of slave devices is limited by availability of chips with particular slave addresses.	Cannot easily support multiple slave devices.
Has built-in Arduino hardware and software support.	Has built-in Arduino hardware and software support.	Has built-in Arduino hardware and software support.

Communicating with an SPI Accelerometer

Now that you've got all the basics down, it's time to actually implement what you've learned. You'll start by using a digital 3-axis accelerometer to build an orientation sensor. Specifically, you'll use the STMicroelectronics LIS3DH 3-axis accelerometer to control a red/green LED that will change color to indicate device orientation. Once you've got that working, you'll expand upon it with sound effects to build an audiovisual musical instrument that responds to movement.

DEVICE MINIATURIZATION AND SMT

Developing integrated circuits is an expensive and time-intensive business! That goes double for MEMS (Micro Electro Mechanical Systems). An accelerometer is an example of a MEMS device—it includes both a silicon wafer and microscopic mechanical elements that are used to sense acceleration.

To complete the examples in this chapter, the recommended product is actually a breadboard-friendly “breakout” of the LIS3DH accelerometer. Breakout boards adapt small, surface-mounted chips to be plugged into a breadboard. Like

an increasing number of devices today, the LIS3DH is only available in an SMT (Surface Mount Technology) package. This means that STMicroelectronics does not make the product in a form factor that can be readily inserted in a breadboard. Instead, they've actually worked hard to miniaturize the integrated circuit as much as possible; this chip measures only 3 mm by 3 mm, or about half the size of an average grain of rice.

While it is still possible to find devices like shift registers and H-bridges in breadboard-sized Dual In-line Packages (DIPs), more modern marvels like MEMS devices are exclusively made in SMT packages. This is because older chips (like shift registers and H-bridges) are still needed to repair and maintain older equipment, and because shrinking silicon fabrication technology enables new chips to consume fewer raw materials, thus costing less money. But, it's likely that at some point, the DIPs of many devices will be phased out of production as well. Economically speaking, there is little incentive for device manufacturers to create new chips in large packages because they are primarily targeted at our ever-shrinking electronic devices—smartphones, IoT devices, and smart watches, for example.

This is a double-edged sword, though. Although this has made DIPs harder to find, it has also dramatically reduced the cost of incredibly sophisticated devices. LIS3DH accelerometers sell for roughly US\$1.50 each (or approximately half that price if you're buying them in large quantities). You have the smartphone industry to thank for these low-cost sensors. Because every smartphone on earth now has an accelerometer inside, the number of these devices has skyrocketed, and the cost has plummeted.

Now, you might be excited that so many sophisticated integrated circuits are available so cheaply, but discouraged to learn that they are all in tiny SMT packages that are basically impossible to use without a custom-printed circuit board. But fear not! Open source hardware companies like Seeed, Pololu, Parallax, Adafruit, SparkFun, and others make breakout boards that “convert” these popular SMT products into breadboard-friendly form factors. They also often add useful peripheral features like voltage regulators and level shifters. The Adafruit breakout board for the LIS3DH is the recommended product for the exercises in this chapter.

What Is an Accelerometer?

Before you get an accelerometer wired up, it's worth understanding what it is and how it works. The name is pretty self-explanatory: it measures acceleration. An accelerometer is one of three common positioning sensors that you might find in a modern smartphone (the other two being a gyroscope and a magnetometer). Gyroscopes measure rotational motion, magnetometers measure magnetic fields (including that of the

Earth), and accelerometers measure linear acceleration. Paired with GPS data, these sensors enable the mapping app on your smartphone to function so intuitively.

One of the accelerometer's roles is to determine a device's orientation relative to the Earth (which is what you'll use it for in this chapter). Thanks to gravity, the z-axis of an accelerometer (the one pointing away from the ground, and towards the sky) will always experience a constant linear acceleration of 9.8 m/s^2 . When in freefall, the acceleration experienced by the z-axis of the accelerometer will decrease to 0 m/s^2 .

Laptops with mechanical hard drives use an accelerometer to park the read/write heads of the drive in the event that a freefall is detected. This prevents the read/write head from crashing into the spinning disk when the laptop impacts the ground. If you've ever dropped a laptop with a mechanical drive and still successfully booted it afterwards, you have an accelerometer to thank!

Figure 11-2 shows a simplified diagram of how an accelerometer works. The individual structures shown are generally on the order of a few microns in size (a human hair is roughly 100 microns thick). They are etched from silicon using processes that are similar to those used to create silicon integrated circuits. In the figure, the red elements are fixed in place, and the blue mass is permitted to wiggle back and forth on the green spring elements.

Similar to how accelerating your car will push you into your seat, accelerating this MEMS accelerometer will push the movable mass in the direction opposite of the acceleration. This, in turn, causes the distance between the immovable plates and the moving plates to increase or decrease. These plates form tiny capacitors, with the

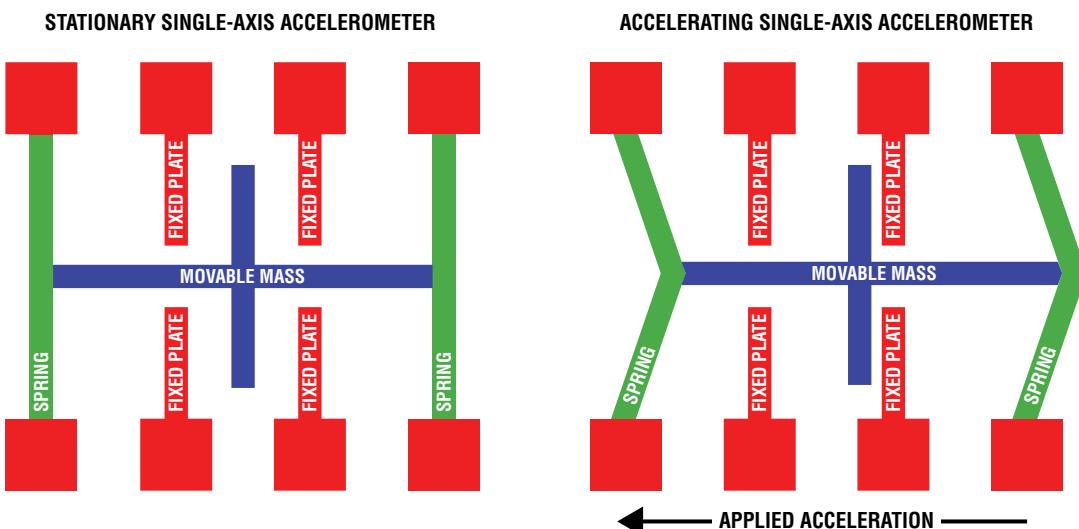


Figure 11-2: Simplified single-axis accelerometer

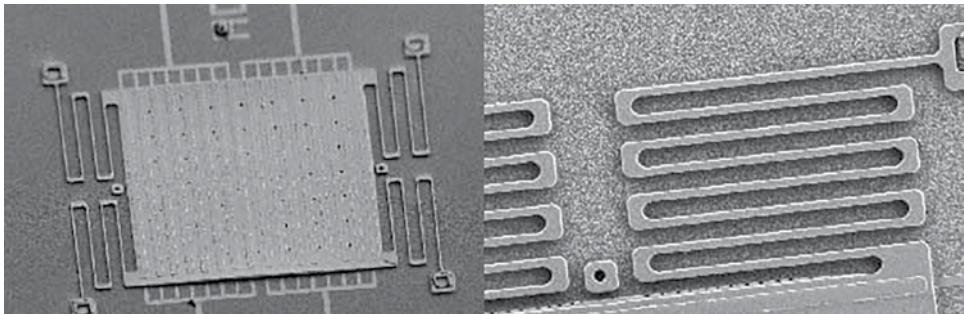


Figure 11-3: Surface of a micro-machined 3-axis accelerometer

Credit: Cornell NanoScale Science & Technology Facility, cnf.cornell.edu

capacitance changing as a function of the plates' distance from one another. As the plates move closer, the increased capacitance can be measured, and this can be correlated to a rate of acceleration.

Figure 11-3 shows an electron microscope view of a 3-axis MEMS accelerometer machined at the Cornell University NanoScale Science and Technology Facility. Note how you can see the springs and moving plates, as in the simplified illustration.

Gathering Information from the Datasheet

Although you'll be using a breakout board with well-labelled pins and a comprehensive software library, it's always good practice to understand the small details of any new part that you'll be using. This will minimize surprises later, and help you debug any potential problems that you run into. A quick Google search for **LIS3DH** will turn up the datasheet. You can also find a link to the datasheet on the Exploring Arduino website at exploringarduino.com/content2/ch11. The datasheet answers the following questions:

- What is the pin-out of the integrated circuit (IC), and which pins are the control pins?
- What are the acceleration axes for the device? That is, what orientation does the device need to be mounted in to detect an acceleration in the direction of interest for our project?
- What SPI commands are available for us to communicate with this chip?

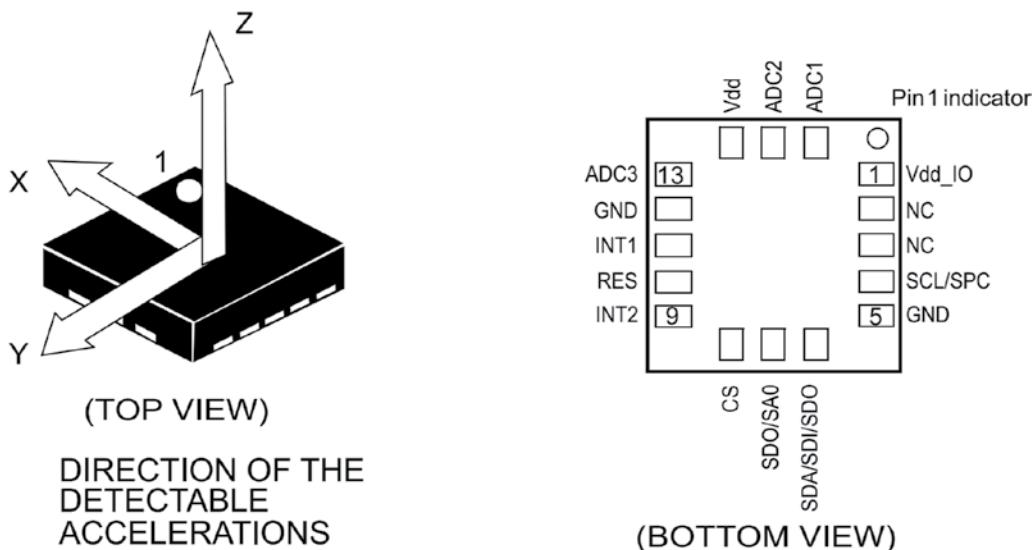


Figure 11-4: STMicroelectronics LIS3DH pin-out and orientation diagram

Credit: © STMicroelectronics. Used with permission.

To help you reference this information, Figures 11-4 and 11-5 show some of the pin-out details. First, take a look at Figure 11-4, which shows the pin-out and orientation drawing from page 8 of the datasheet.

For the first exercise with this chip, you'll be concerned with the z-axis that is shown in the diagram, because the chip will be sitting flat on the table (with the pin-side down), and you'll be looking at the acceleration due to gravity.

The pin-out is usually your first step when getting ready to work with a new device. Figure 11-5 shows an excerpt from page 9 of the datasheet, and shows the functionality of all the pins on the device.

You've probably already noticed from this pin-out that this device can actually operate in both I²C and SPI modes. In this chapter, you'll operate it in SPI mode. Why does the chip include both options? STMicroelectronics likely chose to support both I²C and SPI to give designers more options without having to manufacture multiple versions of a similar chip. Some designers might be using an accelerometer for periodic data collection that doesn't require high transfer speeds, but benefits from using fewer communication lines. Conversely, other designers might be streaming continuous acceleration data from the accelerometer to their microcontroller and want to run it at the fastest speed possible, regardless of how many extra wires it requires.

If you were going to be writing all the communication code for this chip, your next step would be to review the device's data registers, and the available SPI commands. However,

Pin#	Name	Function
1	Vdd_IO	Power supply for I/O pins
2	NC	Not connected
3	NC	Not connected
4	SCL SPC	I ² C serial clock (SCL) SPI serial port clock (SPC)
5	GND	0 V supply
6	SDA SDI SDO	I ² C serial data (SDA) SPI serial data input (SDI) 3-wire interface serial data output (SDO)
7 ⁽¹⁾	SDO SA0	SPI serial data output (SDO) I ² C less significant bit of the device address (SA0)
8	CS	SPI enable I ² C/SPI mode selection: 1: SPI idle mode / I ² C communication enabled 0: SPI communication mode / I ² C disabled
9	INT2	Inertial interrupt 2
10	RES	Connect to GND
11	INT1	Inertial interrupt 1
12	GND	0 V supply
13	ADC3	Analog-to-digital converter input 3
14	Vdd	Power supply
15	ADC2	Analog-to-digital converter input 2
16	ADC1	Analog-to-digital converter input 1

Figure 11-5: STMicroelectronics LIS3DH pin details

Credit: © STMicroelectronics. Used with permission.

you'll be using a pre-existing library to facilitate data exchange with the LIS3DH, so you can skip that step. You'll find that many popular chips have existing Arduino libraries.

Setting Up the Hardware

To get started with the LIS3DH accelerometer, you'll make a simple orientation detector. When upright, the detector will light a green LED. When turned upside down, the detector will light a red LED. You can choose to use two discrete LEDs for this exercise,

or a single RGB LED with two of the channels being controlled. I recommend the latter, as the final project in this chapter will make use of the RGB LED.

The Adafruit breakout board for this accelerometer takes care of the voltage level conversion for this chip so that you can use it with your 5V Arduino. As with I²C, the Arduino Uno has certain pins that are multiplexed to the ATmega's SPI hardware interface. Check out the Arduino website for details on which pins should be used for which function. This information is duplicated in Table 11-4 for your reference.

Table 11-4: Arduino Uno SPI pins

Arduino Uno pin	SPI Function
10	Chip Select (CS) / Slave Select (SS)
11	Master Out/Slave In (MOSI)
12	Master In/Slave Out (MISO)
13	Serial Clock (SCLK)

While this accelerometer library can support emulating the hardware SPI interface via arbitrary pins, using the native hardware interface is almost always preferable to performing software emulation. When you use the hardware interface, the ATmega can buffer incoming and outgoing data in a way that better utilizes CPU cycles than when you manually control the incoming and outgoing data streams with a software-emulated SPI.

Wire the SPI pins from the breakout board to the Arduino's pins that are listed in Table 11-4. Then, connect an RGB LED to pins 6, 5, and 3 (all PWM-capable, in case you want to control the brightness of the LED) via 220Ω current-limiting resistors. Remember, this is a common-anode LED, so the common pin (the longest one) should connect to 5V from the Arduino. Don't forget to also connect the breakout board's 5V and GND lines to the Arduino. When complete, your setup should look like Figure 11-6.

For this first test, the hardware will utilize the accelerometer to make a basic orientation sensor. As a result, you'll need to grab your Arduino Uno and breadboard to turn it upside down and right-side up again. If you have a bunch of dangling wires and resistors with long leads while you do this, it's very likely that something will come unplugged. Therefore, you should consider making a more robust assembly. You can search online for 3D-printable Arduino Uno-and-breadboard enclosures, you can tape both elements down to a wooden or plastic board, or you can tape them back-to-back to make the assembly more compact.

Figure 11-7 shows a very simple example of this, where I used painter's tape to adhere the breadboard to the back of the Arduino Uno. The tape covers some of the pins on the Uno, but you can easily poke your jumper wires right through the tape.

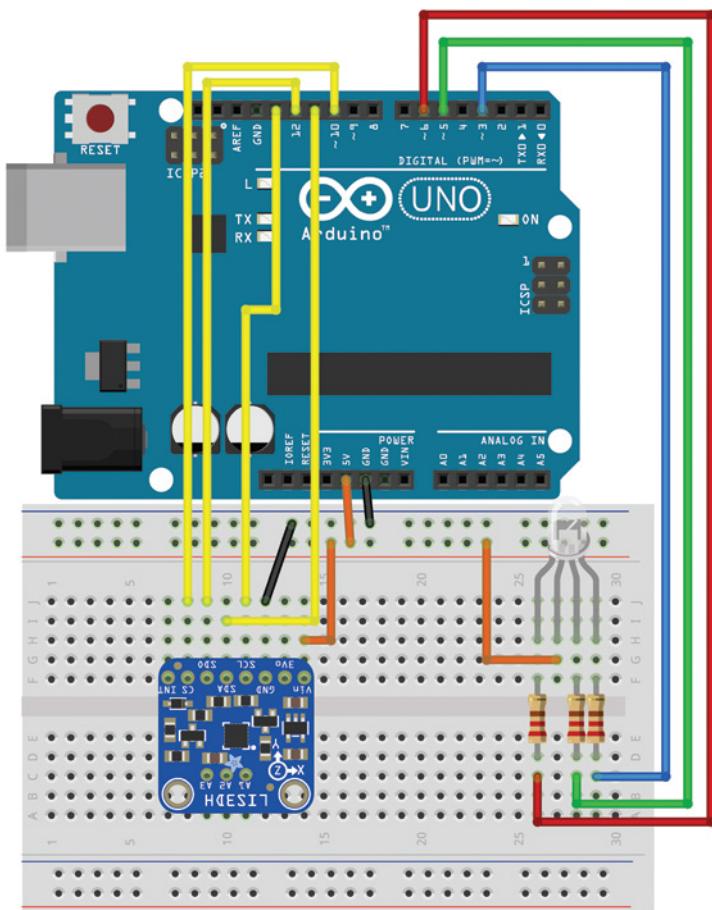


Figure 11-6: Accelerometer breakout and RGB LED wired to Arduino Uno

Created with Fritzing

This makes the assembly nice and compact, without being a permanent mounting. I also used solid-core wires instead of ordinary stranded jumper wires so I could bend them into place (but bendable stranded wires will work fine too).

Once you are happy with your assembly, double-check that your wiring matches the wiring diagram, and then move on to the next section, where you will install a software library for the accelerometer and write some software to detect the orientation of the Arduino.

Writing the Software

To confirm that your wiring is working and that your accelerometer is functional, you'll write a program to report the orientation of your Arduino. When the accelerometer is

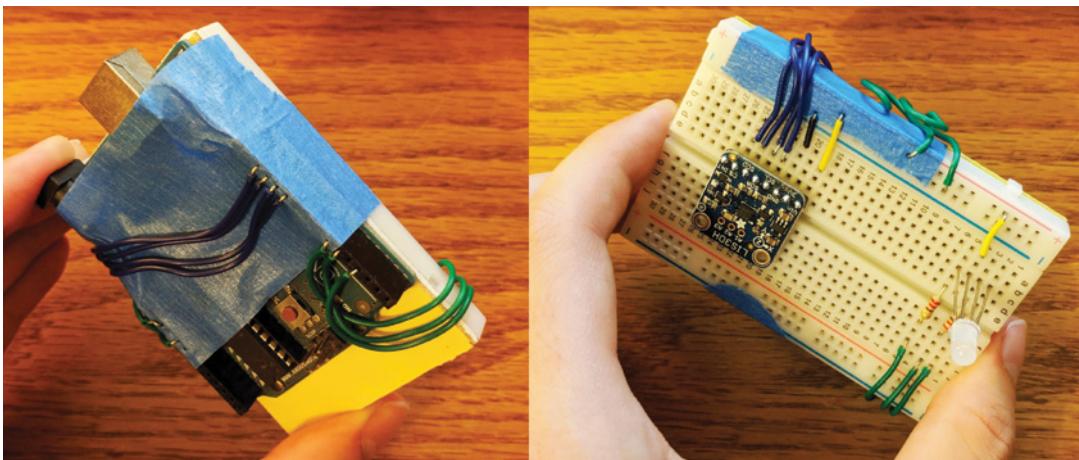


Figure 11-7: Breadboard taped to back of Arduino

facing up, the LED will illuminate green. When the accelerometer is facing down, the LED will illuminate red.

Installing the Adafruit Sensor Libraries

Although the Arduino IDE does have an integrated SPI library that you could use to manually communicate with your accelerometer, doing so would be fairly arduous. Sophisticated devices like accelerometers can be quite complicated to configure. If you revisit the datasheet, you'll see that there are a multitude of registers to configure, and a lot of available data for you to query. This can be overwhelming when you're just trying to get your bearings. Luckily, many popular parts like this accelerometer have community-developed libraries that add an extra abstraction layer atop the Arduino's SPI library. You'll shortly learn how to search for and install these libraries. Once you've mastered the contents of this book, you may want to develop your own libraries for components that aren't already supported by the community!

Manufacturers who sell breakout boards, like Adafruit, will often provide libraries to go along with them. If you visit Adafruit's website that accompanies this breakout board (blum.fyi/adafruit-LIS3DH-tutorial), you'll find details about using their provided libraries. In the olden days (the first edition of this book), you had to manually download library ZIP files, figure out the right place to put them, and then check whether the Arduino IDE properly detected them. Today, it's quite a bit easier! In order to use this accelerometer breakout board, you'll need two libraries provided by Adafruit: the *Adafruit Unified Sensor Library* and the *Adafruit LIS3DH Library*. Because the programmers at Adafruit write so many libraries, they built the Unified Sensor Library

as an abstraction layer to make it easier to build libraries for each of the sensors that they sell. The LIS3DH Library depends on Unified Sensor Library and will not compile properly if you do not install both libraries.

To install these libraries, open your Arduino IDE and navigate to Sketch > Include Library > Manage Libraries. A window opens, displaying a search bar. Search for **Adafruit Unified Sensor**. As shown in Figure 11-8, several results appear. Select the Adafruit Unified Sensor item, and click the Install button that appears. Then do the same for the Adafruit LIS3DH item. You now have both libraries installed and are ready to start writing software that uses these libraries!

To see how the LIS3DH library is used, you can view the sample sketches that come with it by going to File > Examples > Adafruit LIS3DH and selecting one of them.

Leveraging the Library

With the libraries installed, and the hardware built, it's time to write some code! The code in Listing 11-1 loads the libraries, connects to the accelerometer, and then grabs the current z-axis acceleration every 100 ms. It uses this data to update the state of the LED and to print the computed acceleration out over the serial port.

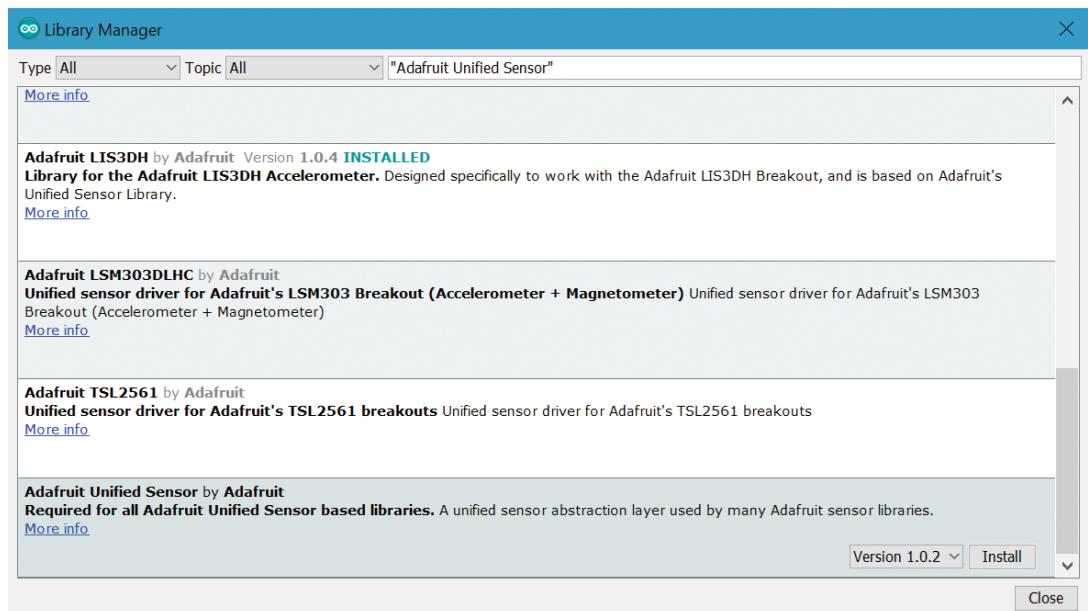


Figure 11-8: Installing Adafruit libraries

Listing 11-1

Accelerometer-based orientation sensor-orientation.ino

```
// Uses the Z-Axis of an Accelerometer to detect orientation

// Include Libraries
// This library will, itself, include the SPI and Universal Sensor Libraries
#include <Adafruit_LIS3DH.h>

// Define the pins (the SPI hardware pins are used by default)
const int RED_PIN = 6;
const int GREEN_PIN = 5;
const int CS_PIN = 10;

// Set up the accelerometer using the hardware SPI interface
Adafruit_LIS3DH accel = Adafruit_LIS3DH(CS_PIN);

void setup()
{
    Serial.begin(9600); // Set up the serial port so we can see readings

    // Connect to the accelerometer
    if (!accel.begin())
    {
        Serial.println("Could not find accelerometer.");
        while (1); // Loop forever
    }

    // Set the sensitivity of the accelerometer to +/-2G
    accel.setRange(LIS3DH_RANGE_2_G);

    // Set the LED cathode pins as outputs and turn them off
    // HIGH is off because this is a common anode LED
    pinMode(RED_PIN, OUTPUT);
    digitalWrite(RED_PIN, HIGH);
    pinMode(GREEN_PIN, OUTPUT);
    digitalWrite(GREEN_PIN, HIGH);
}

void loop()
{
    // Get X, Y, and Z accelerations
    accel.read();

    // Print the Raw Z Reading
    Serial.print("Raw: ");
    Serial.print(accel.z);
```

```
// Map the Raw Z Reading G's based on +/-2G Range
Serial.print("\tActual: ");
Serial.print((float(accel.z)/32768.0)*2.0);
Serial.println("G");

// Check if we are upside-down
if (accel.z < 0)
{
    digitalWrite(RED_PIN, LOW);
    digitalWrite(GREEN_PIN, HIGH);
}
else
{
    digitalWrite(RED_PIN, HIGH);
    digitalWrite(GREEN_PIN, LOW);
}

// Get new data every 100ms
delay(100);
}
```

Study the code in Listing 11-1. Can you decipher what each line does? The LIS3DH library is included first. Note that if you didn't go through the earlier steps to install this library, then the compiler will fail at this line when it cannot find this library. The LIS3DH library itself includes the Adafruit Universal Sensor Library, and any code needed to talk to the SPI interface of the Arduino. After the library inclusion, the relevant pins are defined. Note that only the CS pin must be defined for the SPI interface. This is because the library defaults to using the known hardware SPI pins that are listed in Table 11-4 when you initialize it. `Adafruit_LIS3DH accel = Adafruit_LIS3DH(CS_PIN);` creates an LIS3DH object called `accel`, which is utilized later in the code to communicate with the accelerometer. Note that the only argument is the chip select (CS) pin because the other pins default to their hardware pins.

In the `setup()` function, `accel.begin()` attempts to connect to the accelerometer over its SPI interface. Because this call is placed in an `if()` statement, two things are done at one time: the accelerometer is initialized, and the outcome of that initialization is used to determine whether your sketch should proceed. The `begin()` function that is defined by the LIS3DH library returns a Boolean representing whether communication could be successfully established with the chip. If that function returns `False`, then the `if()` statement evaluates to `True` (because the `!` negates the value of whatever it is in front of). This causes an error message to be printed on the Serial interface, and the program execution is then halted by being put into an endless loop with `while(1);`.

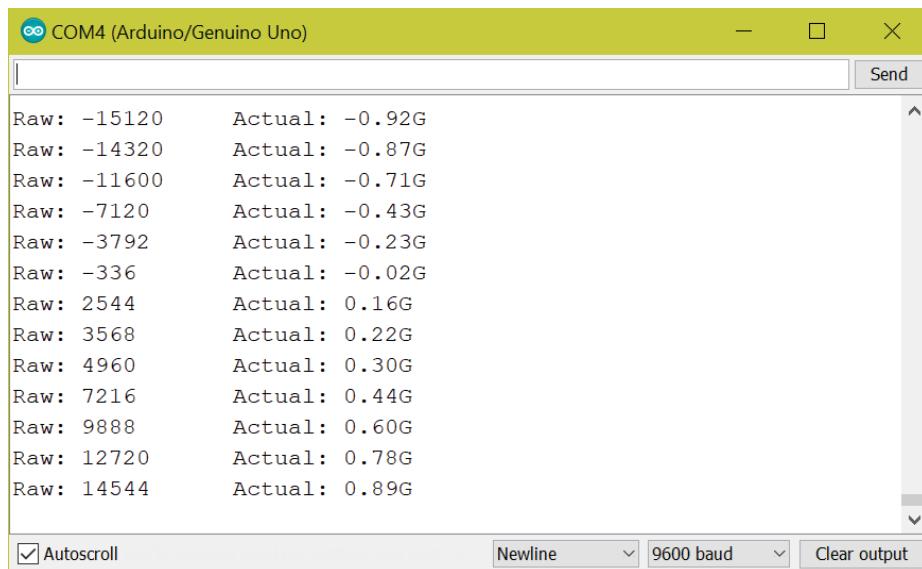
Assuming the initialization is successful, the remainder of the setup function executes: the sensitivity of the accelerometer is set, and LED pins are set as outputs with the LEDs turned off. This accelerometer has several available sensitivity settings. This code is setting the sensitivity to the smallest range: $\pm 2\text{G}$. Because this chip returns a signed 16-bit integer, this means that when experiencing $+2\text{G}$, the returned raw reading is $(+2^{16}/2) - 1$ and when experiencing -2G , the returned raw reading is $-2^{16}/2$. Why? Because 16 bits of information means 2^{16} possible values. If the chip wants to evenly report both a negative and positive range, then those 2^{16} (65,536) possible values must be split in half; when you incorporate 0, that works out to a range from -32768 to $+32767$. If you change the sensitivity from $\pm 2\text{G}$ to $\pm 4\text{G}$ by setting the sensitivity to LIS3DH_RANGE_4_G, then $+32767$ represents $+4\text{G}$ instead of $+2\text{G}$. So, this line of code is setting the scale of the reading, and also its resolution. As you increase the readable range, you trade off for less and less resolution per unit of acceleration.

In the loop, the following actions occur every 100 ms:

1. `accel.read()` gets the current acceleration values from all axes of the accelerometer.
2. The raw value (the one that ranges from -32768 to $+32767$) is printed over the serial interface.
3. The actual acceleration, measured in Gs, is computed from the raw value using the scale that was previously set.
4. The sign of the acceleration is checked. If the acceleration is positive, then the accelerometer is pointing up and the light turns green. If the acceleration is negative, then the accelerometer is pointing down and the light turns red.

Run the software and open your serial monitor. You should see a data stream that looks like Figure 11-9. If you only see “Could not find accelerometer,” then double-check your wiring, because that means the Arduino cannot talk to the accelerometer over the SPI interface. In the serial output snippet shown in Figure 11-9, the accelerometer starts facing down and is quickly turned upright. When upside down, the acceleration should be roughly -1G . When upright, it should be roughly $+1\text{G}$. What will the acceleration be if you drop your Arduino into a freefall? (Note: you probably shouldn’t test your theory unless you have a soft surface for your Arduino to land on.) The LED on your assembly should glow green while upright and red while upside down.

NOTE To watch a demo video of the accelerometer-based orientation sensor, visit exploringarduino.com/content2/ch11.



The screenshot shows the Arduino Serial Monitor window titled "COM4 (Arduino/Genuino Uno)". The window displays a list of 15 data entries, each consisting of a raw value followed by an actual value. The raw values range from -15120 to 14544, while the actual values range from -0.92G to 0.89G. The monitor includes standard controls at the bottom: an "Autoscroll" checkbox (checked), a "Newline" dropdown set to "9600 baud", and a "Clear output" button.

Raw:	Actual:
-15120	-0.92G
-14320	-0.87G
-11600	-0.71G
-7120	-0.43G
-3792	-0.23G
-336	-0.02G
2544	0.16G
3568	0.22G
4960	0.30G
7216	0.44G
9888	0.60G
12720	0.78G
14544	0.89G

Figure 11-9: Data streaming from the accelerometer

Now that you have this orientation sensor working, why not put the other axes of your 3-axis accelerometer to work? In the next section, you will increase the complexity of the system by turning it into an audiovisual instrument.

Creating an Audiovisual Instrument Using a 3-Axis Accelerometer

Detecting orientation is a great start, but that's only one axis of information! What about the other two? In this section, you will integrate data from all three axes of the accelerometer to make a fun musical instrument. And, for good measure, you'll integrate some lighting effects, too. Who doesn't like blinky LEDs? As you learned in Chapter 6, "Making Sounds and Music," the Arduino IDE has a tone library that allows you to easily produce square waves from any pin on the Arduino to drive a speaker. You also learned about the pentatonic scale, which always sounds good. You'll leverage that knowledge for your instrument.

NOTE This project is intentionally designed as a jumping-off point: you will make a fun audiovisual instrument that you can expand on in software to create much more inspired projects. Get this example working first, and then see how you can build on it to make something truly personal. This exercise offers an ideal opportunity to

get creative with your Arduino. You may want to add some buttons to control sound duration, or a light sensor to shift the frequencies.

Setting Up the Hardware

The setup here is an extension of the setup you were already using for the previous exercise. To your existing circuit, add a piezo buzzer or speaker wired into pin 9. I recommend a piezo buzzer for this exercise because it is lightweight and can be easily mounted to your breadboard. A piezo buzzer is similar in functionality to a speaker, but it trades audio fidelity for size. Piezo buzzers are optimized for buzzing at the frequency of a provided square wave, and will not nicely reproduce all the frequencies that are generally required for high fidelity music. As with a speaker, don't forget to include a resistor in series with the piezo buzzer; it doesn't matter if you place it between the piezo buzzer and pin 9, or between the piezo buzzer and ground. The larger the resistor, the quieter the sound from the piezo/speaker; I chose to use another 220Ω resistor. Your wiring should look like Figure 11-10. Don't forget to secure your buzzer so you can wave your instrument around.

Modifying the Software

To begin making some music, you can edit the code from Listing 11-1. Add a pin variable for the speaker and the remaining LED diode (blue), as well as some variables to define the notes you'll play. You can take inspiration from the pentatonic scale that you learned about in Chapter 6. I chose to define six notes—two for each direction of the accelerometer (one positive and one negative). Inside `loop()`, retrieve the present acceleration values from the accelerometer.

You'll need to decide how you want to "hold" this instrument in your hand. Depending on what axis is facing towards the earth, that axis will experience a constant $+1G$ of acceleration. If you want the instrument to only react to relative motion, then you need to compensate for the force of gravity along the axis that is normal to the earth's surface.

In Listing 11-2, I've chosen to use the z-axis for this purpose, so I "normalized" its readings by subtracting $1G$ (in raw value) from the z-axis reading. Recall that with a $+/2G$ sensitivity, a raw value of 32767 represents $+2G$. Thus, subtracting half of that amount (16384 in raw value, or $+1G$) from the z-axis reading will remove the effect of gravity on the readings from that axis (assuming you continue to hold your device with the accelerometer facing up).

Finally, select a suitable threshold (it doesn't have to be the same for each axis), and use that threshold to trigger the Piezo buzzer at the desired frequency. When a certain

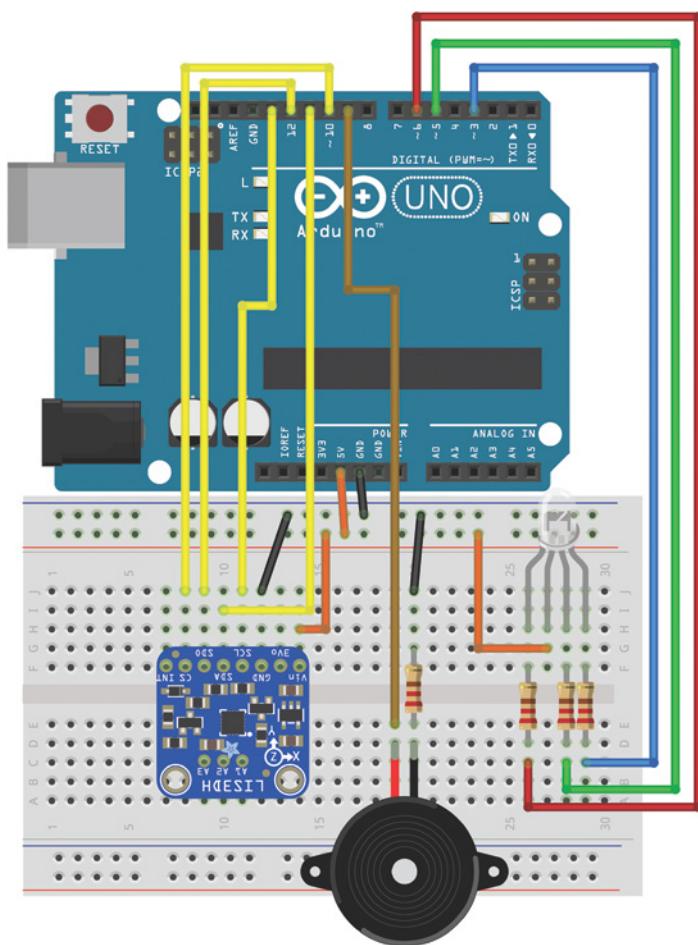


Figure 11-10: Motion-based instrument wiring

Created with Fritzing

axis accelerates beyond a value of your choosing, trigger a sound at the desired frequency. Don't forget to use the normalized acceleration that you've computed for the axis that is normal to the earth's surface.

You can also print all these values to the serial monitor to help you select suitable threshold levels. Use the `map()` and `constrain()` functions to map the raw values from the accelerometer to LED brightness. The LED is in a common anode configuration, so remember to reverse the mapping order—255 turns the LED off, and 0 turns the LED on to full brightness. A completed sketch that takes all of these values into account is provided in Listing 11-2.

Listing 11-2

Accelerometer-based instrument -instrument.ino

```
// Uses the each Axis of an Accelerometer to control lights and sounds

// Include Libraries
// This library will, itself, include the SPI and Universal Sensor Libraries
#include <Adafruit_LIS3DH.h>

// Define the pins (the SPI hardware pins are used by default)
const int RED_PIN = 6;
const int GREEN_PIN = 5;
const int BLUE_PIN = 3;
const int SPEAKER = 9;
const int CS_PIN = 10;

// Pentatonic Piano C D E G A
#define NOTE_C 262 //Hz
#define NOTE_D 294 //Hz
#define NOTE_E 330 //Hz
#define NOTE_G 392 //Hz
#define NOTE_A 440 //Hz
#define NOTE_C2 523 //Hz

// Set up the accelerometer using the hardware SPI interface
Adafruit_LIS3DH accel = Adafruit_LIS3DH(CS_PIN);

void setup()
{
    Serial.begin(9600); // Set up the serial port so we can see readings

    // Connect to the accelerometer
    if (!accel.begin())
    {
        Serial.println("Could not find accelerometer.");
        while (1); // Loop forever
    }

    // Set the sensitivity of the accelerometer to +/-2G
    accel.setRange(LIS3DH_RANGE_2_G);

    // Set the LED cathode pins as outputs and turn them off
    // HIGH is off because this is a common anode LED
    pinMode(RED_PIN, OUTPUT);
    digitalWrite(RED_PIN, HIGH);
```

```
pinMode(GREEN_PIN, OUTPUT);
digitalWrite(GREEN_PIN, HIGH);
pinMode(BLUE_PIN, OUTPUT);
digitalWrite(BLUE_PIN, HIGH);
}

void loop()
{
    // Get X, Y, and Z accelerations
    accel.read();

    // Normalize the axis that is normal to the Earth
    // Subtract the raw equivalent of 1G of acceleration
    long norm_z = accel.z-16384;

    // Print all the accelerations so we can tune the thresholds below
    Serial.print(accel.x);
    Serial.print(" ");
    Serial.print(accel.y);
    Serial.print(" ");
    Serial.println(norm_z);

    // Trigger a different 100ms note based on the direction of acceleration
    if (accel.x < -5000) tone(SPEAKER, NOTE_C, 100);
    if (accel.x > 5000) tone(SPEAKER, NOTE_D, 100);
    if (accel.y < -5000) tone(SPEAKER, NOTE_E, 100);
    if (accel.y > 5000) tone(SPEAKER, NOTE_G, 100);
    if (norm_z < -5000) tone(SPEAKER, NOTE_A, 100);
    if (norm_z > 5000) tone(SPEAKER, NOTE_C2, 100);

    // Light the LEDs proportional to the direction of acceleration
    analogWrite(RED_PIN, constrain(map(abs(accel.x),5000,20000,255,0),0,255));
    analogWrite(GREEN_PIN, constrain(map(abs(accel.y),5000,20000,255,0),0,255));
    analogWrite(BLUE_PIN, constrain(map(abs(norm_z),5000,20000,255,0),0,255));
}
```

Load this program onto your Arduino, and give it a good shake in all directions! How does it sound? What happens when you rotate it about an axis instead of accelerating it linearly? Why is it different? Experiment with different thresholds or ways to combine the data from multiple axes.

Ready for your next challenge? Adafruit also makes a breakout board for the L3GD20H, a 3-axis gyroscope. How do you think a gyroscope would perform differently from an accelerometer in this application? The L3GD20H works similarly, and also has a third-party library available from Adafruit. Try combining it with your accelerometer to make a true inertial measurement unit (IMU).

NOTE To watch a demo video of the accelerometer instrument in action, visit exploringarduino.com/content2/ch11.

Summary

In this chapter, you learned the following:

- The SPI bus uses two data lines, a clock line, and a slave select line. An additional slave select line is added for each slave device, but the other three lines are shared on the bus.
- Accelerometers use tiny moving elements to measure physical forces.
- Third-party libraries can be installed into the Arduino IDE to facilitate easy communication between the Arduino and slave devices.
- Accelerometers can be queried and used to understand acceleration data by the Arduino.
- You also combined your knowledge of SPI sensors, third-party libraries, LED brightness control, and audio output.

12

Interfacing with Liquid Crystal Displays

Parts You'll Need for This Chapter

- Arduino Uno or Adafruit METRO 328
- USB cable (Type A to B for Uno, Type A to Micro-B for METRO)
- Half-size or full-size breadboard
- Assorted jumper wires
- Pushbuttons ($\times 2$)
- 220 Ω resistor
- 1k Ω resistor
- 4.7k Ω resistors ($\times 2$)
- 10k Ω resistors ($\times 2$)
- 10k Ω trim potentiometer (might be included with LCD purchase)
- 9V battery
- 9V battery clip
- L7805CV 5V voltage regulator
- 10 μ F 50V electrolytic capacitors ($\times 2$)
- 1N4001 diode
- PN2222 NPN bipolar junction transistor (BJT)
- 8 Ω loudspeaker
- TC74A0-5.0VAT I²C temperature sensor

Miniature 5V DC brushless cooling fan

16×2 character LCD with header pins

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch12

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

One of the best things about designing embedded systems is that they can operate independently from a computer. Up until now, you've been tethered to the computer if you wanted to display any kind of information more complicated than an illuminated LED. By adding a liquid crystal display (LCD) to your Arduino, you can more easily display complex information (sensor values, timing information, settings, progress bars, and so on) directly on your Arduino project without having to interface with the serial monitor through a computer.

In this chapter, you will learn how to connect an LCD to your Arduino, and how to use the Arduino LiquidCrystal library to write text and arbitrary custom characters to your LCD. After you have the basics down, you will add some components from previous chapters to make a simple thermostat capable of obtaining local temperature data, reporting it to you, and controlling a fan to compensate for heat. An LCD will give you live information, a speaker will alert you when the temperature is getting too hot, and the fan will turn on to automatically cool you down.

NOTE You can watch a tutorial video about how to interface your Arduino with an LCD on this chapter's content web page: exploringarduino.com/content2/ch12.

Setting Up the LCD

To complete the examples in this chapter, you will use a parallel LCD screen. These are extremely common and come in all shapes and sizes. The most common is a 16×2 character display with a single row of 16 pins (14 if it does not have a backlight). In this chapter, you will use a 16-pin LCD display that can show a total of 32 characters (16 columns and 2 rows).

If your display didn't come with a 16-pin header already soldered on, you need to solder one on so that you can easily install it in your breadboard. With the header successfully soldered on, your LCD should look like the one shown in Figure 12-1, and you can insert it into your breadboard.

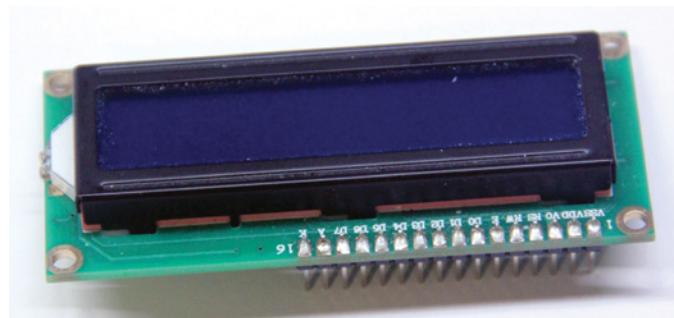


Figure 12-1: LCD with headers soldered on

Next, you wire up your LCD to a breadboard and to your Arduino. All of these parallel LCD modules have the same pin-out and can be wired in one of two modes: 4-pin or 8-pin mode. You can accomplish everything you want to do using just 4 pins for communication; that's how you'll wire it up. There are also pins for enabling the display, setting the display to command mode or character mode, and setting it to read/write mode. Table 12-1 describes all of these pins.

Table 12-1: Parallel LCD pins

pin Number	pin Name	pin Purpose
1	VSS	Ground connection
2	VDD	+5V connection
3	V0	Contrast adjustment (to potentiometer)
4	RS	Register selection (Character versus Command)
5	RW	Read/write
6	EN	Enable
7	D0	Data line 0 (unused in 4-pin mode)
8	D1	Data line 1 (unused in 4-pin mode)
9	D2	Data line 2 (unused in 4-pin mode)
10	D3	Data line 3 (unused in 4-pin mode)
11	D4	Data line 4
12	D5	Data line 5
13	D6	Data line 6
14	D7	Data line 7
15	A	Backlight anode
16	K	Backlight cathode

Here's a breakdown of the pin connections:

- The contrast adjustment pin changes how dark the display is. It connects to the center pin of a potentiometer.
- The register selection pin sets the LCD to command or character mode, so it knows how to interpret the next set of data that is transmitted via the data lines. Based on the state of this pin, data sent to the LCD is interpreted as either a command (for example, to move the cursor) or characters (for example, the letter *a*).
- The RW pin is always tied to ground in this implementation, meaning that you are only writing to the display and never reading from it.
- The EN pin is used to tell the LCD when data is ready.
- Data pins 4 to 7 are used for actually transmitting data, and data pins 0 to 3 are left unconnected.
- You can illuminate the backlight by connecting the anode pin to 5V and the cathode pin to ground if you are using an LCD with a built-in resistor for the backlight. If you are not, you must put a current-limiting resistor in-line with the anode or cathode pin. The datasheet for your device will tell you if you need to do this.

You can connect the communication pins of the LCD to any I/O pins on the Arduino. In this chapter, they are connected as shown in Table 12-2.

Reference the wiring diagram shown in Figure 12-2, and hook up your LCD accordingly.

Now your LCD is ready for action! Once you get the code loaded in the next section, you can start displaying text on the screen. The potentiometer will adjust the contrast between the text and the background color of the screen.

Table 12-2: Communication pin Connections

LCD pin	Arduino pin Number
RS	pin 2
EN	pin 3
D4	pin 4
D5	pin 5
D6	pin 6
D7	pin 7

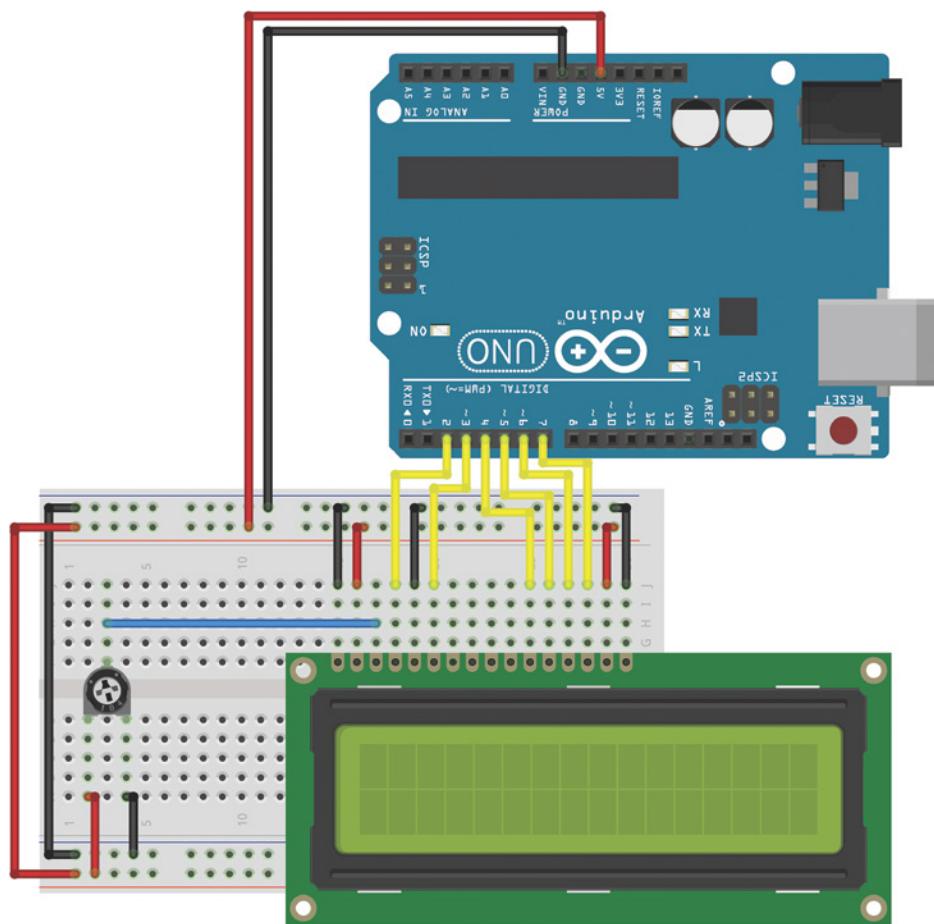


Figure 12-2: LCD wired to breadboard and Arduino

Created with Fritzing

Using the LiquidCrystal Library to Write to the LCD

The Arduino IDE includes the `LiquidCrystal` library, a set of functions that makes it very easy to interface with the parallel LCD that you are using. The `LiquidCrystal` library has an impressive amount of functionality, including blinking the cursor, automatically scrolling text, creating custom characters, and changing the direction of text printing. This chapter does not cover every function, but instead gives you the tools you need to interface with the display's most important functions. You can find descriptions of the library functions and examples illustrating their use on the Arduino website, at blum.fyi/arduino-lcd-library. (This is also linked from exploringarduino.com/content2/ch12.)

Adding Text to the Display

In this first example, you add some text and an incrementing number to the display. This exercise demonstrates how to initialize the display, how to write text, and how to move the cursor. First, include the LiquidCrystal library:

```
#include <LiquidCrystal.h>
```

Then, initialize an LCD object, as follows:

```
LiquidCrystal lcd (2,3,4,5,6,7);
```

The arguments for the LCD initialization represent the Arduino pins connected to RS, EN, D4, D5, D6, and D7, in that order. In the setup, you call the library's `begin()` function to set up the LCD display with the character size. (The one I'm using is a 16×2 display, but you may be using another size, such as 20×4.) The arguments for this command represent the number of columns and the number of rows, respectively:

```
lcd.begin(16, 2);
```

After adding this code, you can call the library's `print()` and `setCursor()` commands to print text to a given location on the display. For example, if you want to print my name on the second line, you issue these commands:

```
lcd.setCursor(0,1);
lcd.print("Jeremy Blum");
```

The positions on the screen are indexed starting with $(0,0)$ in the top-left position. The first argument of `setCursor()` specifies the column number, and the second argument specifies the row number. By default, the starting location is $(0,0)$. So, if you call `print()` without first changing the cursor location, the text starts in the top-left corner.

WARNING The library does not check for strings that are too long. So, if you try to print a string starting at position 0 that is longer than the number of characters in the row you are addressing (16 in the case of the LCD used in these examples), you may notice strange behavior. Make sure to check that whatever you are printing will fit on the display!

Using this knowledge, you can now write a simple program that displays some text on the first row and that prints a counter that increments once every second on the

second row. Listing 12-1 shows the complete program you need to accomplish this. Load it on to your Arduino and confirm that it works as expected. If you don't see anything, adjust the contrast with the potentiometer.

Listing 12-1

LCD text with an incrementing number-LCD_text.ino

```
//LCD text with incrementing number

//Include the library code:
#include <LiquidCrystal.h>

//Start the time at 0
int time = 0;

//Initialize the library with the numbers of the interface pins
LiquidCrystal lcd(2, 3, 4, 5, 6, 7);

void setup()
{
    //Set up the LCD's number of columns and rows:
    lcd.begin(16, 2);
    // Print a message to the LCD.
    lcd.print("Jeremy's Display");
}

void loop()
{
    //Move cursor to second line, first position
    lcd.setCursor(0,1);
    //Print Current Time
    lcd.print(time);
    //Wait 1 second
    delay(1000);
    //Increment the time
    time++;
}
```

This program combines all the steps that you learned about earlier. The library is first included at the top of the program. A time variable is initialized to 0, so that it can be incremented once per second during the `loop()`. A `LiquidCrystal` object called `lcd` is created with the proper pins assigned based on the circuit you've already wired up. In the `setup`, the LCD is configured as having 16 columns and 2 rows, by calling `lcd.begin(16,2)`. Because the first line never changes, it can be written in the `setup`. This is accomplished

with a call to `lcd.print()`. Note that the cursor position does not need to be set first, because you want the text to be printed to position $(0,0)$, which is already the default starting location. In the loop, the cursor is always set back to position $(0,1)$ so that the number you print every second overwrites the previous number. The display updates once per second with the incremented time value.

Creating Special Characters and Animations

What if you want to display information that cannot be expressed using normal text? Maybe you want to add a Greek letter, a degree sign, or some progress bars. Thankfully, the LiquidCrystal library supports the definition of custom characters that can be written to the display. In the next example, you will use this capability to make an animated progress bar that scrolls across the display. After that, you will take advantage of custom characters to add a degree sign when measuring and displaying temperature.

Creating a custom character is pretty straightforward. If you take a close look at your LCD, you'll see that each character block is actually made up of a 5x8 grid of pixels. (Figure 12-3 shows a magnified view of this.) To create a custom character, you simply have to define the value of each of these pixels and send that information to the display. To try this out, you'll make a series of characters that will fill the second row of the display with an animated progress bar. Because each character space is 5 pixels wide, there will be a total of five custom characters: one with one column filled, one with two columns filled, and so on. To write the code for this, it helps to first visualize exactly what the pixel array will look like. Figure 12-3 shows how each of the five custom characters will look.

At the top of your sketch where you want to use the custom characters, create a byte array with 1s representing pixels that will be turned on, and with 0s representing pixels that will be turned off. The byte array representing the character that fills the first column (or the first 20 percent of the character) looks like this:

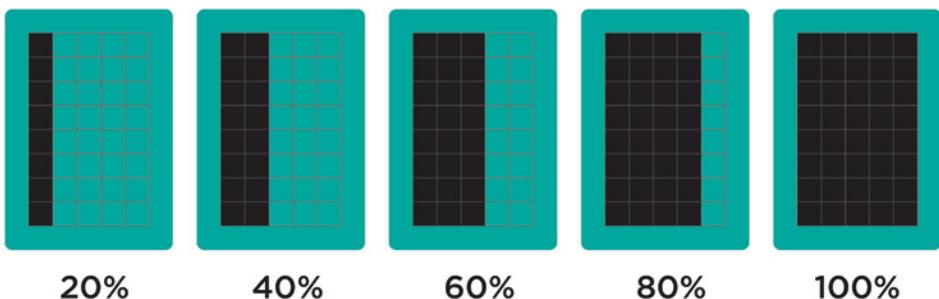


Figure 12-3: Five custom progress bar characters

I chose to call this byte array p20, to represent that it is filling 20 percent of one character block (the p stands for percent). Note how the ones and zeros corresponded to the filled pixel positions from Figure 12-3.

In the `setup()` function, call the `createChar()` function to assign your byte array to a custom character ID. Custom character IDs start at 0 and go up to 7, so you can have a total of eight custom characters. To map the 20-percent character byte array to custom character 0, type the following within your `setup()` function:

```
lcd.createChar(0, p20);
```

When you're ready to write a custom character to the display, place the cursor in the right location and use the library's `write()` function with the ID number:

```
lcd.write((byte)0);
```

In the preceding line, `(byte)` casts, or changes, the `0` to a byte value. This is necessary *only* when writing character ID 0 directly (without a variable that is defined to 0), to prevent the Arduino compiler from throwing an error caused by the variable type being ambiguous. Try removing `(byte)` from this command and observe the error that the Arduino IDE displays. You can write other character IDs without it, like this:

```
lcd.write(1);
```

Putting this all together, you can add the rest of the characters and put two nested `for()` loops in your program loop to handle updating the progress bar. The completed code looks like Listing 12-2.

Listing 12-2

LCD updating progress bar code—LCD_progress_bar.ino

```
//LCD with Progress Bar
```



```
byte p100[8] = {  
    B11111,  
    B11111,  
    B11111,  
    B11111,  
    B11111,  
    B11111,  
    B11111,  
    B11111,  
};  
  
void setup()  
{  
    //Set up the LCDs number of columns and rows:  
    lcd.begin(16, 2);  
    // Print a message to the LCD.  
    lcd.print("Jeremy's Display");  
  
    //Make progress characters  
    lcd.createChar(0, p20);  
    lcd.createChar(1, p40);  
    lcd.createChar(2, p60);  
    lcd.createChar(3, p80);  
    lcd.createChar(4, p100);  
}  
  
void loop()  
{  
    //Move cursor to second line  
    lcd.setCursor(0,1);  
    //Clear the line each time it reaches the end  
    //with 16 " " (spaces)  
    lcd.print("          ");  
  
    //Iterate through each character on the second line  
    for (int i = 0; i<16; i++)  
    {  
        //Iterate through each progress value for each character  
        for (int j=0; j<5; j++)  
        {  
            lcd.setCursor(i, 1); //Move the cursor to this location  
            lcd.write(j); //update progress bar  
            delay(100); //wait  
        }  
    }  
}
```

At the beginning of each pass through the loop, the 16-character-long string of spaces is written to the display, clearing the progress bar before it starts again. The outer `for()` loop iterates through all 16 positions. At each character position, the inner `for()` loop keeps the cursor there and writes an incrementing progress bar custom character to that location. The byte cast is not required here because the ID 0 is defined by the `j` variable in the `for()` loop.

NOTE To watch a demo video of the updating progress bar, visit exploringarduino.com/content2/ch12.

Building a Personal Thermostat

Now, let's make this display a bit more useful. To do so, you add the temperature sensor from Chapter 10, "The I²C Bus," a fan (using your motor skills from Chapter 4, "Using Transistors and Driving DC Motors"), and the speaker from Chapter 6, "Making Sounds and Music." The display shows the temperature and the current fan state. When it gets too hot, the speaker makes a noise to alert you, and the fan turns on. When it gets sufficiently cool again, the fan turns off. Using two pushbuttons and the debounce code in Listing 2-5 in Chapter 2, "Digital Inputs, Outputs, and Pulse-Width Modulation," you add the ability to increment or decrement the desired temperature.

Setting Up the Hardware

The hardware setup for this project is a conglomeration of previous projects. You should treat the fan similarly to the motors you learned about in Chapter 4. The recommended 5V brushless fan for this project will consume more power than your Arduino can provide from an I/O pin, so you'll need to drive it with a transistor. To drive the fan, use an NPN transistor, referencing the schematic that you used in Chapter 4 (Figure 4-1). Similarly to the 5V DC motors that you used in the roving car project from Chapter 4, this fan should be powered off its own 5V supply. Plug a 9V battery into the Arduino's barrel jack and use that as the input voltage to the linear regulator (the VIN pin on the Arduino can be used to connect the battery's 9V supply to the regulator). The regulator will generate a 5V supply to be used by the fan. This will ensure that electrical noise from the fan turning on and off does not impact the performance of the temperature sensor (which is powered using the Arduino's on-board voltage regulator). As you did in Chapter 4, ensure that you equip the regulator with 10 μ F decoupling capacitors on its input and output. Remember that the regulator's ground must be connected to your Arduino's ground. Figure 12-4 shows a schematic of the components that you'll be adding to the LCD that you've already wired up (it does not show the LCD and potentiometer that you've already wired).

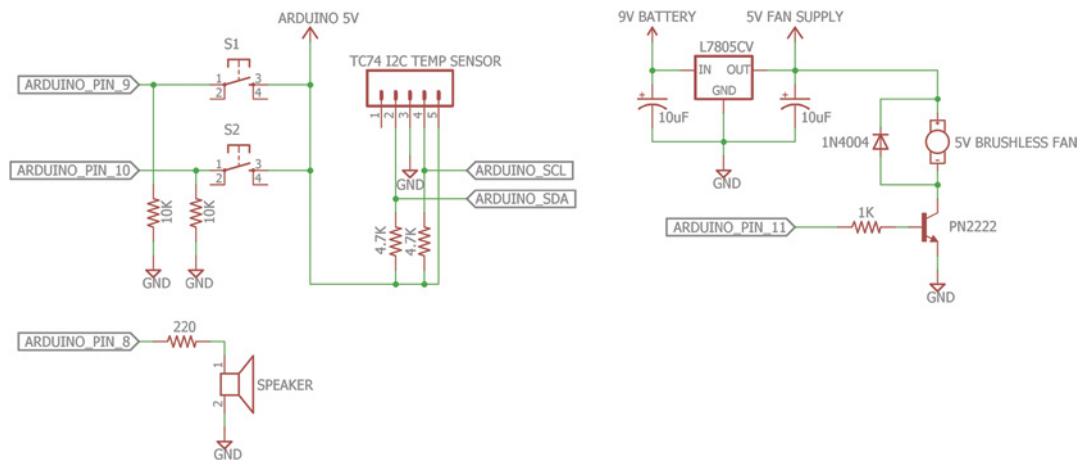


Figure 12-4: LCD thermostat additions schematic

Created with EAGLE

Referencing only the schematic drawing, try to add the fan, 5V regulator, drive transistor, protection diode, temperature sensor, speaker, and pushbuttons. Since this fan is a brushless motor, you can omit the small capacitor that you put across the leads of the brushed DC motor in Chapter 4; there are no brushes to make the RF (radio frequency) interference that the capacitor is normally used to reduce. Note how the 5V supply generated by the L7805CV voltage regulator is distinct from the Arduino's 5V supply.

You might need to rearrange your placement of the LCD and trim potentiometer to make room for all your circuitry.

The two buttons have one side connected to power; the other side is connected to ground through $10\text{k}\Omega$ pull-down resistors and to the Arduino.

The speaker is connected to an I/O pin through a 220Ω resistor and to ground. The frequency of the sound will be set in the program.

You connect the I²C temperature sensor exactly as you did in Chapter 10. Don't forget the pull-up resistors!

Plug a 9V battery holder into the Arduino's barrel jack. This will enable you to draw 9V power from the Arduino's VIN pin for powering your 5V linear regulator.

When wiring up the linear regulator, recall that the stripe on the decoupling capacitors represents the negative pin of the capacitor, which should be connected to the shared ground. On the diode, the side with the stripe should be connected to the fan's positive wire and the fan's 5V supply; the other side should be connected to the fan's negative wire and the NPN transistor's collector pin.

The diagram in Figure 12-5 shows the complete wiring setup with everything you need to create this project. It's possible to fit this all onto a half-sized breadboard, but as you can see from the wiring diagram, it is quite cramped. You may wish to consider using a full size breadboard for this project.

NOTE You must have the 9V battery (or a wall 9V/12V supply) plugged into the Arduino for the fan to work properly. If you plug the Arduino into your USB port without the battery or wall power connected, you'll be able to program the Arduino and control the LCD, but the fan will never spin because the input to the linear regulator will be connected to the 5V USB supply (the Arduino automatically falls back to USB power if a higher voltage source connected to VIN or the barrel jack is not present). The regulator will be unable to generate a 5V output with only a 5V input because the regulator requires its input voltage to be at least 2V higher than its output voltage. If you are using an Adafruit METRO instead of an Arduino Uno, make sure that the DC jack switch is in the on position to draw power from the DC jack instead of the USB port.

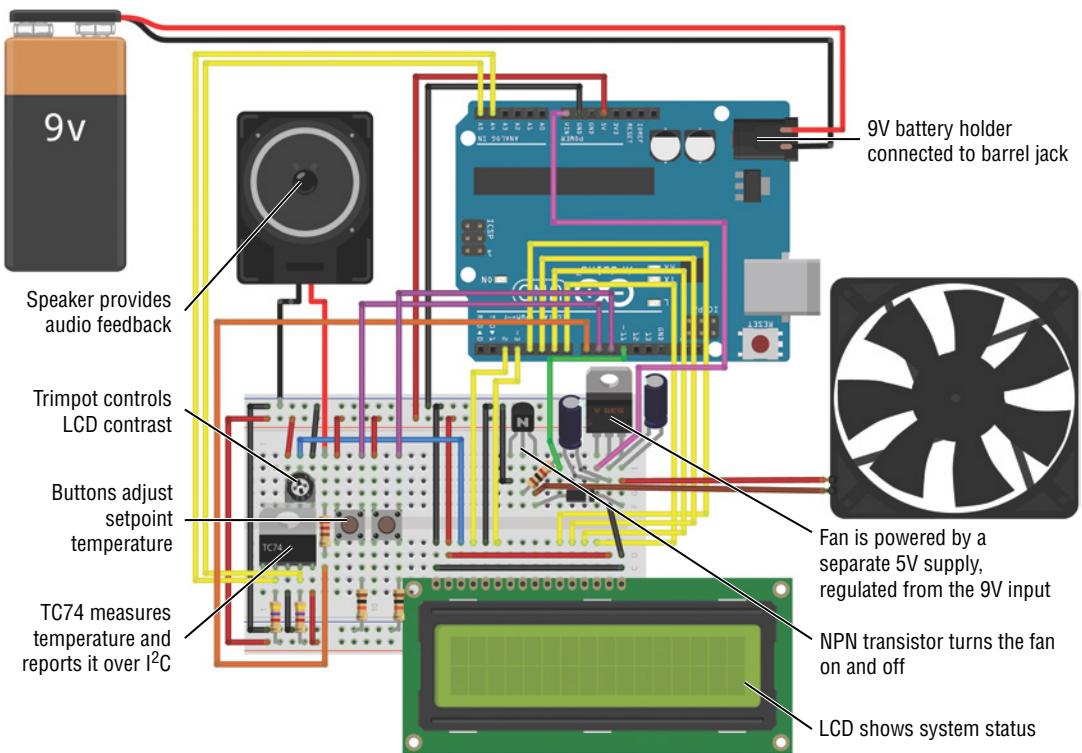


Figure 12-5: LCD thermostat system

Image created with Fritzing

Displaying Data on the LCD

Having some parameters in place beforehand makes writing information to the LCD screen easier. First, use degrees Celsius for the display, and second, assume that you'll always be showing two digits for the temperature. Once the software is running, the LCD display will look something like Figure 12-6.

The Current:and Set:strings are static; they can be written to the screen once at the beginning and left there. Similarly, because the temperatures are assumed to be two digits, you can statically place both °C strings into the correct locations. The current reading will be displayed in position (8,0) and will be updated on every run through the loop(). The desired, or set, temperature will be placed in position (8,1) and updated every time a button is used to adjust its value. The fan indicator in the lower-right corner of the display will be at position (15,1). It should update to reflect the fan's state every time it changes.



Figure 12-6: LCD display

The degree symbol, fan off indicator, and fan on indicator are not part of the LCD character set. Before using them in your sketch, you need to create them as byte arrays at the beginning of your program. As before, visualize the custom characters as pixel arrays first, as shown in Figure 12-7.

Then, define the custom characters to the specifications using the following snippet:

```
//Custom degree character
byte degree[8] = {
    B00110,
    B01001,
    B01001,
    B00110,
    B00000,
    B00000,
    B00000,
    B00000,
};

//Custom "fan on" indicator
byte fan_on[8] = {
    B00100,
    B10101,
    B01110,
    B11111,
    B01110,
    B10101,
    B00100,
    B00000,
};

//Custom "fan off" indicator
byte fan_off[8] = {
    B00100,
    B00100,
    B00100,
    B11111,
    B00100,
```

```
B00100,  
B00100,  
B00000,  
};
```

You write the static parts of the display in `setup()`. Move the cursor to the right locations, and with the LCD library's `write()` and `print()` functions, update the screen, as shown in the following snippet:

```
//Make custom characters  
lcd.createChar(0, degree);  
lcd.createChar(1, fan_off);  
lcd.createChar(2, fan_on);  
  
//Print a static message to the LCD  
lcd.setCursor(0,0);  
lcd.print("Current:");  
lcd.setCursor(10,0);  
lcd.write((byte)0);  
lcd.setCursor(11,0);  
lcd.print("C");  
lcd.setCursor(0,1);  
lcd.print("Set:");  
lcd.setCursor(10,1);  
lcd.write((byte)0);  
lcd.setCursor(11,1);  
lcd.print("C");  
lcd.setCursor(15,1);  
lcd.write(1);
```

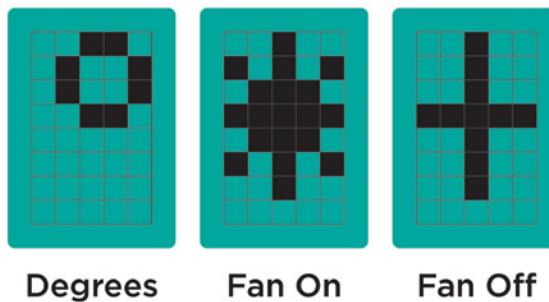


Figure 12-7: LCD thermostat custom characters

You also update the fan indicator and temperature values each time through `loop()`. You need to move the cursor to the right location each time before you update these characters.

In the event that the I²C temperature sensor returns no data, you can halt the program (with `while(1);` as you did in Chapter 10). Instead of sending an error message to the serial monitor, use `lcd.clear()` to empty the LCD screen and return the cursor to the (0,0) position. Then, print an error message: `lcd.print("I2C Error");`.

Adjusting the Set Point with a Button

In Chapter 2, you used a `debounce()` function. Here, you modify it slightly to use it with multiple buttons. One button will increase the set point, and the other will decrease it. You need to define variables for holding the previous and current button states:

```
//Variables for debouncing
boolean lastDownTempButton = LOW;
boolean currentDownTempButton = LOW;
boolean lastUpTempButton = LOW;
boolean currentUpTempButton = LOW;
```

You can modify the `debounce()` function to support multiple buttons. To accomplish this, add a second argument that specifies which button you want to debounce:

```
//A debouncing function that can be used by both buttons
boolean debounce(boolean last, int pin)
{
    boolean current = digitalRead(pin);
    if (last != current)
    {
        delay(5);
        current = digitalRead(pin);
    }
    return current;
}
```

In `loop()`, you want to check both buttons using the `debounce()` function, change the `set_temp` variable as needed, and update the set value that is displayed on the LCD:

```
//Debounce both buttons
currentDownTempButton = debounce(lastDownTempButton, DOWN_BUTTON);
currentUpTempButton = debounce(lastUpTempButton, UP_BUTTON);

//Turn down the set temp
if (lastDownTempButton == LOW && currentDownTempButton == HIGH)
{
    set_temp--;
}
```

```
//Turn up the set temp
else if (lastUpTempButton == LOW && currentUpTempButton == HIGH)
{
    set_temp++;
}
//Print the set temp
lcd.setCursor(8,1);
lcd.print(set_temp);
//Update the button state with the current
lastDownTempButton = currentDownTempButton;
lastUpTempButton = currentUpTempButton;
```

The preceding code snippet first runs the `debounce()` function for each button, and then adjusts the set temperature variable if one of the buttons has been pressed. Afterward, the temperature displayed on the LCD is updated, as are the button state variables.

Adding an Audible Warning and a Fan

In this section, you add code to control the fan and the speaker. Although the LCD showing you live information is nice, you'll often find it useful to have an additional form of feedback to tell you when something is happening—for example, having the speaker beep when the fan turns on. In this example, you use `tone()` paired with `delay()` and a `noTone()` command. You could instead add a duration argument to `tone()` to determine the duration of the sound. You want to make sure that the tone plays only one time per alert condition (and does not beep forever when above the set temperature).

Using a state variable, you can detect when the speaker has beeped and thus keep it from beeping again until after the temperature dips below the set temperature and resets the state variable.

When the fan turns on, an indicator changes on the LCD (represented by the custom character you defined at the top of the program). The following code snippet checks the temperature and controls the speaker, the fan indicator on the LCD, and the fan:

```
//If it's too hot!
if (c >= set_temp)
{
    //Check if the speaker has already beeped
    if (!one_time)
    {
        tone(SPEAKER, 400);
        delay(500);
        one_time = true;
    }
}
```

```
//Turn off the speaker when it's done
else
{
    noTone(SPEAKER);
}
//Turn the Fan on and update display
digitalWrite(FAN, HIGH);
lcd.setCursor(15,1);
lcd.write(2);
}
//If it's not too hot!
else
{
    //Make sure the speaker is off
    //reset the "one beep" variable
    //update the fan state and LCD display
    noTone(SPEAKER);
    one_time = false;
    digitalWrite(FAN, LOW);
    lcd.setCursor(15,1);
    lcd.write(1);
}
```

The `one_time` variable is used to make sure that the beep plays only one time instead of continuously. Once the speaker has beeped for 500 ms at 400 Hz, the variable is set to true and is reset to false only when the temperature drops back below the desired temperature.

Bringing It All Together: The Complete Program

It's time to bring all the parts together into a cohesive whole. You need to make sure that you include the appropriate libraries, define the pins, and initialize the state variables at the top of the sketch. Listing 12-3 shows the complete program. Load it on to your Arduino and compare your results to the demo video showing the system in action.

Listing 12-3

Personal thermostat program—`LCD_thermostat.ino`

```
//Keep yourself cool! This is a thermostat.
//This assumes temperatures are always two digits.

//Include Wire I2C library and set the address
#include <Wire.h>
#define TEMP_ADDR 72
```

```
//Include the LCD library and initialize:  
#include <LiquidCrystal.h>  
LiquidCrystal lcd(2, 3, 4, 5, 6, 7);  
  
//Custom degree character  
byte degree[8] = {  
    B00110,  
    B01001,  
    B01001,  
    B00110,  
    B00000,  
    B00000,  
    B00000,  
    B00000,  
};  
  
//Custom "Fan On" indicator  
byte fan_on[8] = {  
    B00100,  
    B10101,  
    B01110,  
    B11111,  
    B01110,  
    B10101,  
    B00100,  
    B00000,  
};  
  
//Custom "Fan Off" indicator  
byte fan_off[8] = {  
    B00100,  
    B00100,  
    B00100,  
    B11111,  
    B00100,  
    B00100,  
    B00100,  
    B00000,  
};  
  
//Pin Connections  
const int SPEAKER      =8;  
const int DOWN_BUTTON   =9;  
const int UP_BUTTON     =10;  
const int FAN           =11;  
  
//Variables for debouncing  
boolean lastDownTempButton = LOW;  
boolean currentDownTempButton = LOW;
```

```
boolean lastUpTempButton = LOW;
boolean currentUpTempButton = LOW;

int set_temp = 23;      //The Default desired temperature
boolean one_time = false; //Used for making the speaker beep only one time

void setup()
{
  pinMode(FAN, OUTPUT);

  //Create a wire object for the temp sensor
  Wire.begin();

  //Set up the LCD's number of columns and rows
  lcd.begin(16, 2);

  //Make custom characters
  lcd.createChar(0, degree);
  lcd.createChar(1, fan_off);
  lcd.createChar(2, fan_on);

  //Print a static message to the LCD
  lcd.setCursor(0,0);
  lcd.print("Current:");
  lcd.setCursor(10,0);
  lcd.write((byte)0);
  lcd.setCursor(11,0);
  lcd.print("C");
  lcd.setCursor(0,1);
  lcd.print("Set:");
  lcd.setCursor(10,1);
  lcd.write((byte)0);
  lcd.setCursor(11,1);
  lcd.print("C");
  lcd.setCursor(15,1);
  lcd.write(1);
}

//A debouncing function that can be used by multiple buttons
boolean debounce(boolean last, int pin)
{
  boolean current = digitalRead(pin);
  if (last != current)
  {
    delay(5);
    current = digitalRead(pin);
  }
  return current;
}
```

```
void loop()
{
    //Get the Temperature
    Wire.beginTransmission(TEMP_ADDR); //Start talking
    Wire.write(0); //Ask for register zero
    Wire.endTransmission(); //Complete transmission

    //Request 1 byte
    int returned_bytes = Wire.requestFrom(TEMP_ADDR, 1);

    //If no data was returned, then something is wrong.
    if (returned_bytes == 0)
    {
        lcd.clear(); //Clear the display
        lcd.print("I2C Error"); //Show an error
        while(1); //Halt the program
    }

    int c = Wire.read(); //Get the temp in C
    lcd.setCursor(8,0); //Move the cursor
    lcd.print(c); //Print this new value

    //Debounce both buttons
    currentDownTempButton = debounce(lastDownTempButton, DOWN_BUTTON);
    currentUpTempButton = debounce(lastUpTempButton, UP_BUTTON);

    //Turn down the set temp
    if (lastDownTempButton== LOW && currentDownTempButton == HIGH)
    {
        set_temp--;
    }
    //Turn up the set temp
    else if (lastUpTempButton== LOW && currentUpTempButton == HIGH)
    {
        set_temp++;
    }
    //Print the set temp
    lcd.setCursor(8,1);
    lcd.print(set_temp);
    lastDownTempButton = currentDownTempButton;
    lastUpTempButton = currentUpTempButton;

    //It's too hot!
    if (c >= set_temp)
    {
        //So that the speaker will only beep one time...
        if (!one_time)
        {
```

```
tone(SPEAKER, 400);
delay(500);
one_time = true;
}
//Turn off the speaker if it's done
else
{
    noTone(SPEAKER);
}
//Turn the fan on and update display
digitalWrite(FAN, HIGH);
lcd.setCursor(15,1);
lcd.write(2);
}
//It's not too hot!
else
{
    //Make sure the speaker is off, reset the "one beep" variable
    //Update the fan state, and LCD display
    noTone(SPEAKER);
    one_time = false;
    digitalWrite(FAN, LOW);
    lcd.setCursor(15,1);
    lcd.write(1);
}
}
```

You no longer need to have the Arduino and components tethered to the computer to see what the temperature is. Since this project has already been designed to be powered with a battery, simply unplug the USB cable, and your thermostat will continue to run using the connected battery. If you like, you can plug in a battery or wall power supply and place it anywhere in your room.

NOTE To watch a demo video of this personal thermostat in action, check out exploringarduino.com/content2/ch12.

Taking This Project to the Next Level

You could expand the functionality of this program in all kinds of ways. Here are a few suggestions for further improvements you can make:

- Use pulse-width modulation (PWM) to control fan speed so that it changes according to how far over the set temperature you are.

- Add LED indicators that display visual alerts.
- Make the speaker alert into a melody instead of a tone.
- Add a light sensor and automatically adjust the backlight brightness of the display by putting an NPN transistor on the backlight cathode pin and using PWM to drive it based on the ambient brightness.

Summary

In this chapter, you learned the following:

- Parallel LCDs can be interfaced with the Arduino through a standard wiring scheme.
- You can create custom characters for your LCD by generating arbitrary bitmaps.
- You can modify your debounce function from Chapter 2 to debounce multiple buttons.
- You can combine multiple sensors, motors, buttons, and displays into one coherent project.

IV Digging Deeper and Combining Functions

Chapter 13: Interrupts and Other Special Functions

Chapter 14: Data Logging with SD Cards

13

Interrupts and Other Special Functions

Parts You'll Need for This Chapter

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

Half-size or full-size breadboard

Assorted jumper wires

Pushbutton

100 Ω resistor

220 Ω resistors ($\times 3$)

10k Ω resistor

5 mm Common-anode RGB LED

10 μ F 50V electrolytic capacitor

Piezo buzzer

74AHCT14 hex inverting Schmitt trigger

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at: exploringarduino.com/content2/ch13

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

Up to this point, every Arduino program you've written has been synchronous. This presents a few problems, one being that using `delay()` can preclude your Arduino from doing other things. There are a variety of ways to make your Arduino multitask so that you don't waste any valuable processor cycles doing nothing.

In this chapter, you will learn how to leverage both timer and hardware interrupts to make your Arduino sketches asynchronous. Interrupts make it possible to execute code asynchronously by triggering certain events (time elapsed, input state change, and so on). Interrupts, as their name implies, allow you to stop whatever your Arduino is currently doing, complete a different task, and then return to whatever command the Arduino was previously executing. You will also learn how to execute interrupts when timed events occur or when input pins change state. You will use this knowledge to build a nonblocking hardware interrupt system, as well as a sound machine using timer interrupts.

Using Hardware Interrupts

Hardware interrupts are triggered depending on the state (or change in state) of an input I/O pin. Hardware interrupts can be particularly useful if you want to change some state variable within your code without having to constantly poll the state of a button. In some previous chapters, you used a software debounce routine along with a check for the button state each time through the loop. This works well if the other content in your loop does not take a long time to execute.

Suppose, however, that you want to run a procedure in your loop that takes a while. For example, perhaps you want to slowly ramp up the brightness of an LED or the speed of a motor using a `for()` loop with some `delay()` statements. If you want button presses to adjust the color or speed of such an LED fade, you will miss any presses of the button that occur while the `delay()` is happening. Ordinarily, human reaction time is slow enough that you can execute many functions within the `loop()` of an Arduino program, and can poll a button once every time you go through the loop without missing the button press. However, when there are slow components to your code within the `loop()`, you risk missing external inputs.

That's where interrupts come in. Select pins on your Arduino can function as external hardware interrupts. Hardware within the microcontroller knows the state of these pins and can report their values to your code asynchronously. Hence, you can execute your main program, and have it interrupted to run a special function whenever an external interrupt event is detected. This interrupt can happen anywhere in the program's execution. Figure 13-1 shows what this process looks like in practice.

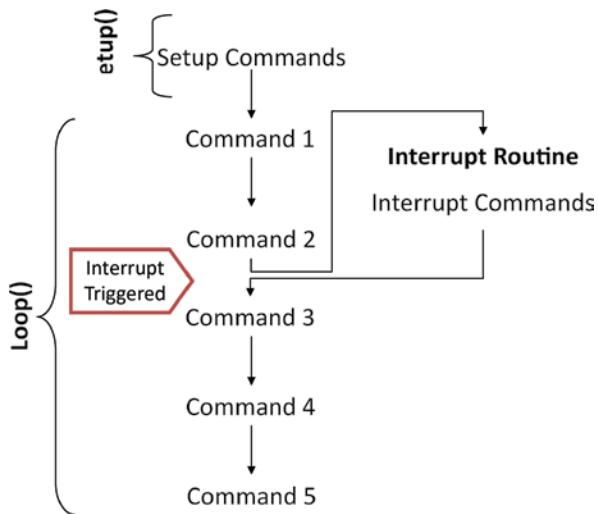


Figure 13-1: How an external interrupt affects program flow

Knowing the Tradeoffs Between Polling and Interrupting

Hardware interrupts are an alternative to repeatedly polling external inputs in `loop()`. They are not better or worse; instead, there are tradeoffs between using the two. When designing a system, you must consider all your options and choose the appropriate one for your application. This section describes the main differences between polling inputs and using interrupts so that you can decide for yourself which option is best for your particular project.

Ease of Implementation (Software)

Thanks to the excellent programming language that has been constructed for the Arduino, attaching external interrupts in software is actually very straightforward. Using polling to detect inputs to the Arduino is still easier because all you have to do is call `digitalRead()`. If you don't need to use hardware interrupts, don't bother to use them over polling, because it does require you to write a little more code.

Ease of Implementation (Hardware)

For most digital inputs, the hardware for an input that triggers via polling or interrupting is exactly the same, because you are just looking for a state change in the input. However, in one situation, you need to adjust your hardware if you are using an edge-triggered interrupt: bouncy inputs. As discussed in Chapter 2, “Digital Inputs, Outputs, and Pulse-Width Modulation,” many buttons (something you will commonly want to use to trigger an input) bounce when you press them. This can be a significant problem

because it will cause the interrupt routine to trigger multiple times when you want it to trigger only once. What's worse, it is not possible to use the software debouncing function that you had previously written because you cannot use a `delay()` in an interrupt routine. Therefore, if you need to use a bouncy input with a hardware interrupt, you need to first debounce it with hardware. If your input does not bounce (like a rotary encoder, or a digital signal output from another integrated circuit), then you don't have to worry, and your hardware will be no different than it was with a polling setup.

Multitasking

One of the primary reasons for using interrupts is to enable pseudo-multitasking. You can never achieve true multitasking on an Arduino because there is only one microcontroller unit (MCU), and because it can execute only one command at a time. However, because it executes commands so quickly, you can use interrupts to “weave” tasks together so that they appear to execute simultaneously. For instance, using interrupts, you can be dimming LEDs with `delay()` while appearing to simultaneously respond to a button input that adjusts the fade speed or color. When polling an external input, you can only read the input once you get to a `digitalRead()` in your program loop, meaning that having slower functions in your program could make it hard to effectively listen for an input.

Acquisition Accuracy

For certain fast acquisition tasks, interrupting is an absolute necessity. For example, suppose that you are using a rotary encoder. Rotary encoders are commonly mounted on direct current (DC) motors and send a pulse to the microcontroller every time some percentage of a revolution is completed. You can use them to create a feedback system for DC motors that allows you to keep track of their position, instead of just their speed. This enables you to dynamically adjust speed based on torque requirements or to keep track of how much a DC motor has moved.

However, you need to be absolutely sure that every pulse is captured by the Arduino. These pulses are fairly short (much shorter than a pulse created by you manually pushing a button) and can potentially be missed if you check for them by polling within `loop()`. In the case of a rotary encoder that triggers only once per revolution, missing a pulse could cause your program to believe that the motor is moving at half of its actual speed! To ensure that you capture timing for important events like this, using a hardware input is a must. If you are using a slowly changing input (like a button), polling might suffice.

Understanding the Arduino Hardware Interrupt Capabilities

With most Arduino boards, you can use only certain pins as interrupts. To see which pins are interrupt-capable on your Arduino, consult the interrupt documentation on the Arduino website, at blum.fyi/arduino-attach-interrupt.

To configure a pin to act as a hardware interrupt, you'll use `attachInterrupt()`. The first argument is the ID of the interrupt—this is *not* necessarily the same as the pin number. Thankfully, the Arduino language includes a helper function for easily obtaining the interrupt ID from the physical pin number: `digitalPinToInterrupt()`. Simply pass the digital pin number to that function and it will determine the appropriate interrupt ID based on whichever Arduino you are compiling your sketch for. So, if you wanted to attach a button to pin 2 of the Arduino Uno (or an equivalent clone that also used the ATmega328P), the first argument to the `attachInterrupt()` function would be `digitalPinToInterrupt(2)`. Using one function's output directly as an argument in another function is perfectly acceptable, and helps to keep your code tidy.

Hardware interrupts work by “attaching” interrupt pins to certain functions. So, the second argument of `attachInterrupt()` is a function name. If you want to toggle the state of a Boolean variable every time an interrupt is triggered, you might write a function like this, which you pass to the second argument of `attachInterrupt()`:

```
void toggleLed()
{
    var = !var;
}
```

When this function is called, the Boolean `var` is toggled to the opposite of its previous state, and the rest of your program continues running where it left off.

The final argument passed to `attachInterrupt()` is the trigger mode. Arduino interrupts can be triggered on LOW, CHANGE, RISING, or FALLING. (A select set of Arduinos can also be triggered on HIGH.) CHANGE, RISING, and FALLING are the most common events to trigger on because they cause an interrupt to execute exactly one time when an external input changes state, like a button going from LOW to HIGH. The transition from LOW to HIGH is RISING, and from HIGH to LOW is FALLING. It is less common to trigger on LOW or HIGH because doing so would cause the interrupt to fire continuously as long as that state is true, effectively blocking the rest of the program from running.

Bringing all this functionality together, if you wanted to execute the `toggleLED()` function every time a button on an Uno's pin 2 went from LOW to HIGH, you would add this code to your `setup()`:

```
attachInterrupt(digitalPinToInterrupt(2), toggleLED, RISING);
```

Building and Testing a Hardware-Debounced Button Interrupt Circuit

To test out your newfound knowledge, you will construct a circuit with an RGB LED and a hardware-debounced pushbutton. The LED fades up and down on a selected color. When the button is pressed, the LED immediately changes the fade color to another one, while using `delay()` to accomplish the fading.

Creating a Hardware-Debouncing Circuit

As you learned in Chapter 2, most buttons actually “bounce” up and down when you press them. This action presents a serious problem when you are using hardware interrupts because it might cause an action to be triggered more times than you intended. Luckily, you can debounce a button in hardware so that you always get a clean signal going into your microcontroller.

First, take a look at an ordinary button signal that is hooked up using a pull-up resistor. Using a pull-up resistor instead of a pull-down resistor does exactly what you would expect: By default, the button state is pulled high by the resistor; when the button is pressed, it connects ground to the I/O pin and the input signal goes LOW. You use a pull-up circuit instead of a pull-down circuit in this example and invert the output later. Figure 13-2 shows the button signal being probed with an oscilloscope (a piece of equipment for inspecting the voltage at the probed location in the circuit, in the time domain). When I press the button, it bounces up and down before finally settling at a LOW state (zero volts).

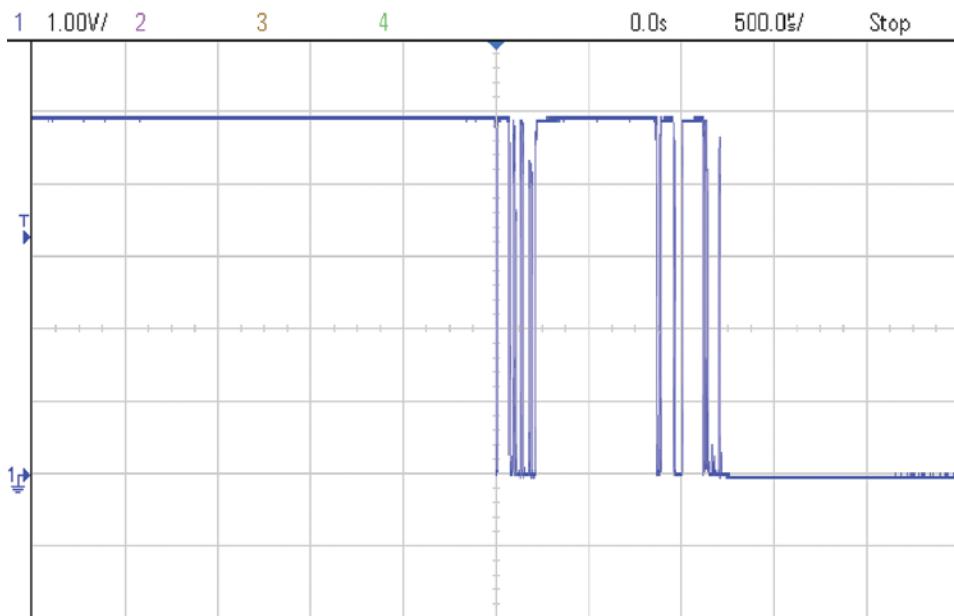


Figure 13-2: Ordinary pushbutton bouncing before settling

If you trigger an interrupt off the signal shown in Figure 13-2, it will execute the interrupt function several times while the signal bounces up and down. But, using something called a resistor-capacitor network (commonly called an RC circuit), you can prevent this problem.

If you connect a resistor in between the switch and the microcontroller input, and a capacitor from the microcontroller input to ground, it creates a resistor-capacitor filter network. While the switch is not pressed, the capacitor charges up to the VCC voltage (5V) through the added resistor and the pull-up resistor. Think of the capacitor as an energy reservoir, which fills up with electrical charge from the positive voltage rail. While the switch is unpressed, this reservoir fills up. When you push the button, the capacitor starts to discharge its energy to ground through the path of the resistor and the now-closed switch.

Because the capacitor is not instantaneously discharging, it can hold the observed output voltage up near 5V while the switch opens and closes for milliseconds at a time. Each time the switch reopens, the capacitor recharges. Only after the switch has settled to an unbouncing, closed state, will enough time pass to fully discharge the capacitor down to ground (zero volts). As a result of this process, you'll get a signal that transitions only one time from HIGH to LOW voltage. The values of the resistor and capacitor determine how long the discharge and recharge will take, and how long of a bounce the circuit can "ride over" without transitioning to a LOW voltage. The schematic for this circuit is shown in Figure 13-3.

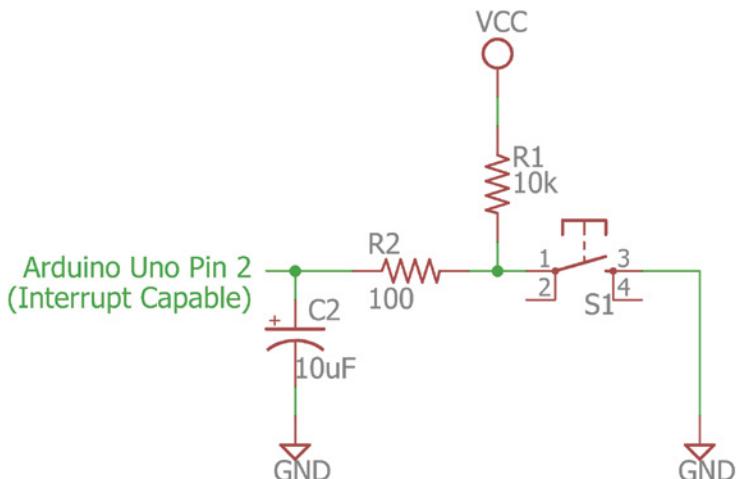


Figure 13-3: Creating a debounce circuit—adding an RC filter network

Created with EAGLE

The resistor in series with the switch (R2 in Figure 13-3) prevents the capacitor from discharging (nearly) instantly. This adds a discharge curve to your output. You can see this effect in the oscilloscope in Figure 13-4.

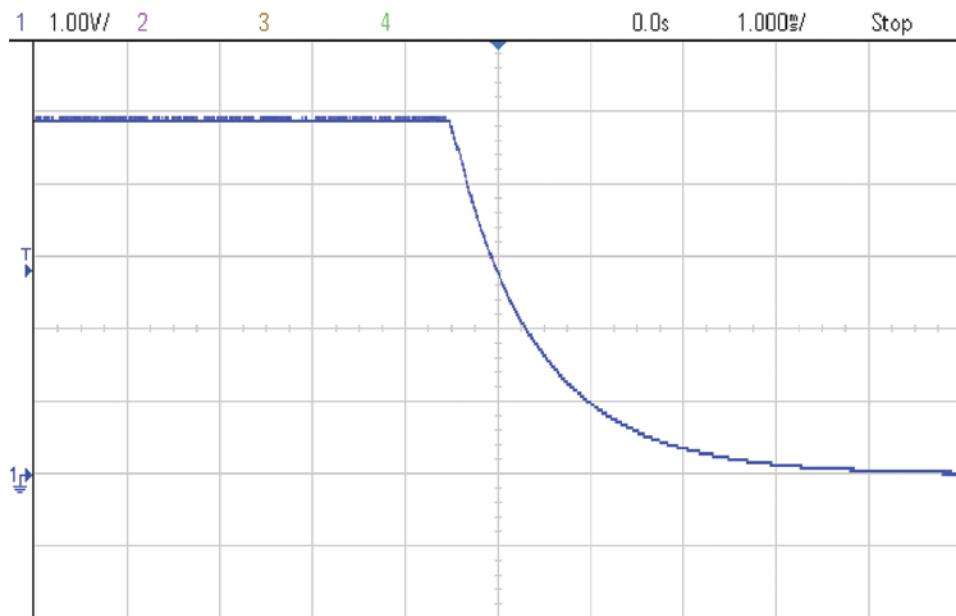


Figure 13-4: Signal bouncing removed with an RC circuit

The RC circuit that you just created will make a “curved” input signal to the Arduino’s I/O pin. My interrupt is looking for an edge, which is detected when a shift from high to low or from low to high occurs at a certain speed.

Most modern microcontrollers are already designed to handle slowly rising or falling digital input voltages. When looking at the datasheet of the ATMega microcontroller, for example, you’ll find that the input pin low and high voltage thresholds are different. When an input signal is transitioning from high to low, it must drop below 0.2VCC to register as a logic LOW. When an input signal is going from low to high, it must rise above 0.7VCC to be registered as a logic HIGH. VCC represents the supply voltage of the chip (5V in my setup). This gap ensures that the value does not flutter during the transition step, and is called *hysteresis*.

Although the Arduino interrupts are capable of triggering off of this slowly changing signal, it’s useful to understand exactly how that feat is accomplished by utilizing a Schmitt trigger in your circuit to accomplish this independently of the Arduino. *Schmitt triggers* are integrated circuits (ICs) that create a sharp edge when the input signal surpasses a certain threshold. The output from the trigger can then be fed right into the Arduino I/O pin. In this case, you use an inverting Schmitt trigger, the 74AHCT14 IC. This chip has six separate inverting Schmitt triggers in it, but you use only one. Many

manufacturers make functionally identical logic chips in the 7400 series. Figure 13-5 shows the datasheet image of a hex inverting Schmitt trigger chip from STMicroelectronics.

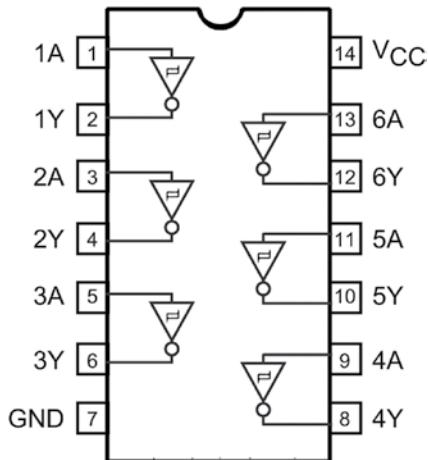


Figure 13-5: Inverting Schmitt trigger pin-out

Credit: © STMicroelectronics. Used with permission.

The output from your debounce circuit will go through one of these inverting Schmitt triggers before finally being fed into the Arduino. The resulting circuit diagram looks like Figure 13-6.

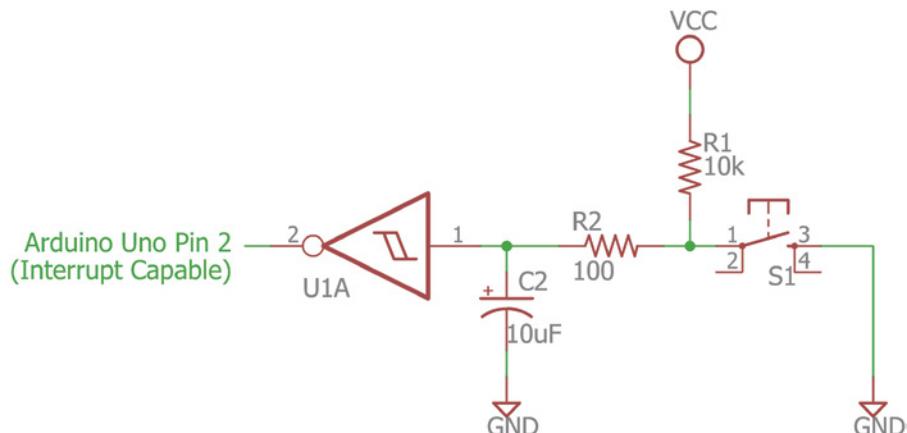


Figure 13-6: Final step for creating a debounce circuit—adding an inverting Schmitt trigger
Created with EAGLE

Because this is an inverting trigger, the signal will also be flipped. This means that when the button is held down, the final signal will be a logical high, and vice versa. So, in the next step, when you write the code, you want to look for a rising edge to detect when the button is first pressed. The final output looks like a nice, clean, bounce-free signal (see Figure 13-7).

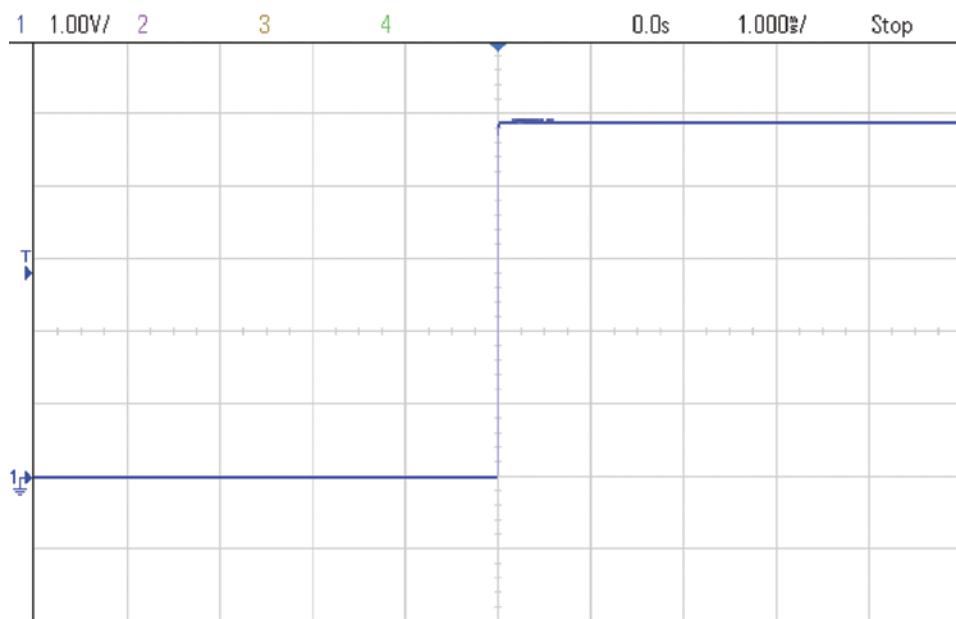


Figure 13-7: Final output of debounce circuit

You now have a nice, clean signal that you can feed into your hardware interrupt function!

Assembling the Complete Test Circuit

From a schematic level, you now understand how to wire up a button debouncer. For the tests that you'll run momentarily, you will use an RGB LED in tandem with a button to test your hardware-debouncing and interrupt code. Wire up a complete circuit as shown in the wiring diagram in Figure 13-8.

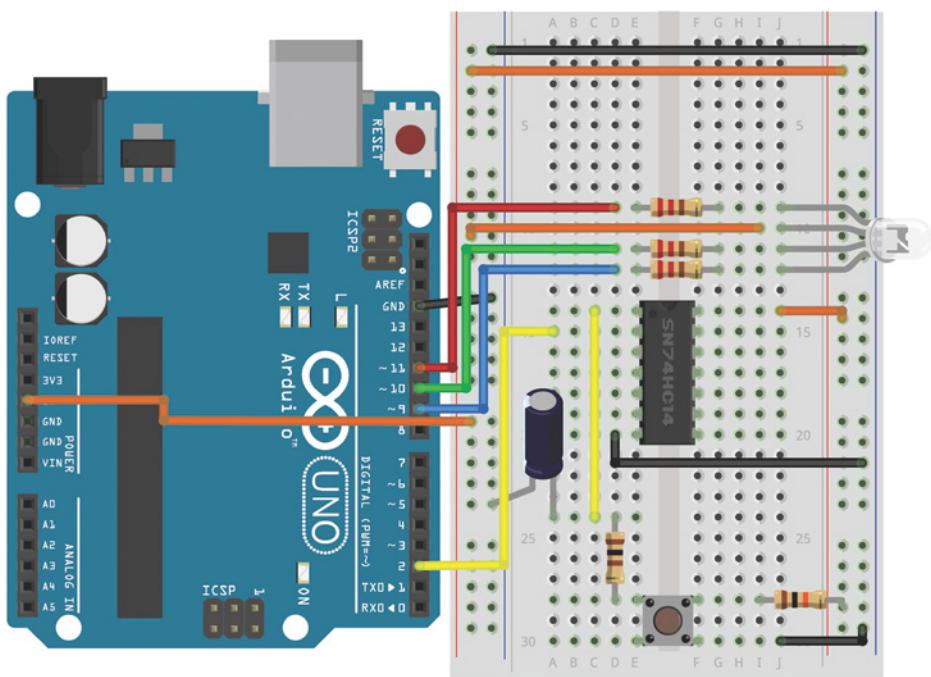


Figure 13-8: Complete hardware interrupt wiring diagram

Created with Fritzing

Writing the Software

It's now time to write a simple program to test both your debouncing and the hardware interrupt capabilities of the Arduino. The most obvious and useful implementation of hardware interrupts on the Arduino is to allow you to listen for external inputs even while running timed operations that use `delay()`. There are many scenarios where this might happen, but a simple one occurs when fading an LED using pulse-width modulation (PWM) via `analogWrite()`. In this sketch, you have one of the three RGB LEDs always fading up and down slowly from 0 to 255 and back again. Every time you press the button, the color that is being faded immediately changes. This would not be possible using polling because you would only be checking the button state after completing a fade cycle; you would almost certainly miss the button press.

Before writing the described program, you need to understand *volatile* variables. Whenever a variable will be changing within an interrupt, it must be declared as *volatile*. This is necessary to ensure that the compiler handles the variable correctly. To declare a variable as volatile, simply add `volatile` before the declaration:

```
volatile int selectedLED = 9;
```

To ensure that your Arduino is listening for an interrupt, you use `attachInterrupt()` in `setup()`. The inputs to the function are the ID of the interrupt (as returned by the `digitalPinToInterrupt()` function), the function that should be run when an interrupt occurs, and the triggering mode (RISING, FALLING, and so on). In this program, the button is connected to pin 2, which maps to interrupt ID 0, although the `digitalPinToInterrupt()` abstracts away that detail for you. The program runs the `swap()` function when triggered, and it triggers on the rising edge:

```
attachInterrupt(digitalPinToInterrupt(2), swap, RISING);
```

You need to write `swap()` and add it to your program; this is included in the complete program code shown in Listing 13-1. That's all you have to do! After you've attached the interrupt and written your interrupt function, you can write the rest of your program to do whatever you want. Whenever the interrupt is triggered, the rest of program pauses, the interrupt function runs, and then your program resumes where it left off. Because interrupts pause your program, they are generally very short and do not contain delays of any kind. In fact, `delay()` does not even work inside of an interrupt-triggered function. Understanding all of this, you can now write the following program to cycle through all the LED colors and switch them based on your button press.

Listing 13-1

Hardware interrupts for multitasking-hw_multitask.ino

```
//Use a Hardware-Debounced Switch to Control an Interrupt

//Button pins
const int BTN    = 2; //Output of debounced button on pin 2
const int RED    = 11; //Red Cathode LED on pin 11
const int GREEN  = 10; //Green Cathode LED on pin 10
const int BLUE   = 9; //Blue Cathode LED on pin 9

//Volatile variables can change inside interrupts
volatile int selectedLED = RED;
```

```
void setup()
{
    pinMode(RED, OUTPUT);
    pinMode(GREEN, OUTPUT);
    pinMode(BLUE, OUTPUT);

    //Turn the RGB LED off to start
    //(Inverted because we are controlling the cathode)
    digitalWrite(RED, HIGH);
    digitalWrite(BLUE, HIGH);
    digitalWrite(GREEN, HIGH);

    //The pin is inverted, so we want to look at the rising edge
    attachInterrupt(digitalPinToInterrupt(BTN), swap, RISING);
}

void swap()
{
    //Turn off the current LED (Common Anode, so HIGH is Off)
    digitalWrite(selectedLED, HIGH);
    //Then, choose a new one.
    if (selectedLED == GREEN)
        selectedLED = RED;
    else if (selectedLED == RED)
        selectedLED = BLUE;
    else if (selectedLED == BLUE)
        selectedLED = GREEN;
}

void loop()
{
    //Ramp Brightness Up
    //(Inverted because we are controlling the cathode)
    for (int i=255; i>=0; i--)
    {
        analogWrite(selectedLED, i);
        delay(10);
    }
    //Ramp Brightness Down
    //(Inverted because we are controlling the cathode)
    for (int i=0; i<=255; i++)
    {
        analogWrite(selectedLED, i);
        delay(10);
    }
    delay(1000);
}
```

When you load this program, your RGB LED should start fading back and forth in one color. Every time you press the button, a new color will take over, with the same brightness as the previous color.

NOTE You can watch a demo video of the hardware-interrupted Arduino with button debouncing at exploringarduino.com/content2/ch13.

Using Timer Interrupts

Hardware interrupts are not the only kind of interrupt you can trigger on an Arduino; there are also timer-based interrupts. The ATmega328P (the chip used in the Uno) has three hardware timers, which you can use for all kinds of things. In fact, the default Arduino library already uses these timers to increment `millis()`, operate `delay()`, and enable PWM output with `analogWrite()`. You can also take manual control of one of these timers to initiate timed functions, generate arbitrary PWM signals on any pin, and more. In this section, you learn how to use a third-party library (the `TimerOne` library) to take manual control of the 16-bit Timer1 on the ATmega328P-based Arduinos. Similar libraries are available for doing these tricks on the Leonardo and other Arduino boards, but this section focuses on ATmega328P-based Arduinos.

NOTE Timer1 is used to enable PWM output on pins 9 and 10; as a result, when you use this library, you will be unable to run `analogWrite()` on those pins.

Understanding Timer Interrupts

Just like a timer on your watch, timers on the Arduino count up from zero, incrementing with every clock cycle of the oscillating crystal that drives the Arduino. Timer1 is a 16-bit timer, meaning that it can count up from zero to $2^{16} - 1$, or 65,535. Once that number is reached, it resets back to zero and starts counting again. How quickly it reaches that number depends on the clock divider. With no divider, the clock would go through 16 million cycles per second (16 MHz), and would overflow and reset this counter many times per second. However, you can “divide” the clock, which is an approach taken by many underlying Arduino functions and libraries. The `TimerOne` library abstracts away much of the complexity of dealing with the timer, allowing you to simply set a trigger period. When you use the timer, a function can be triggered every set number of microseconds.

Getting the Library

To get started, you'll follow the same process that you learned about in Chapter 11, "The SPI Bus and Third-Party Libraries," to install the third-party library for the SPI accelerometer. From within the Arduino IDE, navigate to Sketch > Include Library > Manage Libraries. Install the library called TimerOne. You are now ready to take control of Timer1 with your Arduino.

NOTE The TimerOne library is maintained by PJRC. You can visit their documentation page for the library at blum.fyi/pjrc-timerone.

Executing Two Tasks Simultaneously(ish)

It's important to keep in mind that there is no such thing as true simultaneous execution on an Arduino. Interrupts merely make it seem like multiple things are happening at the same time, by allowing you to switch between multiple tasks extremely quickly. Using the TimerOne library you just installed, you can make an LED blink using the timer while you execute other functions within `loop()`. At the end of the chapter, you will execute serial print statements in the main loop with delays, while using timer interrupts to control sounds simultaneously.

To confirm that the library is installed properly, you can load the program shown in Listing 13-2 on to an Arduino Uno (with no other components connected). It should blink the onboard LED connected to pin 13. This LED will blink on and off every second and is controlled by the timer. If you put any other code in `loop()`, it will appear to execute simultaneously.

Listing 13-2

Simple timer interrupt blink test-timer1.ino

```
//Using Timer Interrupts with the Arduino
#include <TimerOne.h>
const int LED=13;

void setup()
{
    pinMode(LED, OUTPUT);
    Timer1.initialize(1000000); //Set a timer of length 1000000 microseconds
    Timer1.attachInterrupt(blinky); //Runs "blinky" on each timer interrupt
}
```

```
void loop()
{
    //Put any other code here.

}

//Timer interrupt function
void blinky()
{
    digitalWrite(LED, !digitalRead(LED)); //Toggle LED State
}
```

When you call `Timer1.initialize`, you are setting the period of the timer in microseconds. In this case, it has been set to trigger every 1 second. (There are a million microseconds in 1 second.) When you run `Timer1.attachInterrupt()`, you can choose a function that will be executed every time the specified period elapses. The function you call must take less time to execute than the time between executions (or else you'll starve your main loop of CPU resources).

Now that you can implement both timer and hardware interrupts, you can develop hardware that takes advantage of both of them. You will do this in the next section.

Building an Interrupt-Driven Sound Machine

To finalize and confirm your understanding of hardware and timer interrupts, you will build a “sound machine” that enables you to step through and listen to multiple octaves of each note on a musical major scale. The system uses a hardware-debounced pushbutton interrupt to select the note played (C, A, B, and so forth). A timer interrupt steps through all the octaves of the note in order until the next note is selected with the push button. In `loop()`, you can run a simple serial debugging interface that prints the current key and pitch to the screen of your computer. The notes start at octave 2 (it doesn’t sound very good below that) and go up toward octave 6.

Computing the frequency of each octave is easy once you know the initial frequency. Consider C, for example. C2, where you will be starting, has a frequency of about 65 Hz. To get to C3 (130 Hz), you multiply the frequency of C2 by 2. To get to C4, you multiply by 2 again, for 260 Hz. The frequency of each step can be computed as a power of 2 related to the initial frequency. Knowing this, you can construct a timer interrupt that increases by a power of 2 with each time step.

You can switch between notes in the same way you switched between LED colors in the earlier example with the pushbutton. Assign base frequencies to each note, and switch which base frequency is used for tone() every time the button is pressed.

Sound Machine Hardware

The hardware setup here is very simple. Keep the debounced button wired as you had it in the RGB LED example, and add a speaker to pin 12 through a 220Ω resistor. I used a piezo buzzer, but you can use a larger speaker as well. The circuit should look like the one shown in Figure 13-9.

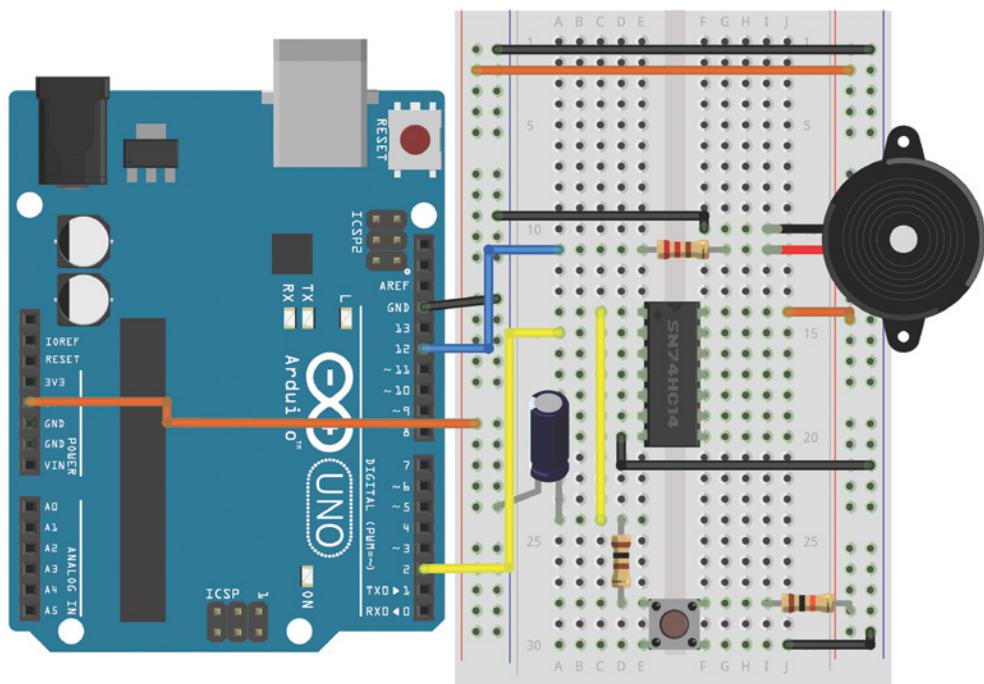


Figure 13-9: Sound machine wiring diagram

Created with Fritzing

Sound Machine Software

The software for the sound machine utilizes software and hardware interrupts in addition to serial communication and tone() to control a speaker. Load the code from Listing 13-3 on to your Arduino and press the button on the breadboard to cycle through base frequencies. You can open the serial monitor to see the frequency that is currently playing.

Listing 13-3

Sound machine code—fun_with_sound.ino

```
//Use Hardware and Timer Interrupts for Fun with Sound

//Include the TimerOne library
#include <TimerOne.h>

//Button pins
const int BTN      = 2; //Output of debounced button on pin 2
const int SPEAKER = 12; //Speaker on pin 12

//Music keys
#define NOTE_C 65
#define NOTE_D 73
#define NOTE_E 82
#define NOTE_F 87
#define NOTE_G 98
#define NOTE_A 110
#define NOTE_B 123

//Volatile variables can change inside interrupts
volatile int key = NOTE_C;
volatile int octave_multiplier = 1;

void setup()
{
    //Set up serial
    Serial.begin(9600);

    pinMode (SPEAKER, OUTPUT);

    //The pin is inverted, so we want to look at the rising edge
    attachInterrupt(digitalPinToInterrupt(BTN), changeKey, RISING);

    //Set up timer interrupt
    Timer1.initialize(500000);           // Trigger every 0.5 seconds
    Timer1.attachInterrupt(changePitch); //Runs "changePitch" on each interrupt
}

void changeKey()
{
    octave_multiplier = 1;
    if (key == NOTE_C)
        key = NOTE_D;
    else if (key == NOTE_D)
        key = NOTE_E;
```

```
else if (key == NOTE_E)
    key = NOTE_F;
else if (key == NOTE_F)
    key = NOTE_G;
else if (key == NOTE_G)
    key = NOTE_A;
else if (key == NOTE_A)
    key = NOTE_B;
else if (key == NOTE_B)
    key = NOTE_C;
}

//Timer interrupt function
void changePitch()
{
    octave_multiplier = octave_multiplier * 2;
    if (octave_multiplier > 16) octave_multiplier = 1;
    tone(SPEAKER, key*octave_multiplier);
}

void loop()
{
    Serial.print("Key: ");
    Serial.print(key);
    Serial.print(" Multiplier: ");
    Serial.print(octave_multiplier);
    Serial.print(" Frequency: ");
    Serial.println(key*octave_multiplier);
    delay(100);
}
```

You can easily find the music keys defined at the beginning with a search on the Internet. They are the frequencies of the second octave of those notes. Note that the key and octave_multiplier must be declared as volatile integers because they are going to be changed within interrupt routines. changeKey() is called every time the button interrupt is triggered. It changes the octave's base value by moving from key to key. changePitch() calls tone() to set the frequency for the speaker. It is triggered every .5 seconds by the timer interrupt. Each time it is triggered, it doubles the frequency of the original note until it reaches 16 times its original frequency. It then loops back around and starts again at the base frequency for the current note. Within loop(), the current key, multiplier, and frequency are printed to the serial monitor every 100 milliseconds

NOTE To watch a demo video of the sound machine, check out exploringarduino.com/content2/ch13.

Summary

In this chapter, you learned the following:

- There are tradeoffs between polling inputs and using interrupts.
- Different Arduinos have different interrupt capabilities. Some Arduino boards can interrupt on any I/O pin, but other Arduinos have particular interrupt-enabled pins. The Uno only has two hardware interrupt-capable pins.
- Buttons can be debounced in hardware using an RC circuit and a Schmitt trigger.
- The Arduino can be made to respond to inputs asynchronously by attaching interrupt functions.
- You can install a third-party timer library to add timer interrupt functionality to the Arduino.
- You can combine timer interrupts, hardware interrupts, and polling into one program to enable pseudo-simultaneous code execution.

14

Data Logging with SD Cards

Parts You'll Need for This Chapter

Arduino Uno or Adafruit METRO 328

USB cable (Type A to B for Uno, Type A to Micro-B for METRO)

Assorted jumper wires

Sharp GP2Y0A21YK0F IR distance sensor with JST cable

Adafruit Arduino data logging shield with header pins

CR1220 12 mm 3V coin cell battery

SD/MicroSD Memory Card (8 GB SDHC recommended)

5V 1A USB port wall power supply

Computer with SD card reader (or USB SD card reader)

Painter's tape and/or 3M Command Strips

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch14

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

Countless examples of Arduinos are being used to log weather conditions, atmospheric conditions from weather balloons, building entry data, electrical loads in buildings, and much more. Given their small size, minimal power consumption,

and ease of interfacing with a vast array of sensors, Arduinos are an obvious choice for building data loggers, which are devices that record and store information over a period of time. Data loggers are often deployed into all kinds of environments to collect environmental or user data and to store it into some kind of nonvolatile memory, such as an SD card.

In this chapter, you will learn everything you could want to know about interfacing with an SD card from an Arduino. You will learn how to both write data to a file and read existing information off an SD card. You will use a real-time clock to add accurate timestamps to your data. You will also learn how to display the data on your computer after you have retrieved it.

NOTE On the content web page for this chapter, you'll find a video tutorial about data logging with the Arduino, as well as a more advanced tutorial about logging location with a GPS receiver: exploringarduino.com/content2/ch14.

Getting Ready for Data Logging

Data logging systems are very simple. They generally consist of some kind of acquisition system, such as analog sensors, to obtain data. They also contain some kind of memory for storing sizeable quantities of data over a long period of time.

This chapter highlights a few common ways that you can use an SD card with your Arduino to record useful data. Actually, there are many uses for data logging. Here is a brief list of projects in which you could use it:

- A weather station for tracking light, temperature, and humidity over time
- A GPS tracker and logger that keeps a record of where you've been over the course of a day
- A temperature monitor for your desktop computer to keep track of which components are heating up the most
- A light logger that keeps track of when, and for how long, the lights are left on in your home or office

Later in this chapter, you will create a data logging system that uses an infrared (IR) distance sensor to create a log of when people enter and exit a room.

Formatting Data with CSV Files

CSV, or comma-separated value, files will be the format of choice for storing data with your SD card. CSV files are easy to implement with a microcontroller platform and can easily be read and parsed by a wide range of desktop applications, making them well suited for this kind of task. A standard CSV file generally looks something like this:

```
Date,Time,Value1,Value2  
2019-08-31,12:00,125,255  
2019-08-31,12:30,100,200  
2019-08-31,13:00,110,215
```

Rows are delimited by new lines, and columns are delimited by commas. Because commas are used to distinguish columns of data, the main requirement is that your data cannot have commas within it. Furthermore, each row should always have the same number of entries. When opened with a spreadsheet program on your computer, the preceding CSV file would look something like Table 14-1.

Table 14-1: An Imported CSV File

Date	Time	Value1	Value2
2019-08-31	12:00	125	255
2019-08-31	12:30	100	200
2019-08-31	13:00	110	215

Because CSV files are just plain text, your Arduino can easily write to them using familiar `print()` and `println()`-style commands. Conversely, Arduinos can also parse CSV files with relative ease by looking for newline and command delimiters to find the right information.

Preparing an SD Card for Data Logging

Before you start logging data with your Arduino, you need to prepare the SD card you plan to use. Which kind of SD card you use will depend on the kind of shield you are using. Some will use full-size SD cards, while others will use micro SD cards. Most micro SD cards ship with an adapter that lets you plug them into standard-sized SD card readers. To complete the exercises in this chapter, you need an SD card reader for your computer (either built-in or external).

Most new SD cards will already be properly formatted and ready to use with an Arduino. If your card is not new, or already contains files, you need to first format the card in either FAT16 (sometimes just called FAT) or FAT32 format. Cards less than or equal to 2 GB should be formatted as FAT16, and larger cards should be formatted as FAT32. In this chapter, the examples use an 8 GB micro SD card formatted as FAT32. This card will be installed into an Adafruit data logging shield using a micro SD-to-SD adapter. Note that formatting the card removes everything on it, but doing so ensures that it is ready for use with your Arduino. If your SD card is new, you can skip these steps and come back to complete them only if you have difficulty accessing the card from the Arduino when you run the sketch later in this chapter. Instructions for formatting with Windows, Mac OS, or Linux follow.

Formatting Your SD Card Using a Windows PC

1. Insert the SD card into your card reader; it then appears in This PC (see Figure 14-1).

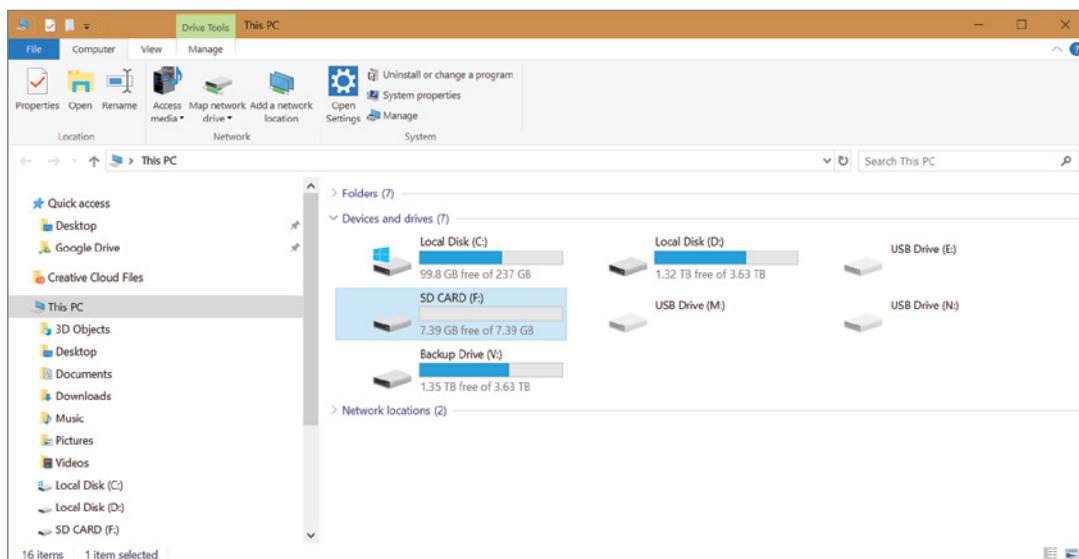


Figure 14-1: SD card shown in This PC

2. Right-click the card (it probably has a different name), and select the Format option (see Figure 14-2). A window appears with options for formatting the card.

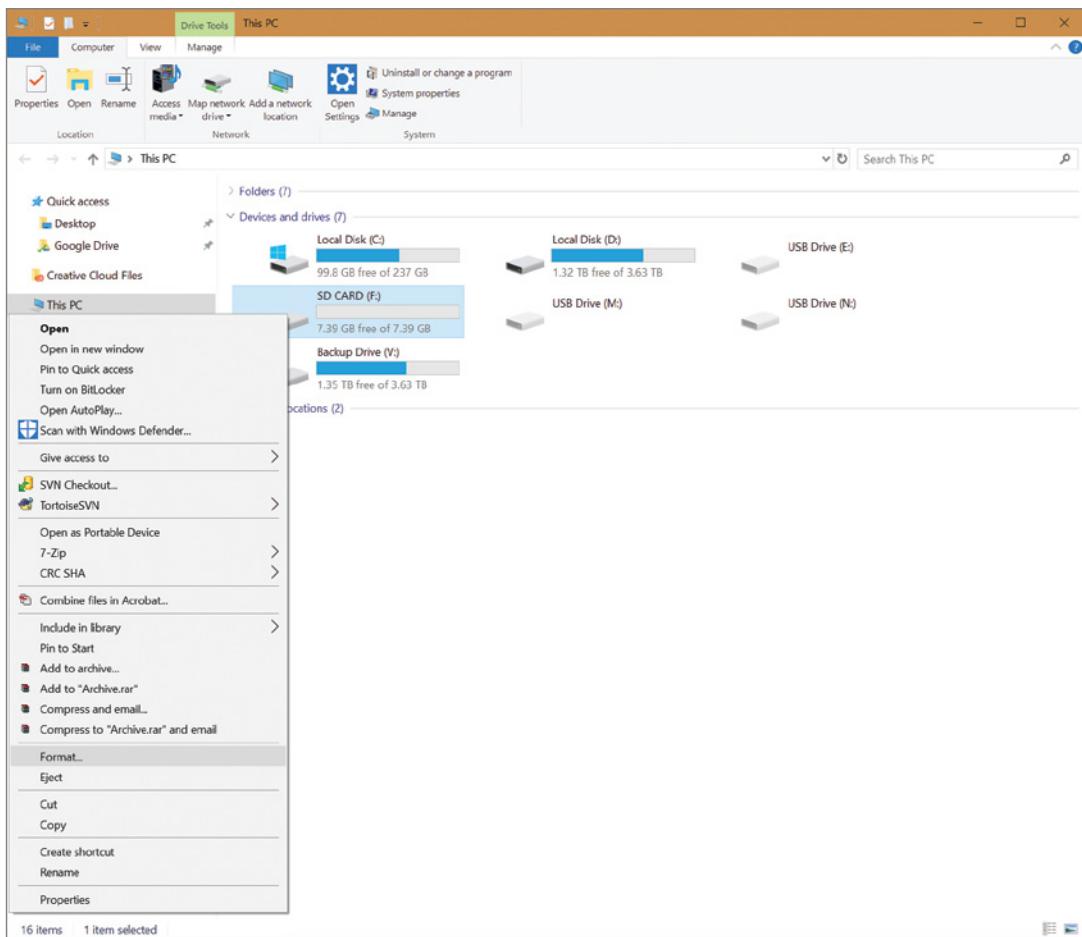


Figure 14-2: Format option selected

3. Choose the file system type (FAT for cards 2 GB and under, and FAT32 for larger cards), use the default allocation size, and choose a volume label. (I chose LOG, but you can choose whatever you want.) Figure 14-3 shows the configuration for an 8 GB card.

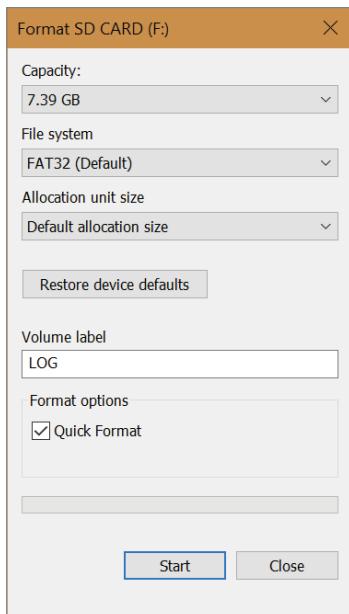


Figure 14-3: Format option window

4. Click the Start button to format the SD card. After completion, the card is ready to use with your Arduino.

Formatting Your SD Card Using Mac OS

1. Insert the SD card. Use Finder (click the Magnifier Glass icon on the toolbar) to search for and open the Disk Utility application. Then select the card in the left menu (see Figure 14-4).

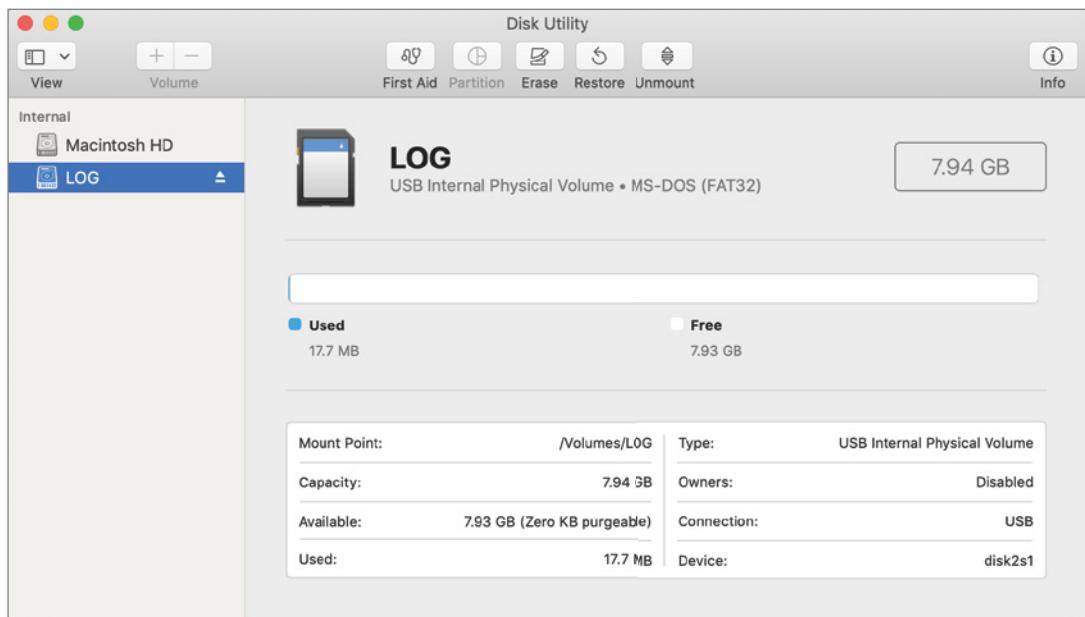


Figure 14-4: Disk Utility application with a card selected

- Click the Erase button at the top of the window. In the window that pops up, name the disk. (I chose LOG, but you can choose whatever you want.) Choose MS-DOS (FAT) for the format (see Figure 14-5).

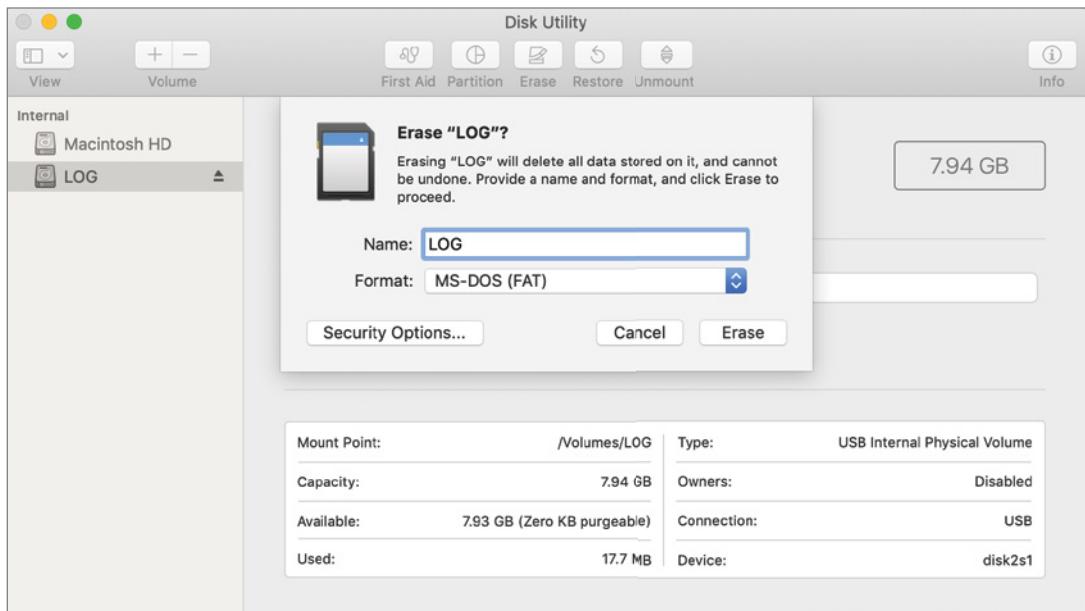


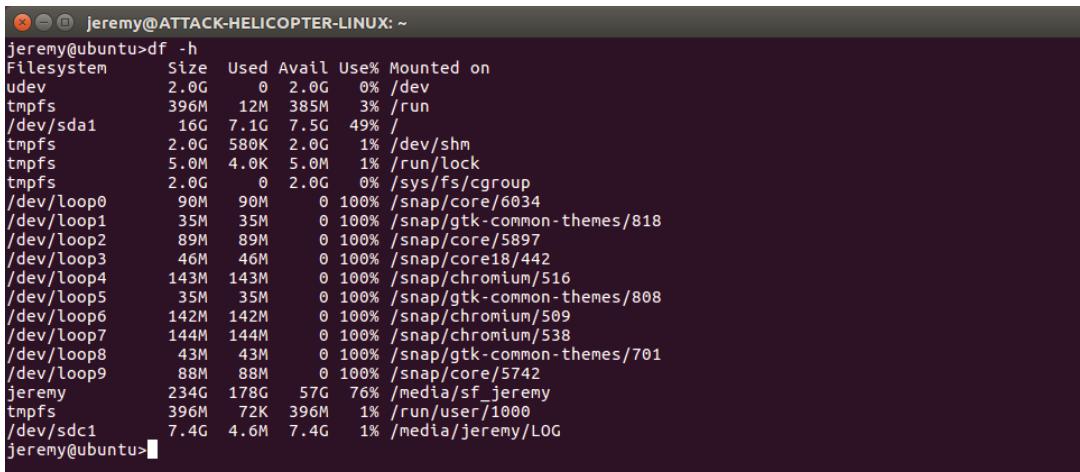
Figure 14-5: Disk Utility Erase options

3. Click Erase. This formats the card as FAT16, regardless of its capacity. (Mac OS computers cannot natively format cards as FAT32.) The card is now formatted and ready for use with your Arduino.

Formatting Your SD Card Using Linux

These instructions assume the use of a Debian-based distribution, such as Ubuntu. Most Linux distributions should mount the card automatically when you insert it.

1. Insert the card. A window pops up showing the card.
2. Open a terminal, and type `df -h` to get a list of the mounted media. The result will look like Figure 14-6.



```
jeremy@ATTACK-HELICOPTER-LINUX: ~
jeremy@ubuntu:~$ df -h
Filesystem      Size   Used  Avail Use% Mounted on
udev            2.0G     0    2.0G  0% /dev
tmpfs           396M   12M   385M  3% /run
/dev/sda1        16G   7.1G   7.5G  49% /
tmpfs           2.0G  580K  2.0G  1% /dev/shm
tmpfs           5.0M   4.0K   5.0M  1% /run/lock
tmpfs           2.0G     0    2.0G  0% /sys/fs/cgroup
/dev/loop0       90M   90M     0 100% /snap/core/6034
/dev/loop1       35M   35M     0 100% /snap/gtk-common-themes/818
/dev/loop2       89M   89M     0 100% /snap/core/5897
/dev/loop3       46M   46M     0 100% /snap/core18/442
/dev/loop4       143M  143M     0 100% /snap/chromium/516
/dev/loop5       35M   35M     0 100% /snap/gtk-common-themes/808
/dev/loop6       142M  142M     0 100% /snap/chromium/509
/dev/loop7       144M  144M     0 100% /snap/chromium/538
/dev/loop8       43M   43M     0 100% /snap/gtk-common-themes/701
/dev/loop9       88M   88M     0 100% /snap/core/5742
jeremy          234G  178G   57G  76% /media/sf_jeremy
tmpfs           396M   72K   396M  1% /run/user/1000
/dev/sdc1        7.4G  4.6M   7.4G  1% /media/jeremy/LOG
jeremy@ubuntu:~$
```

Figure 14-6: Linux `df` command output

The last entry will be your SD card. Confirm this by checking that the size matches what you expect. (You can see that my 8 GB card is reported as “7.4G,” which is correct; the value reported here is always slightly less than what the card is labeled as because of how the SD card manufacturers define the number of bytes in a gigabyte.) On my system, it was mounted as `/dev/sdc1`, but the exact location may be different, depending on your computer and whether or not the card was already previously formatted.

3. Unmount the card before you format it by using the `umount` command. The argument is the name of your SD card (see Figure 14-7). So, for my SD card, the command is `umount /dev/sdc1`.

```

jeremy@ATTACK-HELICOPTER-LINUX: ~
Filesystem      Size   Used  Avail Use% Mounted on
udev            2.0G     0    2.0G  0% /dev
tmpfs           396M   12M   385M  3% /run
/dev/sda1        16G   7.1G   7.5G  49% /
tmpfs           2.0G  248K   2.0G  1% /dev/shm
tmpfs           5.0M   4.0K   5.0M  1% /run/lock
tmpfs           2.0G     0    2.0G  0% /sys/fs/cgroup
/dev/loop0       90M    90M    0 100% /snap/core/6034
/dev/loop1       35M   35M    0 100% /snap/gtk-common-themes/818
/dev/loop2       89M   89M    0 100% /snap/core/5897
/dev/loop3       46M   46M    0 100% /snap/core18/442
/dev/loop4      143M  143M    0 100% /snap/chromium/516
/dev/loop5       35M   35M    0 100% /snap/gtk-common-themes/808
/dev/loop6      142M  142M    0 100% /snap/chromium/509
/dev/loop7      144M  144M    0 100% /snap/chromium/538
/dev/loop8       43M   43M    0 100% /snap/gtk-common-themes/701
/dev/loop9       88M   88M    0 100% /snap/core/5742
jeremy          234G  178G   57G  76% /media/sf_jeremy
tmpfs           396M   72K   396M  1% /run/user/1000
/dev/sdc1        7.4G   4.6M   7.4G  1% /media/jeremy/LOG
jeremy@ubuntu>umount /dev/sdc1
jeremy@ubuntu>

```

Figure 14-7: Unmounting the SD card in Linux

4. Format the card using the **mkdosfs** command. You may need to run the command as a super user (using the **sudo** command). You will pass the **-F** flag, specifying to use a FAT file system. You can include either 16 or 32 as the flag argument to choose FAT16 or FAT32. To format an 8 GB card that was mounted as **/dev/sdc1**, you use the command **sudo mkdosfs -F 32 /dev/sdc1** (see Figure 14-8). Unplug and replug your card after formatting it. It will mount automatically.

```

tmpfs           396M   12M   385M  3% /run
/dev/sda1        16G   7.1G   7.5G  49% /
tmpfs           2.0G  248K   2.0G  1% /dev/shm
tmpfs           5.0M   4.0K   5.0M  1% /run/lock
tmpfs           2.0G     0    2.0G  0% /sys/fs/cgroup
/dev/loop0       90M    90M    0 100% /snap/core/6034
/dev/loop1       35M   35M    0 100% /snap/gtk-common-themes/818
/dev/loop2       89M   89M    0 100% /snap/core/5897
/dev/loop3       46M   46M    0 100% /snap/core18/442
/dev/loop4      143M  143M    0 100% /snap/chromium/516
/dev/loop5       35M   35M    0 100% /snap/gtk-common-themes/808
/dev/loop6      142M  142M    0 100% /snap/chromium/509
/dev/loop7      144M  144M    0 100% /snap/chromium/538
/dev/loop8       43M   43M    0 100% /snap/gtk-common-themes/701
/dev/loop9       88M   88M    0 100% /snap/core/5742
jeremy          234G  178G   57G  76% /media/sf_jeremy
tmpfs           396M   72K   396M  1% /run/user/1000
/dev/sdc1        7.4G   4.6M   7.4G  1% /media/jeremy/LOG
jeremy@ubuntu>umount /dev/sdc1
jeremy@ubuntu>sudo mkdosfs -F 32 /dev/sdc1
[sudo] password for jeremy:
mkfs.fat 3.0.28 (2015-05-16)
jeremy@ubuntu>

```

Figure 14-8: Formatting the SD card in Linux

You're now ready to start interfacing with the SD card via an SD card shield.

Interfacing the Arduino with an SD Card

SD cards are 3.3V devices. Therefore, it's important to connect to SD cards through a shield that properly handles the logic-level shifting and voltage supply to your SD card. Furthermore, SD communication can be accomplished using the serial peripheral interface (SPI) bus, something that you should already be familiar with after having read Chapter 11, "The SPI Bus and Third-Party Libraries." The Arduino language comes with a handy library (the SD library) that abstracts away the lower-level SPI communication and allows you to easily read and write files stored on your SD card. You will use this library throughout this chapter.

SD Card Shields

You have a tremendous number of options for adding data logging capabilities to your Arduino. It is impossible to provide documentation for every shield that is available, so this discussion keeps the examples general enough to apply to most shields with SD card connection capabilities. This section identifies some of the more popular shields and the pros and cons of using each one.

All shields have the following in common:

- They connect to SPI pins via either the 6-pin programming header or multiplexed digital pins. These are pins 11, 12, and 13 on the Uno, and pins 50, 51, and 52 on Mega boards. The Leonardo's SPI pins are located on the in-circuit serial programming (ICSP) header only.
- They designate a chip select (CS) pin, which may or may not be the default CS pin (10 on non-Mega boards, 53 on Mega boards). In practice, this doesn't matter much, as you can designate any digital pin to serve as the CS pin.
- They supply 3.3V to the SD card and will level-shift the logic levels if necessary.

Not all shields work with all Arduinos. For example, a shield that only uses the multiplexed digital SPI pins (without a header to connect to the ICSP pins) will not work with the Leonardo, as the Leonardo only has SPI available on the ICSP header. A shield that assumes a 5V input logic may not work with a 3.3V Arduino, such as the

Due. Always check the documentation from the manufacturer to ensure that a shield will work with your Arduino of choice.

Here's a list of the most common SD card shields:

- **Adafruit data logging shield** (exploringarduino.com/part/adafruit-data-logging-shield): This shield is particularly well suited to the experiments that you will be doing later in this chapter because it includes both a real-time clock (RTC) chip and an SD card interface. This shield connects the SD card to the default CS pin and connects a real-time clock chip to the I²C bus (see Figure 14-9). This is the shield that will be used for the remainder of this chapter.

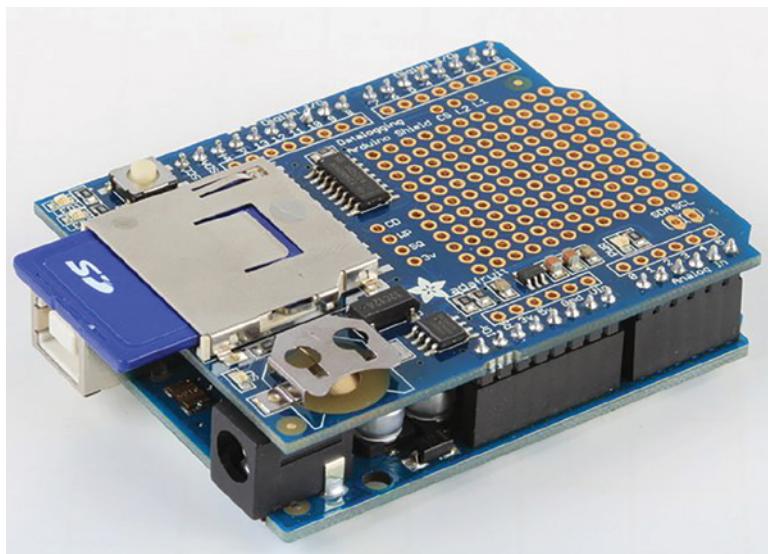


Figure 14-9: Adafruit data logging shield, mounted on an Uno

Credit: Adafruit, adafruit.com

- **SparkFun Micro SD shield** (exploringarduino.com/part/sparkfun-microSD-shield): This is a minimalist shield that only has a Micro SD card slot.

However, it also has a large prototyping area that allows you to solder on additional components. It connects the SD card's CS pin to pin 8 on the Arduino, so you must specify this when using the SD card library with this shield (see Figure 14-10).

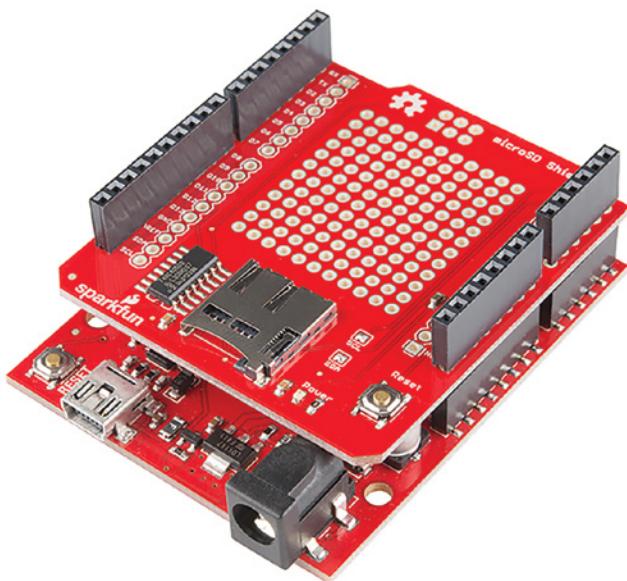


Figure 14-10: SparkFun Micro SD shield, mounted on a SparkFun RedBoard

Credit: SparkFun Electronics Inc., sparkfun.com

- **Seeed Studio SD Card Shield V4** (exploringarduino.com/parts/seeed-sd-card-shield): The Seeed Studio shield is another minimalist shield that supports a full-sized SD card. It also sports I²C breakout connectors and UART connectors that can be used for interfacing to Seeed's "Grove" line of Arduino peripherals. It connects the CS line to digital pin 4 of the Uno. Figure 14-11 shows this shield.



Figure 14-11: Seeed Studio SD card shield V4

Credit: Seeed Technology Co.,Ltd., seeed.cc

SD Card SPI Interface

As mentioned earlier, your Arduino communicates with the SD card over an SPI interface. This necessitates the use of a MOSI (master output, slave input), MISO (master input, slave output), SCLK (serial clock), and CS (chip select) pin. You will use the SD card Arduino library to complete the following examples. These examples assume that you are using the default SPI pins on your Arduino (as opposed to software-emulated SPI on other digital pins) and either a default or custom CS pin. The SD card library must have the default CS pin set as an output to function correctly, even if you are using a different CS pin. In the case of the Uno, this is pin 10; in the case of the Mega, this is pin 53. The following examples use the Uno with the default CS pin 10.

Writing to an SD Card

First, you will use the SD card library to write some sample data to your SD card. Later in the chapter, you will capture some sensor data and write that data directly to

the SD card. The data is stored in a file called `log.csv` that you can later open on your computer. Importantly, if you formatted your card as FAT16, the filenames you use must be in 8.3 format. This means that the extension must be three characters, and the filename must be eight or fewer characters.

Ensure that your SD shield is mounted correctly to your Arduino and that you have an SD card inserted. When mounted, the Adafruit data logging shield should look like Figure 14-9.

ASSEMBLING THE DATA LOGGING SHIELD

If your shield requires assembly, follow the instructions on the Adafruit website to solder the pins into place: blum.fyi/soldering-shield-headers. The shield ships with normal pins for soldering into the board, but you can alternatively get *shield stacking headers* that will make it easier to plug other components in (particularly for the project at the end of this chapter). If you don't have stacking headers, just be aware that you will also need to solder in additional header pins in order to plug additional components into your Arduino plus shield.

For the sake of debugging, you will take advantage of the reporting capability of many of the SD card functions. For example, to initialize communication with an SD card, you call the following function in your setup:

```
if (!SD.begin(CS_pin))
{
    Serial.println("Card Failure");
    return;
}
Serial.println("Card Ready");
```

Note that instead of just calling `SD.begin(CS_pin)`, it executes within an `if` statement. This tries to initialize the SD card, and it returns a status. If it returns true, the program moves on, and a success message is printed to the serial terminal. If it returns false, a failure message is reported, and the `return` command halts further execution of the program.

You use a similar approach when you are ready to write a new line of data to a log file. If you wanted to write “hello” to a new line in the file, the code would look like this:

```
File dataFile = SD.open("log.csv", FILE_WRITE);
if (dataFile)
{
```

```
    dataFile.println("hello");
    dataFile.close(); //Data isn't written until we close the connection!
}
else
{
    Serial.println("Couldn't open log file");
}
```

This first line creates a new file (or opens the file if it exists) called `log.csv` on the SD card. If the file is opened successfully, the `dataFile` variable is `true`, and the write process is initiated. If it is `false`, an error is reported to the serial monitor. Writing new lines to a file is easy: Just execute `dataFile.println()` and pass what you want to write to a new line. You can also use `print()` to avoid appending a newline character to the end. This is sent to a buffer, and is only actually added to the file once the `close` command is called on the same File.

Now, you can bring all this knowledge together into a simple program that will create a `log.csv` file on your SD card and write a comma-separated timestamp and phrase every 5 seconds. On each line of the CSV file, you record the current time from `millis()` and a simple phrase. This might not seem very useful, but it is an important step to test before you start adding actual measurements in the coming examples. The code should look like Listing 14-1.

Listing 14-1

SD card write test-write_to_sd.ino

```
//Write to SD card

#include <SD.h> //Include the SD Card Library

//These are set by default via the SD card library
//MOSI = Pin 11
//MISO = Pin 12
//SCLK = PIN 13

//We always need to set the CS Pin
const int CS_PIN = 10;

void setup()
{
    Serial.begin(9600);
    Serial.println("Initializing Card");
```

```
//CS pin must be configured as an output
pinMode(CS_PIN, OUTPUT);

if (!SD.begin(CS_PIN))
{
    Serial.println("Card Failure");
    while(1);
}
Serial.println("Card Ready");
}

void loop()
{
    long timeStamp = millis();
    String dataString = "Hello There!";

    //Open a file and write to it.
    File dataFile = SD.open("log.csv", FILE_WRITE);
    if (dataFile)
    {
        dataFile.print(timeStamp);
        dataFile.print(",");
        dataFile.println(dataString);
        dataFile.close(); //Data isn't written until we run close()!

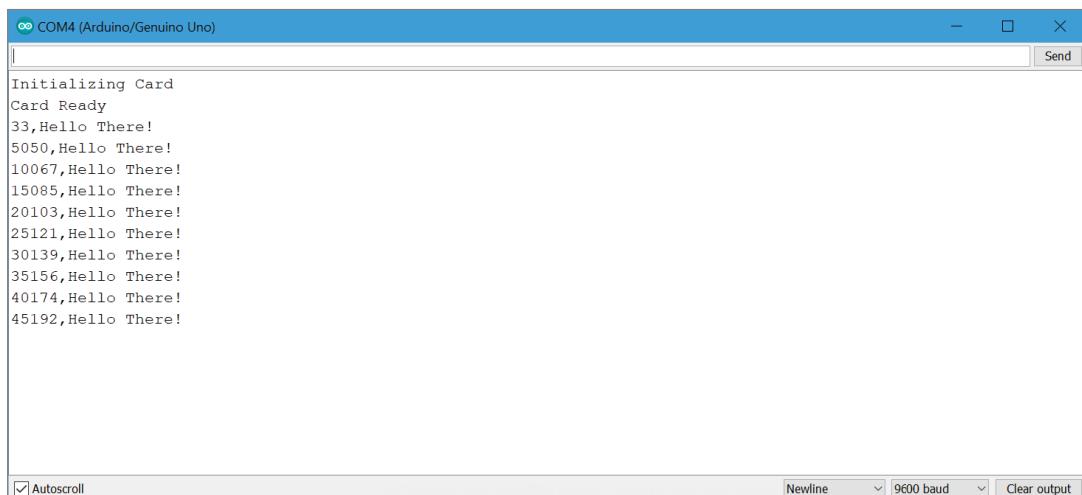
        //Print same thing to the screen for debugging
        Serial.print(timeStamp);
        Serial.print(",");
        Serial.println(dataString);
    }
    else
    {
        Serial.println("Couldn't open log file");
    }
    delay(5000);
}
```

You want to note a few important things in this code sample:

- CS_PIN should be set to whatever pin you have your SD card CS hooked up to. If it is not 10, you must also add `pinMode(10, OUTPUT)` within `setup()`; otherwise, the SD library will not work.

- Each time through the loop, the `timestamp` variable is updated with the current time elapsed in milliseconds. It must be of type `long` because it will generate a number larger than 16 bits (the standard size of an Arduino integer type).

As you saw earlier, the filename is opened for writing and data is appended in a comma-separated format. The same data is also printed out to the serial terminal for debugging purposes. This is not explicitly necessary, and you will not use it once you have the logger “in the field” taking data. However, it is useful for confirming that everything is working. If you open the serial terminal, you should see something like Figure 14-12.



The screenshot shows the Arduino Serial Monitor window titled "COM4 (Arduino/Genuino Uno)". The window displays the following text:

```
Initializing Card
Card Ready
33>Hello There!
5050>Hello There!
10067>Hello There!
15085>Hello There!
20103>Hello There!
25121>Hello There!
30139>Hello There!
35156>Hello There!
40174>Hello There!
45192>Hello There!
```

At the bottom of the window, there are three buttons: "Autoscroll" (checked), "Newline" (dropdown menu), "9600 baud" (dropdown menu), and "Clear output".

Figure 14-12: SD card debugging output

If you receive errors, make sure that your shield is plugged in, that the SD card is inserted fully, and that the card has been properly formatted. You can confirm that the data is being written correctly by removing the SD card, inserting it into your computer, and opening it up with a spreadsheet program (see Figure 14-13). Note how the comma-separated data is automatically placed into rows and columns based on the location of the delimiting commas and newlines.

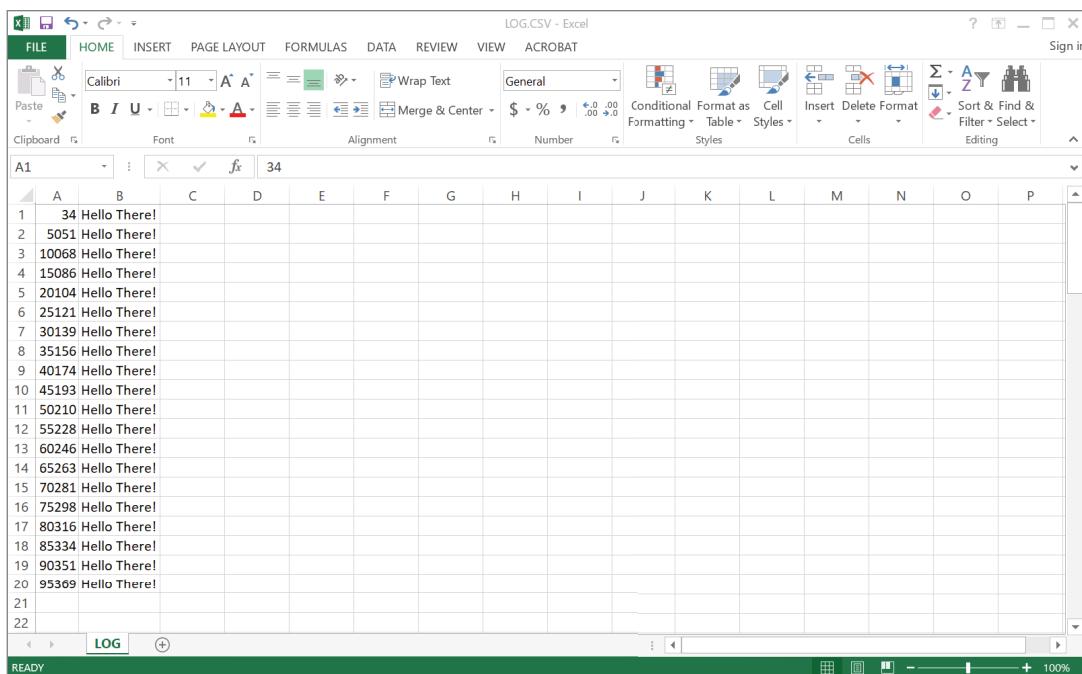


Figure 14-13: Logged data in a spreadsheet

Reading from an SD Card

Now it's time to learn about reading from SD cards. This is not used as commonly for data logging, but it can prove useful for setting program parameters. For instance, you could specify how frequently you want data to be logged. That's what you will do next.

Insert the SD card into your computer and create a new TXT file called speed.txt on the SD card. In this file, simply enter the refresh time in milliseconds that you want to use. In Figure 14-14, you can see that I set it to 1000 ms, or 1 second.

After choosing a desired refresh speed, save the file on the SD card and put it back in your Arduino shield. You can now modify your program to read this file, extract the desired field, and use it to set the refresh speed for data logging.

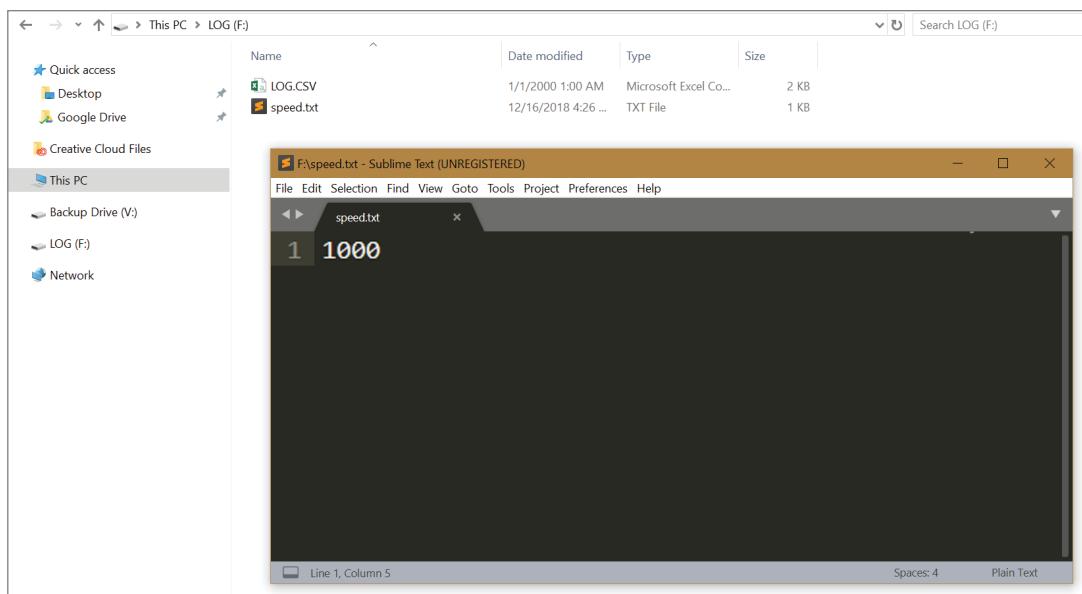


Figure 14-14: Creating the speed command file

To open a file for reading, you use the same `SD.open()` command that you used earlier, but you do not have to specify the `FILE_WRITE` parameter. Because the `File` class that you are using inherits from the `Stream` class (just like the `Serial` class), you can use many of the same useful commands, such as `parseInt()`, that you used in Chapter 7, “USB Serial Communication,” to open and read the update speed from the file. All you have to do is write the following code:

```
File commandFile = SD.open("speed.txt");
if (commandFile)
{
    Serial.println("Reading Command File");

    while(commandFile.available())
    {
        refresh_rate = commandFile.parseInt();
    }
    Serial.print("Refresh Rate = ");
```

```
    Serial.print(refresh_rate);
    Serial.println("ms");
}
else
{
    Serial.println("Could not read command file.");
    return;
}
```

This code segment opens the file for reading and parses out any integers that are read. Because you defined only one variable, it grabs that one and saves it to the refresh rate variable, which would need to be defined earlier in the program. You can have only one file open at a time, and it's good practice to close a file when you're finished reading from it or writing to it.

You can now integrate this code into your writing program from earlier to adjust the recording speed based on your `speed.txt` file, as shown in Listing 14-2.

Listing 14-2

SD reading and writing—`sd_read_write.ino`

```
//SD read and write

#include <SD.h> //Include the SD Card Library

//These are set by default via the SD card library
//MOSI = Pin 11
//MISO = Pin 12
//SCLK = PIN 13

//We always need to set the CS Pin
const int CS_PIN = 10;

//Default rate of 5 seconds
int refresh_rate = 5000;

void setup()
{
    Serial.begin(9600);
    Serial.println("Initializing Card");

    //CS pin must be configured as an output
    pinMode(CS_PIN, OUTPUT);
```

```
if (!SD.begin(CS_PIN))
{
    Serial.println("Card Failure");
    while(1);
}
Serial.println("Card Ready");

//Read the configuration information (speed.txt)
File commandFile = SD.open("speed.txt");
if (commandFile)
{
    Serial.println("Reading Command File");

    while(commandFile.available())
    {
        refresh_rate = commandFile.parseInt();
    }
    Serial.print("Refresh Rate = ");
    Serial.print(refresh_rate);
    Serial.println("ms");
    commandFile.close(); //Close the file when finished
}
else
{
    Serial.println("Could not read command file.");
    Serial.print("Will use default refresh rate of ");
    Serial.print(refresh_rate);
    Serial.println("ms!");
}
}

void loop()
{
    long timeStamp = millis();
    String dataString = "Hello There!";

    //Open a file and write to it.
    File dataFile = SD.open("log.csv", FILE_WRITE);
    if (dataFile)
    {
        dataFile.print(timeStamp);
        dataFile.print(",");
        dataFile.println(dataString);
        dataFile.close(); //Data isn't written until we run close()!
    }
}
```

```
//Print same thing to the screen for debugging
Serial.print(timeStamp);
Serial.print(",");
Serial.println(dataString);
}
else
{
    Serial.println("Couldn't open log file");
}
delay(refresh_rate);
}
```

When you now run this program, data is written at the rate you specify. Looking at the serial terminal confirms this fact (see Figure 14-15).

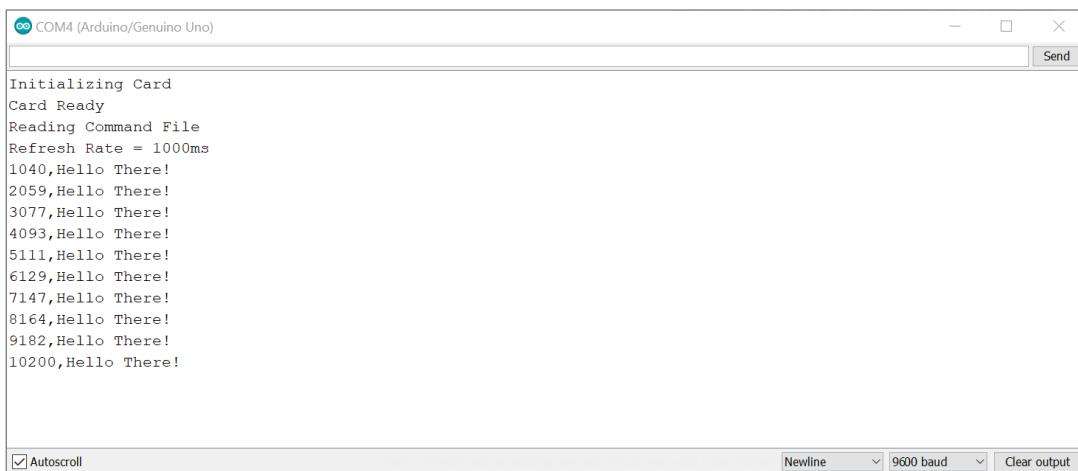


Figure 14-15: Data logging at the rate specified by the command file

If no speed.txt file is present, this sketch will fall back to the default refresh rate of 5000 ms. Also note that this sketch does *not* delete the log.csv file each time before it is run, so new data will be appended to the end of your existing log file each time you reset or power-cycle the Arduino.

Real-Time Clocks

Nearly every data logging application will benefit from the use of a real-time clock. Using a real-time clock within your system allows you to time-stamp measurements so that you can more easily keep track of when certain events occurred. In the previous section, you simply used the `millis()` function to keep track of the time elapsed since the Arduino turned on. In this section, you use a dedicated real-time clock integrated circuit to keep accurate time so that when you save data to the SD card, it corresponds to the time the data was taken.

Understanding Real-Time Clocks

Real-time clocks, or RTCs, do exactly what their name implies. You set the time once, and they keep very accurate time, even accounting for anomalies such as leap years. This example uses the PCF8523 RTC from NXP Semiconductors. It is included on the Adafruit data logging shield. If you happen to have an older version of the Adafruit data logging shield, it may use the DS1307 RTC from Maxim Integrated instead. Either RTC will work for this application, but you'll just need to adjust one line in the code to specify the one you are using. Shields that use the PCF8523 RTC have "PCF8523" written on the silkscreen next to the coin cell battery. If your shield doesn't have PCF8523 written on the silkscreen, then it is using the DS1307 RTC.

Communicating with a Real-Time Clock

The real-time clock communicates with your Arduino over an I²C connection and connects to a coin cell battery that will allow it to keep time for several years. A crystal oscillator connected to the real-time clock enables precision timekeeping.

If you are using a very old Arduino that doesn't have dedicated I²C pins (the ones next to AREF) or an IO Reference pin (the one next to RST), then there are three jumpers you need to solder closed in order for this shield to work. Solder closed the two jumpers for the I²C interface on the rear of the Adafruit shield (see Figure 14-16), and solder the "IOr" middle pad to the 5V pad near the RTC chip (see Figure 14-17). Newer Arduino Unos (R3 and newer) and Adafruit METRO 328 boards do not need these pads to be soldered. If you haven't already, install the battery into the clip on the shield—the RTC will not work properly if one is not installed. The positive side of the battery faces upwards.



Figure 14-16: Solder these jumpers on old Arduinos without dedicated I²C pins

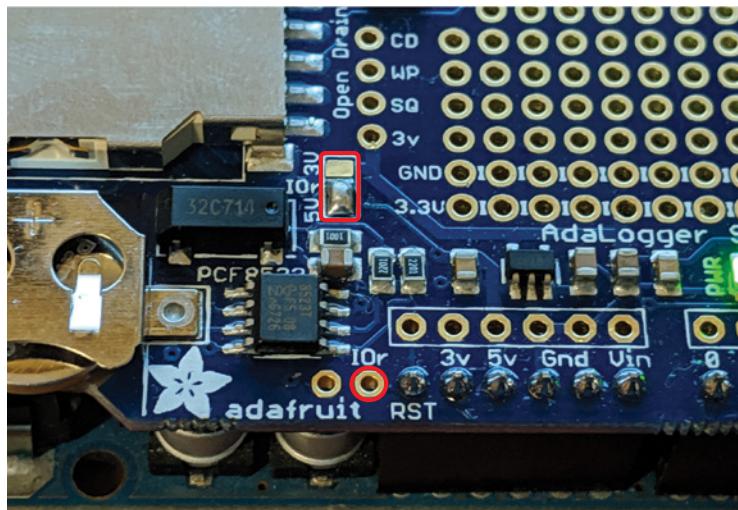


Figure 14-17: Solder this jumper on old Arduinos without a dedicated IO reference pin

Using the RTC Arduino Third-Party Library

As in preceding chapters, you will use a third-party library to extend the Arduino's capabilities. In this case, it's to facilitate easy communication with the real-time clock (RTC) chip. Unsurprisingly, the library is called *RTClib*. The library was originally developed by JeeLabs, and was updated by Adafruit Industries. To install the library, go to Sketch > Include Library > Manage Libraries. Search for *rtclib*, locate the "RTClib by Adafruit" entry, and install it as shown in Figure 14-18.

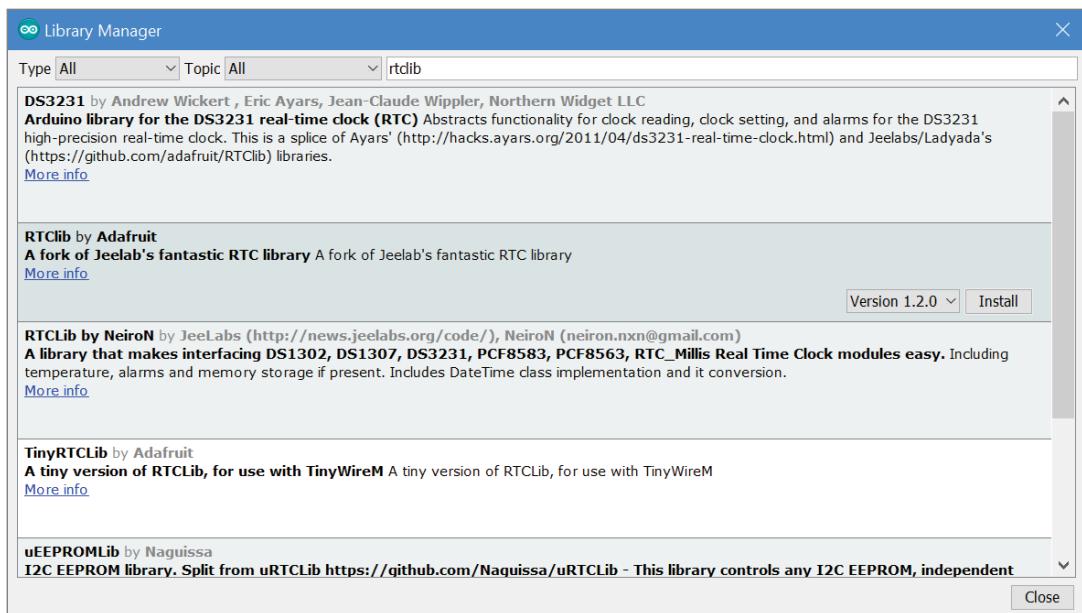


Figure 14-18: Finding the RTC library

The library is easy to use. The first time you run the example code, you use the RTC.`adjust()` function to automatically grab the current date and time from your computer at the time of compilation and use that to set up the clock. From this point on, the RTC runs autonomously, and you can obtain the current date and time from it by executing the RTC.`now()` command. In the next section, you will use this functionality to enable real-time logging.

Using a Real-Time Clock

Now it is time to combine the SD card and real-time clock, along with the RTC library that you just downloaded, to enable logging using actual timestamps. You will update your sketch once again to use the RTC values rather than the `millis` values.

Installing the RTC and SD Card Modules

First, ensure that the data logging shield is connected to your Arduino, the SD card is installed, and the battery is inserted.

Updating the Software

Now, you add the RTC functionality into the software. You need to add a few things to your previous program to integrate the RTC:

- Include the RTC libraries.
- Initialize the correct RTC object based on which chip you have. (You'll use a new concept called a "preprocessor directive" to accomplish this task.)
- Set the RTC time using the computer time (using a compiler macro).
- Write the actual timestamps to the log file.

Furthermore, in this code revision, I added a column header that is printed every time the code starts. This way, even if you are appending to an existing CSV file, you can easily find each time the log was restarted.

WARNING If, when you run your program, you notice that it simply stops after a short while, you may be running out of RAM. In most cases, this can be attributed to strings that take up a large amount of RAM, especially within your `Serial.print` and `Serial.println` statements. You can resolve this problem by removing serial printing statements, or by telling the Arduino to store these strings in flash memory instead of in RAM. You can store strings in flash memory by wrapping the serial print string in `F()`, like this: `Serial.println(F("Hello"));`. This method is used in Listing 14-3.

The updated program is shown in Listing 14-3, using the RTC as a clock for data logging. It moves the majority of the strings into flash memory to save RAM using the technique explained in the previous warning.

Listing 14-3

SD reading and writing with an RTC—`sd_read_write_rtc.ino`

```
//SD read and write with RTC

//Uncomment following line if your board uses the DS1307 instead of the PCF8523
//#define RTC_CHIP_IS_DS1307

//Uncomment following line if you want to force the time to be set
//Should always be commented out before "running in the field"
//#define FORCE_UPDATE

#include <SD.h> //Include the SD Card Library
```

```
#include <Wire.h> //For RTC I2C
#include "RTClib.h" //For RTC

//SD Card is on standard SPI Pins
//RTC is on standard I2C Pins

//We always need to set the CS Pin for the SD Card
const int CS_PIN = 10;

//Default rate of 5 seconds can be overwritten by speed.txt file
int refresh_rate = 5000;

// Use compiler flags to set up the right chip type
#ifndef RTC_CHIP_IS_DS1307
    RTC_DS1307 RTC;
    String chip = "DS1307";
#else
    RTC_PCF8523 RTC;
    String chip = "PCF8532";
#endif

// Use compiler flags to decide if an update should be forced
#ifndef FORCE_UPDATE
    bool update_clock = true;
#else
    bool update_clock = false;
#endif

//Initialize date and time strings
String time, date;

void updateDateTime()
{
    //Get the current date and time info and store in strings
    DateTime datetime = RTC.now();
    String year = String(datetime.year(), DEC);
    String month = String(datetime.month(), DEC);
    String day = String(datetime.day(), DEC);
    String hour = String(datetime.hour(), DEC);
    String minute = String(datetime.minute(), DEC);
    String second = String(datetime.second(), DEC);

    //Concatenate the strings into date and time
    date = year + "/" + month + "/" + day;
    time = hour + ":" + minute + ":" + second;
}
```

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

    //CS pin must be configured as an output
    pinMode(CS_PIN, OUTPUT);

    //Initiate the RTC library
    RTC.begin();

    //Always update the time if the RTC isn't running
    #ifdef RTC_CHIP_IS_DS1307
        if (!RTC.isrunning()) update_clock = true;
    #else
        if (!RTC.initialized()) update_clock = true;
    #endif

    //If RTC not running or if we force it, set RTC to computer's compile time
    if (update_clock)
    {
        Serial.print(F("Setting "));
        Serial.print(chip);
        Serial.print(F(" time to compile time..."));
        RTC.adjust(DateTime(F(__DATE__), F(__TIME__)));
        Serial.println(F("Done!"));
    }
    else
    {
        Serial.print(chip);
        Serial.println(F(" time is already set!"));
    }

    //Show the time
    updateDateTime();
    Serial.print(F("RTC Date: "));
    Serial.println(date);
    Serial.print(F("RTC time: "));
    Serial.println(time);

    //Initialize SD card
    Serial.print(F("Initializing SD Card..."));
    if (!SD.begin(CS_PIN))
    {
        Serial.println(F("Card Failure!"));
        while(1);
    }
    Serial.println(F("Card Ready!"));
}
```

```
//Read the configuration information (speed.txt)
File commandFile = SD.open("speed.txt");
if (commandFile)
{
    Serial.print(F("Reading Command File..."));

    while(commandFile.available())
    {
        refresh_rate = commandFile.parseInt();
    }
    Serial.print(F("Refresh Rate = "));
    Serial.print(refresh_rate);
    Serial.println(F("ms"));
    commandFile.close(); //Close the file when finished
}
else
{
    Serial.println(F("Could not read command file."));
    Serial.print(F("Will use default refresh rate of "));
    Serial.print(refresh_rate);
    Serial.println(F("ms!"));
}

//Write column headers
File dataFile = SD.open("log.csv", FILE_WRITE);
if (dataFile)
{
    dataFile.println(F("\nNew Log Started!"));
    dataFile.println(F("Date,Time,Phrase"));
    dataFile.close(); //Data isn't written until we run close()!

    //Print same thing to the screen for debugging
    Serial.println(F("\nNew Log Started!"));
    Serial.println(F("Date,Time,Phrase"));
}
else
{
    Serial.println(F("Couldn't open log file"));
    while(1);
}

void loop()
{
    updateDateTime(); //Get the current date/time
    String dataString = "Hello There!";
}
```

```

//Open a file and write to it.
File dataFile = SD.open("log.csv", FILE_WRITE);
if (dataFile)
{
    dataFile.print(date);
    dataFile.print(F(","));
    dataFile.print(time);
    dataFile.print(F(","));
    dataFile.println(dataString);
    dataFile.close(); //Data isn't written until we run close()!

    //Print same thing to the screen for debugging
    Serial.print(date);
    Serial.print(F(","));
    Serial.print(time);
    Serial.print(F(","));
    Serial.println(dataString);
}
else
{
    Serial.println(F("Couldn't open log file!"));
}
delay(refresh_rate);
}

```

The RTC library is imported by the sketch via `#include "RTClib.h"`. The RTC is an I²C device, and relies on the Wire library, so that library needs to be included, too. At the top of the file, you'll notice two `#define` statements that are commented out by default:

```

//#define RTC_CHIP_IS_DS1307
//#define FORCE_UPDATE

```

These statements are different from the constants that you are used to. When uncommented, they act as special instructions to the preprocessor that compiles your code from readable code into machine code (ones and zeros) that the Arduino's microcontroller can understand. A few lines down, you'll see where these special flags are first used:

```

// Use compiler flags to setup the right chip type
#ifndef RTC_CHIP_IS_DS1307
    RTC_DS1307 RTC;
    String chip = "DS1307";
#else
    RTC_PCF8523 RTC;
    String chip = "PCF8532";
#endif

```

All lines that start with a `#` are treated as special instructions during the compilation of your code. `#ifdef` does what its name suggests: it includes the code within its block in final compilation of the software if the associated preprocessor definition exists. So, if `#define RTC_CHIP_IS_DS1307` is commented out, then that definition does *not* exist, and so the `#else` statement is compiled into the final software (meaning the RTC class is set up for the PCF8523 chip instead of the DS1307 chip). Through the use of these preprocessor commands, you can write one piece of software that is capable of being compiled into multiple variants, depending on how you adjust the `#define` elements at the top of the sketch. This also saves valuable memory space on the Arduino by only including the bits you actually need in the final machine code.

So, why is this preprocessor variable definition approach used here? The *RTClib* library defines standard functions that can be used for multiple types of RTC chips, but the correct one needs to be initialized so that the Arduino knows its I²C address, and in what registers it stores its data. By defining the type of chip in this way, you can initialize the RTC object with the right class type: either `RTC_DS1307 RTC;` or `RTC_PCF8523 RTC;`. This makes this sketch universal, regardless of which version of the data logging shield you have. When the RTC object is initialized, you also create a string variable that includes the name of the chip for later use in debugging print statements. The `FORCE_UPDATE` preprocessor definition can also be uncommented to set a state variable that will tell the RTC to set its internal clock to that of the programming computer. Regardless of the state of `FORCE_UPDATE`, the state of the RTC chip is always checked in the `setup()` function, and it is updated to the system time if it is currently not initialized.

If an update to the RTC internal time was requested (either by force or because the chip was not initialized), then `RTC.adjust(DateTime(F(__DATE__), F(__TIME__)));` sets the RTC's current internal date and time to the date and time reported by the programming computer. Once this is set, the time will not be reset as long as the battery stays connected to the RTC (the battery will last for several years). The `__DATE__` and `__TIME__` are special macros that the preprocessor replaces with the computer's date and time during compilation of the sketch into machine code.

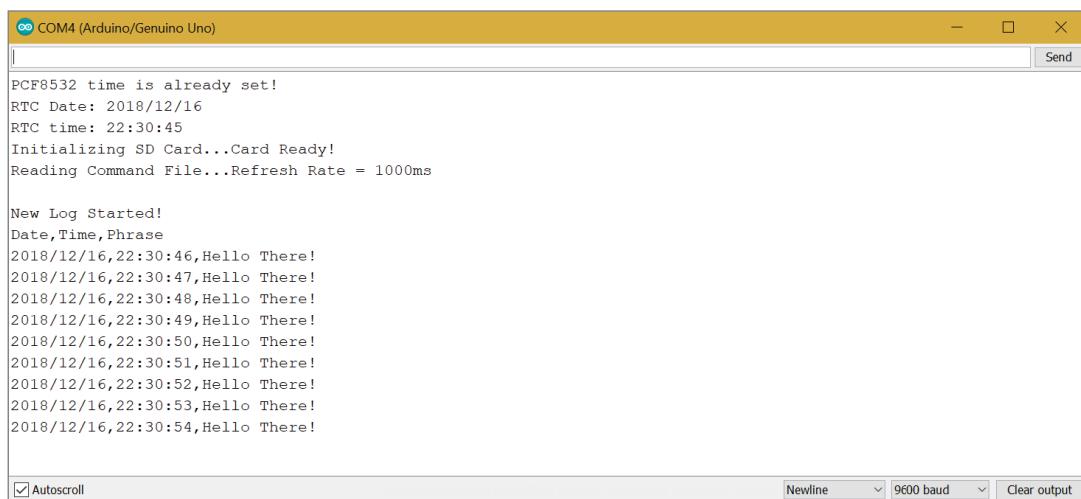
Also in `setup()`, a column header is inserted into the log file, adding a note that the logging has been restarted. This is useful for appending to the log file each time you restart the system.

During each pass through the loop, the `updateDateTime()` function is called. It calls `RTC.now()`, which returns a `DateTime` object from which you can extract the year, month, hour, and so on, for conversion into strings that you can concatenate into the date and time variables. These variables are printed to the serial console and to the SD card log file.

To compile and run this sketch, you must first make sure the `#define` statements are set correctly for your configuration. If you are using a modern data logging shield,

then you should be able to leave them both commented out. Click the Upload button (which always compiles, and then uploads) to send the software over to your attached Arduino. If you compile and then upload later, the date and time will be wrong because they are determined at compilation time, not at programming time.

Open the serial monitor to ensure that it's showing a correct date and time, and to ensure that the sketch is able to talk to the SD card (see Figure 14-19 for an example). If the date or time is wrong, check the following:



The screenshot shows the Arduino Serial Monitor window titled "COM4 (Arduino/Genuino Uno)". The text area displays the following log:

```

PCF8532 time is already set!
RTC Date: 2018/12/16
RTC time: 22:30:45
Initializing SD Card...Card Ready!
Reading Command File...Refresh Rate = 1000ms

New Log Started!
Date,Time,Phrase
2018/12/16,22:30:46>Hello There!
2018/12/16,22:30:47>Hello There!
2018/12/16,22:30:48>Hello There!
2018/12/16,22:30:49>Hello There!
2018/12/16,22:30:50>Hello There!
2018/12/16,22:30:51>Hello There!
2018/12/16,22:30:52>Hello There!
2018/12/16,22:30:53>Hello There!
2018/12/16,22:30:54>Hello There!

```

At the bottom of the window, there are buttons for "Autoscroll", "Newline", "9600 baud", and "Clear output".

Figure 14-19: Example serial output from RTC SD card test

- Did you install the battery?
- Did you solder the I²C jumpers if using an old Arduino without dedicated SDA/SCL pins?
- Did you solder the IO_R jumper if using an old Arduino without a dedicated IO_R pin?
- Did you check your version of the data logging shield and uncomment the #define RTC_CHIP_IS_DS1307 line if your shield is not using the PCF8523 RTC chip?

If you continue to have issues after checking those things, try to force an update to the RTC time by uncommenting the #define FORCE_UPDATE line and uploading the code. If that works, then comment that line out again and re-upload.

You may want to eject the SD card from the Arduino, plug it into your computer, delete the LOG.csv file, and then insert it back into your Arduino. This way, all your newly formatted data won't just be appended to the end of the tests you did earlier.

After running this sketch on your Arduino for a little while, use your computer to read the SD card and to open the log file; it should be populated with the date and time

and look similar to Figure 14-20. Your spreadsheet software may automatically change the dates into your local formatting.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	New Log Started!																			
2	Date	Time	Phrase																	
3	12/16/2018	22:33:44	Hello There!																	
4	12/16/2018	22:33:45	Hello There!																	
5	12/16/2018	22:33:46	Hello There!																	
6	12/16/2018	22:33:47	Hello There!																	
7	12/16/2018	22:33:48	Hello There!																	
8	12/16/2018	22:33:49	Hello There!																	
9	12/16/2018	22:33:50	Hello There!																	
10	12/16/2018	22:33:51	Hello There!																	
11	12/16/2018	22:33:52	Hello There!																	
12	12/16/2018	22:33:53	Hello There!																	
13	12/16/2018	22:33:54	Hello There!																	
14	12/16/2018	22:33:55	Hello There!																	
15	12/16/2018	22:33:56	Hello There!																	
16	12/16/2018	22:33:57	Hello There!																	
17	12/16/2018	22:33:58	Hello There!																	
18	12/16/2018	22:33:59	Hello There!																	
19	12/16/2018	22:34:00	Hello There!																	
20	12/16/2018	22:34:01	Hello There!																	
21	12/16/2018	22:34:02	Hello There!																	
22	12/16/2018	22:34:03	Hello There!																	
23	12/16/2018	22:34:04	Hello There!																	
24	12/16/2018	22:34:05	Hello There!																	
25	12/16/2018	22:34:06	Hello There!																	
26																				
27																				
28																				
29																				
30																				

Figure 14-20: Spreadsheet output from RTC SD card test

TURNING YOUR CHRONOGRAPH INTO A CLOCK

In Chapter 5, “Driving Stepper and Servo Motors,” you learned to how to use a stepper motor and the Arduino’s timing functions to make an accurate chronograph. Now that you know how to get the real time using an RTC, can you update that chronograph project to represent real time? Mark hours (12 or 24 hours) on the “clock face” (the CD) and make the stepper motor into an hour hand that represents the current hour of the day. Alternatively, you can get creative, and turn it into solar clock that tracks sunrise and sunset times. Compute the approximate sunrise and sunset times for your latitude based on the date in the RTC, and draw sunrise and sunset icons on your clock face instead of hours. The clock will need to change its speed based on the time of year since there are fewer hours of sunlight in winter than in summer.

Building an Entrance Logger

Now that you have all the basic skills down, you can put them to use to build an entrance logger for your room. You can use the distance sensor from some of your

previous projects to create a basic motion sensor that can keep track of when people enter or exit through a doorway. The logger will keep track of the times of these events on the SD card for you to review later.

Logger Hardware

All you need to do is to add an analog distance sensor to your existing setup. If you're using the Adafruit data logging shield with the *shield stacking* headers installed, then you can plug your distance sensor directly into the 5V (red wire), GND (black or brown wire), and A0 (white or yellow wire) pins that are brought through the shield via those headers. If you assembled the data logging shield with the headers that shipped with it, then you need to solder in additional headers that you can plug the sensor into. Figure 14-21 shows what this looks like. You can also strip the wires from your sensor and solder them directly into the right through-holes.

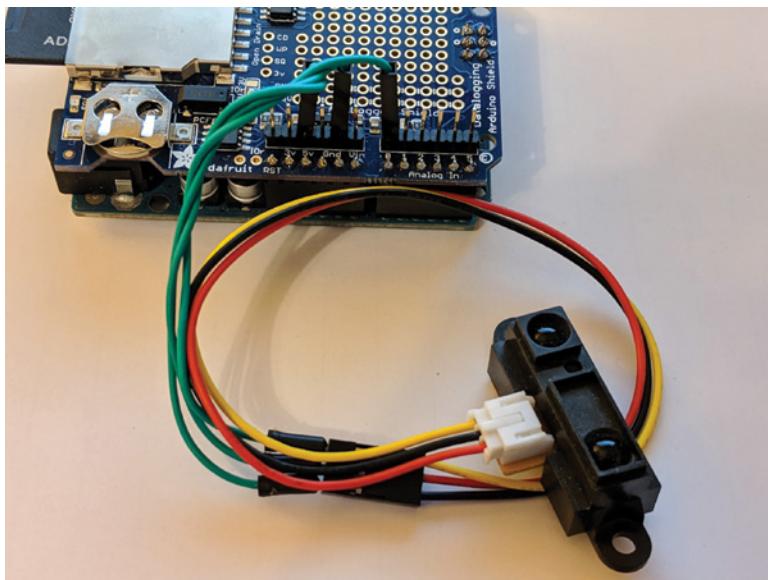


Figure 14-21: Assembled entrance logger hardware

For this to actually work well, you want to mount the IR distance sensor and Arduino on a wall so that the IR beam cuts horizontally across the doorway or hallway. This way, anybody walking through the door must pass in front of the IR distance sensor. Don't affix anything to your wall until you've written the software in the next step and uploaded it. I suggest using easily removable painter's tape or 3M® Command™ Strips to hold it to your wall so that you don't damage anything. Once set up, the system will

look like Figure 14-22. If you’re looking to detect your dog, cat, or child, then you’ll obviously want to position the sensor much lower to the ground!

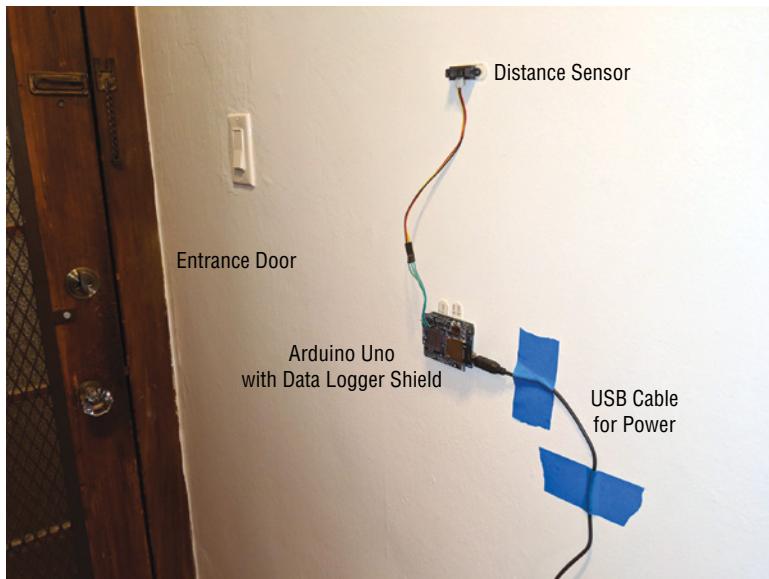


Figure 14-22: Entrance logger trained on an entranceway

You have a few options for power. You can use the USB cable with a USB wall brick (like the one that probably shipped with your smartphone), you can use a 9–12V DC wall brick that plugs into the barrel jack, or you can use a 9V battery clip plugged into the barrel jack. Of course, that battery will eventually run out of power. In Figure 14-22, you can see that I’m using a USB cable and a wall power brick (not pictured).

Logger Software

For the entrance logger, reading configuration variables from the SD card is not particularly useful, so you can remove those parts of the code. You want to add some code to check the state of the distance sensor and to see whether its readings have changed drastically between successive pollings. If they have, you can assume that something moved in front of the distance sensor and that somebody must have entered or exited the room.

You also need to choose a “change threshold.” For my setup, I found that an analog reading change of more than 75 between pollings was a good indication of movement. (Your setup will probably be different. It’s a good idea to check the values of your system

once you have the physical setup fixed.) You want to make sure you're checking the distance sensor frequently enough that you capture movement every time. However, it doesn't make sense to run it so often that you end up with millions of readings for a day's worth of logging.

I recommend that you write to the SD card every time movement is detected, but that you only periodically write to the SD card when there is no movement. This methodology strikes a good balance between storage space required and accuracy. Because you care the most about having accurate timing for when somebody passes the sensor, that detection is recorded with a higher temporal resolution than when nothing is happening in front of the sensor. This technique is implemented in Listing 14-4. The Arduino polls the distance sensor every 50 ms (and writes a 1 to the "active" column every time movement is detected). If movement is not being detected, it only writes a 0 to the "active" column once every second (as opposed to every 50ms).

Listing 14-4 shows the completed code for the entrance logger, given the improvements just described.

Listing 14-4

Entrance logger software—entrance_logger.ino

```
//Logs Room Entrance Activity

//Uncomment following line if your board uses the DS1307 instead of the PCF8523
//#define RTC_CHIP_IS_DS1307

//Uncomment following line if you want to force the time to be set
//Should always be commented out before "running in the field"
//#define FORCE_UPDATE

#include <SD.h> //Include the SD Card Library
#include <Wire.h> //For RTC I2C
#include "RTClib.h" //For RTC

//SD Card is on standard SPI Pins
//RTC is on standard I2C Pins

//We always need to set the CS Pin for the SD Card
const int CS_PIN = 10;

//The distance sensor analog pin is connected to A0
const int IR_PIN = 0;
```

```
// Use compiler flags to set up the right chip type
#ifndef RTC_CHIP_IS_DS1307
    RTC_DS1307 RTC;
    String chip = "DS1307";
#else
    RTC_PCF8523 RTC;
    String chip = "PCF8532";
#endif

// Use compiler flags to decide if an update should be forced
#ifndef FORCE_UPDATE
    bool update_clock = true;
#else
    bool update_clock = false;
#endif

//Initialize date and time strings
String time, date;

//Initialize distance variables
int raw = 0;
int raw_prev = 0;
boolean active = false;
int update_time = 0;

void updateDateTime()
{
    //Get the current date and time info and store in strings
    DateTime datetime = RTC.now();
    String year = String(datetime.year(), DEC);
    String month = String(datetime.month(), DEC);
    String day = String(datetime.day(), DEC);
    String hour = String(datetime.hour(), DEC);
    String minute = String(datetime.minute(), DEC);
    String second = String(datetime.second(), DEC);

    //Concatenate the strings into date and time
    date = year + "/" + month + "/" + day;
    time = hour + ":" + minute + ":" + second;
}

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

```
//CS pin must be configured as an output
pinMode(CS_PIN, OUTPUT);

//Initiate the RTC library
RTC.begin();

//Always update the time if the RTC isn't running
#ifndef RTC_CHIP_IS_DS1307
    if (!RTC.isrunning()) update_clock = true;
#else
    if (!RTC.initialized()) update_clock = true;
#endif

//If RTC not running or if we force it, set RTC to computer's compile time
if (update_clock)
{
    Serial.print(F("Setting "));
    Serial.print(chip);
    Serial.print(F(" time to compile time..."));
    RTC.adjust(DateTime(F(__DATE__), F(__TIME__)));
    Serial.println(F("Done!"));
}
else
{
    Serial.print(chip);
    Serial.println(F(" time is already set!"));
}

//Show the time
updateDateTime();
Serial.print(F("RTC Date: "));
Serial.println(date);
Serial.print(F("RTC time: "));
Serial.println(time);

//Initialize SD card
Serial.print(F("Initializing SD Card..."));
if (!SD.begin(CS_PIN))
{
    Serial.println(F("Card Failure!"));
    while(1);
}
Serial.println(F("Card Ready!"));

//Write Column Headers
File dataFile = SD.open("log.csv", FILE_WRITE);
if (dataFile)
{
```

```
dataFile.println(F("\nNew Log Started!"));
dataFile.println(F("Date,Time,Raw,Active"));
dataFile.close(); //Data isn't written until we run close()!

//Print same thing to the screen for debugging
Serial.println(F("\nNew Log Started!"));
Serial.println(F("Date,Time,Raw,Active"));
}
else
{
    Serial.println(F("Couldn't open log file"));
    while(1);
}

void loop()
{
    updateDateTime(); //Get the current date/time

    //Gather Motion Data
    raw = analogRead(IR_PIN);
    //If the value changes by more than 75 between readings, indicate movement.
    if (abs(raw-raw_prev) > 75)
        active = true;
    else
        active = false;
    raw_prev = raw;

    //Open a file and write to it.
    if (active || update_time == 20)
    {
        File dataFile = SD.open("log.csv", FILE_WRITE);
        if (dataFile)
        {
            dataFile.print(date);
            dataFile.print(F(","));
            dataFile.print(time);
            dataFile.print(F(","));
            dataFile.print(raw);
            dataFile.print(F(","));
            dataFile.println(active);
            dataFile.close(); //Data isn't written until we run close()!

            //Print same thing to the screen for debugging
            Serial.print(date);
            Serial.print(F(","));
            Serial.print(time);
```

```
    Serial.print(F(","));  
    Serial.print(raw);  
    Serial.print(F(","));  
    Serial.println(active);  
}  
else  
{  
    Serial.println(F("Couldn't open log file"));  
}  
update_time = 0;  
}  
delay(50);  
update_time++;  
}
```

Data Analysis

After loading this code on to your Arduino, set it up at your door and let it run for a while. When you are satisfied with the amount of data you have collected, put the SD card in your computer and load the CSV file into your favorite spreadsheet program. Assuming that you only logged over the course of one day, you can now plot the time column against the activity column. Whenever there is no activity, the activity line graph remains at 0. Whenever somebody enters or exits the room, it jumps up to 1, and you can see exactly when it happened.

The procedure for creating a plot will vary with different graphing applications. To make it easy for you, I've created a preformatted online spreadsheet that will do the plotting for you. You must have a Google account to use it. Visit the web page for this chapter (exploringarduino.com/content2/ch14) and follow the link to the graph-generation spreadsheet. It prompts you to create a new spreadsheet in your Google Drive account (this is free with a Google account). Once this is complete, just copy your data in place of where the template data is, and the graph updates for you automatically. Figure 14-23 shows what a graph of data over a few minutes might look like. The template spreadsheet plots both the raw data and the “active” signal that you generate based on your pre-programmed movement threshold. If your raw data shows spikes where your active signal does not, or vice versa, then you may want to adjust that threshold to achieve better performance.

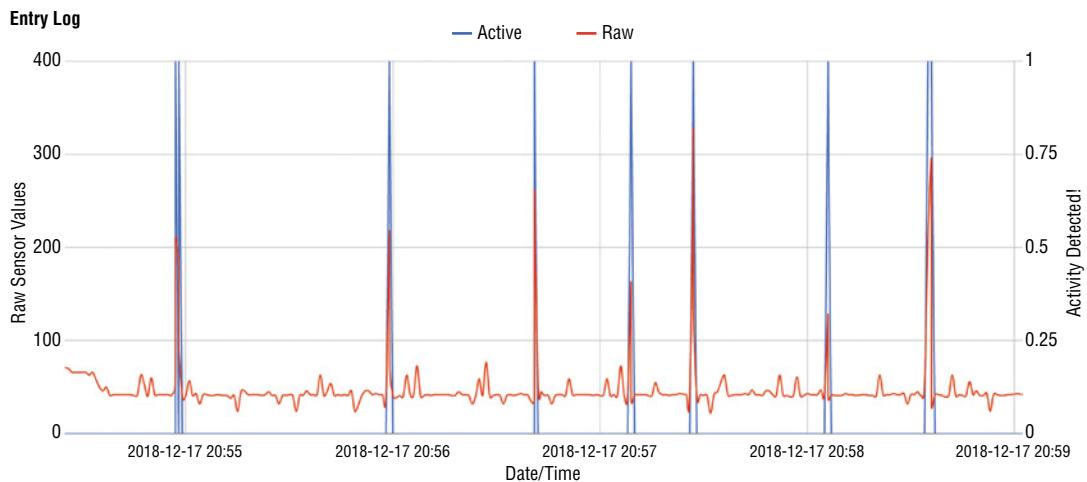


Figure 14-23: Entrance logger data graphed over several minutes

Summary

In this chapter, you learned the following:

- CSV files use newlines and commas as delimiters to easily store data in a plain text format.
- You can format an SD card in Windows, Mac OS, or Linux.
- A plethora of SD card shields are available, each with unique features.
- You can use the SD library to write to and read from a file on an SD card.
- You can interface with an RTC and write software that utilizes it to insert timestamps.
- You can use preprocessor directives and constants to change the way your code is compiled from readable code to machine code.
- You can overcome RAM limitations by storing strings in flash memory.
- You can detect movement by looking for changing analog values produced by a distance sensor.
- You can graph data from a data logger using a spreadsheet on your computer.

V Going Wireless

Chapter 15: Wireless RF Communications

Chapter 16: Bluetooth Connectivity

Chapter 17: Wi-Fi and the Cloud

15

Wireless RF Communications

Parts You'll Need for This Chapter

- Arduino Uno or Adafruit METRO 328
- USB cable (Type A to B for Uno, Type A to Micro-B for METRO)
- Half-size or full-size breadboard
- Assorted jumper wires
- 220 Ω resistor
- Piezo buzzer
- 5V 1A USB port wall power supply
- 315 MHz momentary type RF receiver (or a similar receiver in a frequency appropriate to your country)
- Single-button 315 MHz RF remote control (or a remote in a frequency appropriate to your country)
- Controllable power relay module (IoT Power Relay from Digital Loggers, Inc.)
- AC lamp
- Pocket screwdriver

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at:
exploringarduino.com/content2/ch15

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

It's time to untether! A common requirement in many microcontroller projects is wireless connectivity. There are many ways to achieve wireless connectivity, but the most basic is simple RF (radio frequency) modules. These come in many shapes and sizes, but most have a similar mode of operation: when a single input changes on the transmitter, a matched output changes on the receiver module. Effectively, they operate just like a wire, with one bit of information going in and out at a time. This chapter introduces the concept of wireless communication in general, but specifically focuses on simple RF communications. The next two chapters expand on what you learn here to explain Bluetooth and Wi-Fi connectivity.

NOTE In the first edition of *Exploring Arduino* (Wiley, 2013), I included a chapter about XBee radios, which operate like a wireless serial link. XBee radios are challenging to set up properly, they are expensive, and they only interoperate with each other. In this second edition, I've decided to instead add entire chapters about Bluetooth and Wi-Fi because most people will find them to be more useful and accessible than the proprietary XBee radios. I've also added this chapter about RF links because it lays a better foundation for understanding the electromagnetic spectrum and its implications for radio communications. If you want to learn about XBee radios, I encourage you to read this chapter first, and then head to blum.fyi/xbee-tutorial to watch a tutorial on how to use them.

The Electromagnetic Spectrum

Before you can understand how RF communications operate, it is useful to have a passing understanding of the scientific principles that underpin them. The electromagnetic spectrum may be one of the most important things in your life that you never stop to think about. The spectrum defines all the radiated energy in the universe, ranging from ultra-high-frequency gamma rays to low-frequency radio waves. In the middle of the spectrum, you'll find visible light, the type of electromagnetic radiation you are probably most familiar with. Light from the sun, your lamp, and the LEDs in your Arduino project all reach your eyes in the form of electromagnetic radiation. Similarly to how your eyes interpret this electromagnetic energy as light, an Arduino with an RF receiver can "receive" information in the form of radio waves. Figure 15-1 shows an infographic from NASA that includes all forms of electromagnetic (EM) radiation.

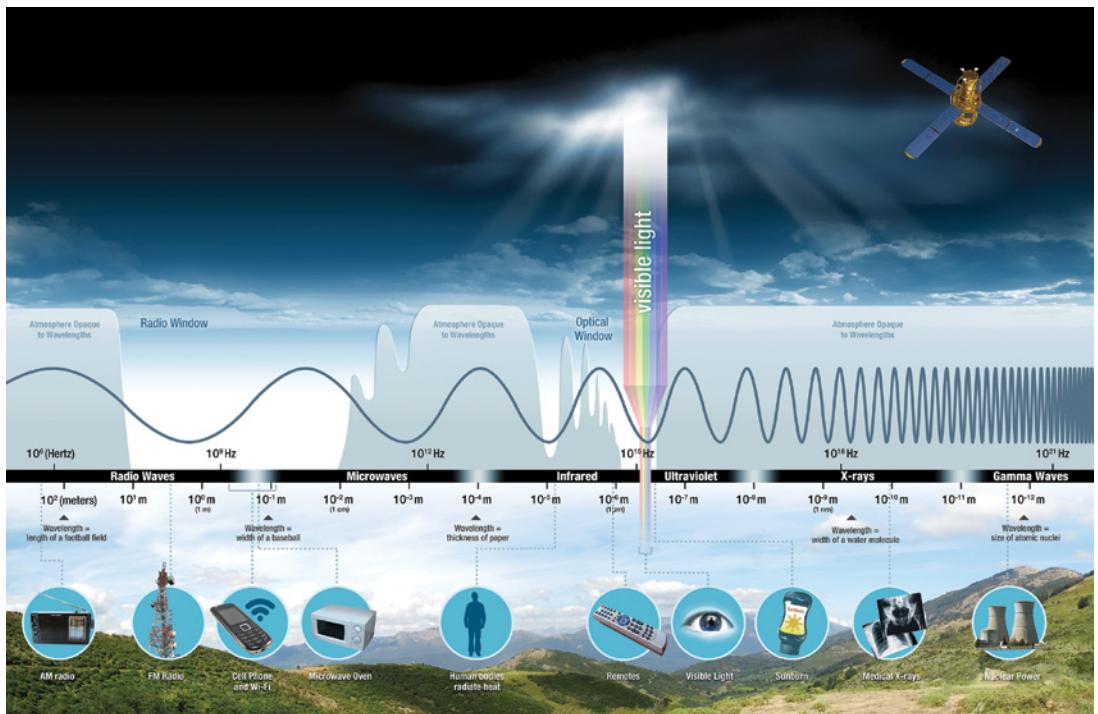


Figure 15-1: Representation of the EM spectrum

Credit: NASA, science.nasa.gov

Frequency, measured in hertz (Hz), represents the number of alternating cycles that a repeating wave makes per second. All electromagnetic energy is transmitted in the form of waves that propagate through free space at roughly the speed of light (about 300,000 kilometers per second). Gamma rays are the highest-frequency wave, at 10 quintillion hertz. On the opposite end of the spectrum are radio waves, which saunter along at frequencies all the way down to about 30 hertz.

Closely related to frequency is the *wavelength* of a radio wave (or any electromagnetic wave). The wavelength represents the physical distance between the peaks of two consecutive waves as an electromagnetic wave propagates through space. The wavelength of an EM wave is just another way of defining the wave—it is always related to the frequency by the propagation speed of the wave. In the vacuum of outer space, the propagation speed is the speed of light, so the wavelength will always equal the speed of light divided by the frequency.

We often use the term *wavelength* when talking about the visible spectrum, and the term *frequency* when talking about radio waves. When you went out to purchase a red LED for your Arduino projects, you may have noticed that its primary wavelength was specified on its datasheet. A red light source produces electromagnetic energy with a wavelength of approximately 630 nanometers. The human eye is capable of detecting EM waves between about 380 nm and 750 nm in the form of color and light. The RF transmitter and receiver used in this chapter operates at 315 MHz. That means the radio waves that it sends through the air have a wavelength of about 0.95 meters (or about half the height of an average person).

The Spectrum

So why do you need to understand wavelengths and frequencies if you just want to send a simple on/off signal wirelessly? Picking the right transmission frequency can have a huge impact on the speed of your data transfer, its ability to pass through certain materials (like the walls in your home), and whether or not you are legally able to transmit in your region. Most of the wireless spectrum is reserved for special applications like military transmissions, over-the-air TV signals, AM/FM radio, police and emergency radio services, satellite communications, cell phones, and so on.

You may have read news articles about radio spectrum allocation in the context of your smartphone. There are frequent debates about how the radio spectrum should be allocated to various technologies such as cellular data, Wi-Fi, and others. If everybody were allowed to transmit whatever they wanted at any frequency and power level, the airwaves would quickly become a huge mess of interference that no receiver would be able to interpret. To combat this, various governing bodies around the world define specific operating rules for devices that transmit. Spectrum allocation and standards are set by the FCC (Federal Communications Commission) in the United States, by IC

(Industry Canada) in Canada, and by CISPR (Comité International Spécial des Perturbations Radioélectriques) in the European Union. In large part, the international standards are compatible with each other—that is, if a portion of the spectrum is allocated for a particular use in one place, it often is in the other places as well. However, there are a few exceptions, so it's the responsibility of the person designing the transmitter to understand local laws.

The examples in this chapter use a 315 MHz transmitter, which falls under rules for the 260–470 MHz band in the United States. This band permits the use of unlicensed transmitters, so long as they comply with various requirements for type and power of transmission. For example, you are not allowed to transmit voice or video, but you are allowed to transmit command signals in short bursts. This frequency range is often chosen for garage door openers and key fobs for cars because it enables a relatively long transmission range, while not experiencing interference from continuous data streams. Note that in Europe, unlicensed transmission at 315 MHz is not permitted, but transmission at 433 MHz is allowed under the relevant CISPR regulations. 433 MHz is a common frequency that is chosen because it is universally permitted—this makes it easier to build a single product for multiple markets.

Another part of the spectrum that you may be familiar with is the *ISM band*. The ISM, or industrial, scientific, and medical radio band, is a set of frequency ranges that are globally allocated for general-purpose use (again, with some minor differences from region to region). In the ISM bands, you'll find cordless phones, RFID, Bluetooth, Wi-Fi, and microwave ovens. Again, just because the ISM band is free to use, it doesn't mean you can broadcast anything you want—you still have to follow regional rules that define transmit power. These rules ensure that appliances like your microwave don't knock out your home Wi-Fi. (They both operate at 2.4-2.5GHz, which is one of the ISM bands.)

Don't stress out too much about the rules around your spectrum use. In the case of Arduino RF accessories, reputable stores will only sell modules that are authorized for use in the country where they are being sold. The Bluetooth and Wi-Fi projects that you'll undertake in the next two chapters operate safely within ISM bands. For this chapter, just be sure to use a module that is listed as compatible with your country's rules.

How Your RF Link Will Send and Receive Data

Okay, so now you understand that your RF transmitter will be sending short bursts of data at a particular approved frequency. But how is it actually doing that? How does the receiving end differentiate between a logic HIGH and a logic LOW being sent by the transmitter at the transmission frequency (315 MHz or 433 MHz in this case)?

The answer to this question depends on the *modulation* being used by the transmitter and receiver. As long as you use the same modulation technology on the sending and receiving ends, you don't have to worry too much about the details of how the modulation works. At a high level, modulation is the process of encoding data into a *carrier wave* (an electromagnetic wave at the frequency that you are transmitting). You are probably already familiar with two kinds of analog modulation: AM and FM radio. *Amplitude modulation* (AM) and *frequency modulation* (FM) encode analog data, like your favorite song on the radio, by changing (or modulating) the amplitude or frequency of the carrier wave, as shown in Figure 15-2. On the receiving end, the original audio signal can be extracted from this wave by recovering the original signal from the changes in amplitude or frequency.

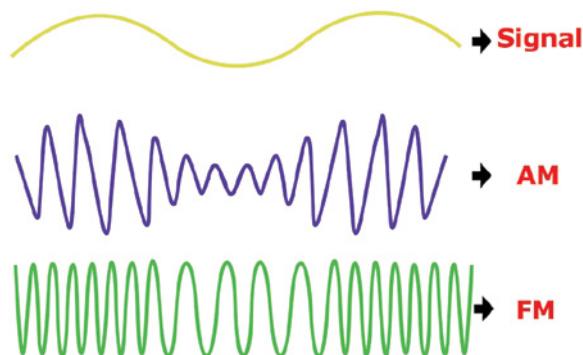


Figure 15-2: AM and FM modulation of an analog signal

Credit: Science ABC, scienceabc.com

Digital modulation is similar to those analog modulation techniques. Instead of encoding an analog signal (like audio), it encodes zeros and ones. The modules used in this chapter, like most simple key transmitters (garage door openers, key fobs, and so on), use ASK or *amplitude-shift keying* modulation. This is very similar to the way in which AM radio transmits, by adjusting the amplitude, or strength, of the signal while maintaining the carrier wave frequency. Figure 15-3 shows the theory of operation. At a basic level, if you receive a signal at the expected frequency for a fixed period of time, that is a logic 1 signal. If you don't receive a signal for a fixed period of time, that is a logic 0 signal.

A little bit more goes into the transmission and receiving; basic addressing and error-checking bits must be transmitted to ensure complete chunks of data are received without corruption. The modules you'll use in this chapter take care of all of this for you; you just have to worry about receiving the decoded logic signals.

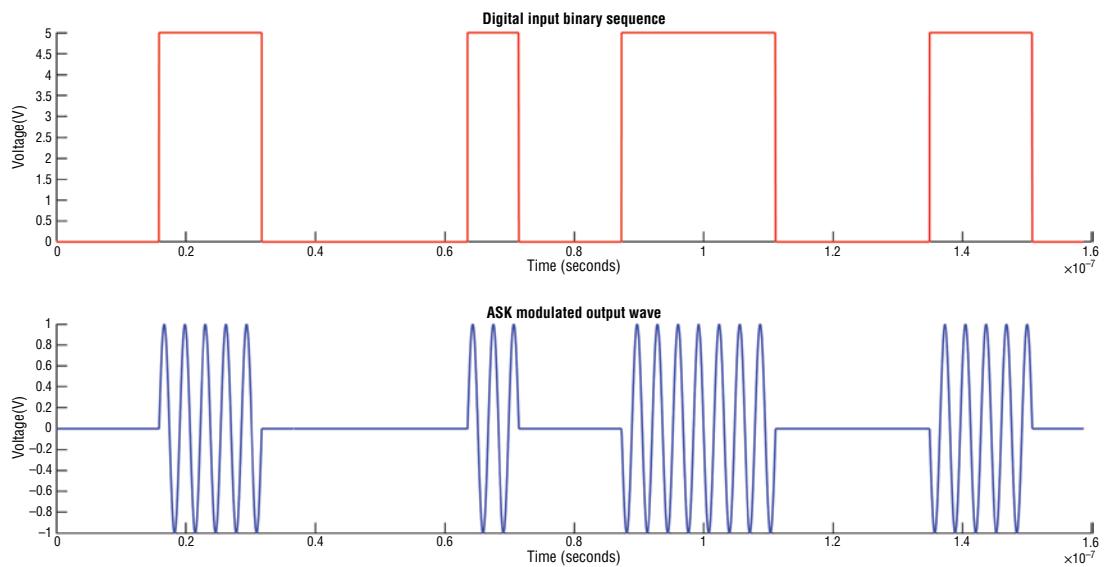


Figure 15-3: ASK modulation of a digital signal

Created with MATLAB

Receiving Key Presses with the RF Link

Now that you understand how wireless transmission works, it's time to set up your wireless receiver.

Connecting Your Receiver

The particular receiver module used for this chapter (product ID 1096 from Adafruit) can actually receive up to four keypress events. If you are using the recommended remote with a single button, then that button triggers the pin labelled D2 on the receiver module. D2 stays at 5V while a remote key is held down, and returns to 0V if there is no keypress signal from a remote. Your Arduino can receive that 0V–5V signal on one of its digital inputs, and take action based on its state.

First, to ensure you get the maximum range from your remote, I suggest you (carefully) uncoil the antenna as shown in Figure 15-4.

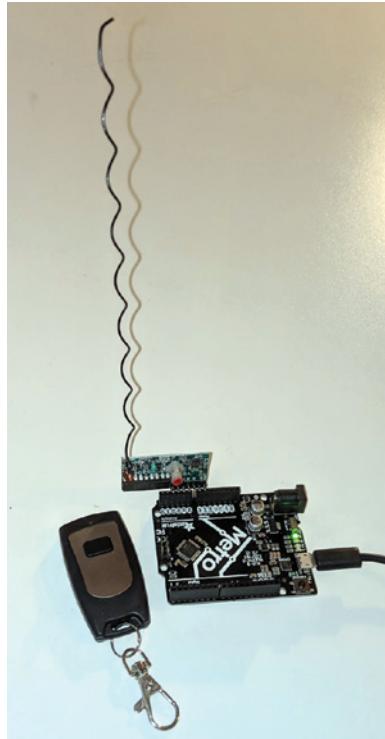


Figure 15-4: Antenna fully extended

NOTE If you measure the antenna wire, you'll find that it is approximately 0.24 meters long. That's because this is a type of antenna called a "quarter-wave monopole" antenna. Its length is equal to one-quarter of the wavelength of the frequency it is matched to; 315 MHz has a wavelength of about 0.96 meters (or four times the length of this antenna).

Once your antenna is uncoiled, you can connect the receiver module directly to your Arduino. If you are using an Uno, a Leonardo, or a compatible board (such as the Adafruit METRO 328, which works identically to the Arduino Uno, and is shown in the figures throughout this chapter), it's possible to connect the module directly. Position it such that D0 sits in between the Analog header pins and power pins. This aligns the module such that its 5V input is supplied by VIN (5V if powered over USB), its GND input is connected to GND, and D2 is connected to the A1 pin on the Arduino board. Recall that analog input pins can be used as digital inputs as well. Figure 15-5 shows a close-up of this connection.

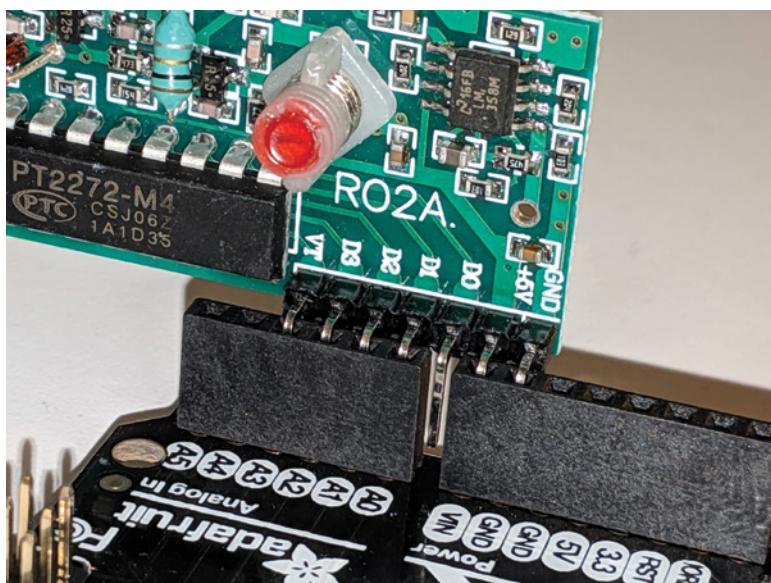


Figure 15-5: Module directly installed

Programming Your Receiver

Once you have the receiver connected to the Arduino, you're ready to receive signals. If your transmitting remote came with a plastic pull-tab installed, be sure to remove it—its purpose is to isolate the battery in shipping so it doesn't discharge before you get it.

Because the module used in these examples is a “momentary” variant, its D2 pin will be HIGH whenever the remote button is held down. To test this out, write a program that monitors the value of pin A1 (where D2 from the module is connected), and sets the LED on the Arduino board to be on whenever that pin is HIGH, and off whenever that pin is LOW. This program is incredibly simple, and you can create it with just one `setup()` line to set the LED pin as an output, and one `loop()` line to set the LED output to the value of the digital input:

```
void setup(){digitalWrite(13, OUTPUT);}
void loop(){digitalWrite(13, digitalRead(A1));}
```

That’s really the entire program! Note the use of functions inside of other functions. For very simple programs like this one, it’s not necessary to set the outputs of functions to intermediate variables. In this example, you could perform the `digitalRead()` first, assign its value to a Boolean variable, and then feed that value into the `digitalWrite()` function. However, you can shortcut that process by performing the digital reading directly inside the outer function. It executes and substitutes its output value (HIGH or LOW) into that argument position.

NOTE If you find that pin D2 on the receiving module is not triggering, it’s possible that your transmitting remote is configured to trigger one of the other channels. Try to read the signal generated by the VT pin from the module instead (plugged into A3 on the Arduino); that pin should trigger if any of the input channels is triggered. If VT does work, then use the process of elimination to try the other channels to see which one is receiving the signals from your remote. Also ensure that your remote blinks a red LED when you press the button. If it doesn’t, its battery may be dead.

How about doing something a little more interesting? In addition to controlling the LED, you can use the `millis()` function to measure the amount of time the button has been held down, and output the duration to the serial console at the end of each button press. To accomplish that, you need to add some variables that keep track of state. The code in Listing 15-1 will do exactly that.

Listing 15-1

A simple test of the RF link—`rf_test`

```
//A Simple Test of the RF Link

//Assumes that the MOMENTARY Type RF Receiver is being used!
//The M4 momentary type RF receiver acts like a push button.
//When the remote button is held down, the D2 pin on the module goes HIGH.
```

```
//When the remote button is released, the D2 pin on the module goes LOW.  
  
//The Arduino I/O Pin connected to pin labeled "D2" on the RF Module  
const int TRIGGER_PIN = A1;  
  
//We'll just light up the on-board LED while the button is held  
const int LED_PIN = 13;  
  
//Initialize variable for holding press start time  
unsigned long start_time;  
  
//And variable for tracking state of whether we announced a new press  
boolean announced;  
  
void setup()  
{  
    Serial.begin(9600);  
    Serial.println("RF Test");  
  
    //LED pin must be configured as an output  
    pinMode(LED_PIN, OUTPUT);  
}  
  
void loop()  
{  
    digitalWrite(LED_PIN, LOW); //LED off when unpressed  
    announced = false; //So we only announce a new press once  
  
    //Stay in the following loop while button is held  
    while (digitalRead(TRIGGER_PIN))  
    {  
        //Once-per-press, do this:  
        if (!announced)  
        {  
            start_time = millis();  
            Serial.print("PRESSED...");  
            announced = true; // Only print message once per hold  
        }  
        digitalWrite(LED_PIN, HIGH);  
    }  
  
    //Once the button is released, note how long it was held for  
    if (announced)  
    {  
        Serial.print("RELEASED after ");  
        unsigned long duration = millis() - start_time;  
        Serial.print(round(duration/1000.0)); //Print press duration in seconds  
        Serial.println(" second(s).");  
    }  
}
```

Note the use of so-called *state variables* to keep track of your serial notifications and press duration. The announced variable gets reset to false between each detected button press (outside of the `while()` loop, but inside the main `loop()` function). The `while()` loop inside the `main loop()` continues to execute while the button is held down. On its first iteration, the announced variable is set to true. After that, `!announced` returns false (recall that the exclamation point inverts the value of the variable it is placed in front of), so it does not print the `PRESSED` message again or restart the `millis()` timer. Once the `while()` loop is exited (because the button is released, or the signal is lost), the main program `loop()` returns to waiting for the button press. Because the announced variable is only set to true inside the `while()` loop, the `if (announced)` block only executes one time (right after button release) because on the next run through the main program `loop()`, that announced variable is set back to false again.

When you run this code, it still lights up the onboard LED as it did before. However, now, it also notes the system time (using `millis()`) at the start of each button press, and then reports the total button press duration once the button is released. Figure 15-6 shows a screenshot of a serial monitor with the output from this test.

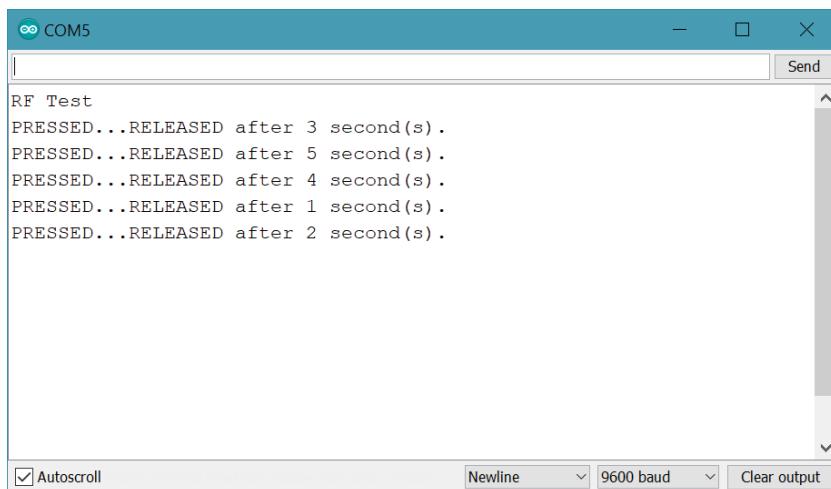


Figure 15-6: Screenshot of serial output from RF test

If you are having issues with your signal reception being intermittent, try getting closer to the antenna, or readjusting it. The range on these simple transmitters is not incredible, but you should be able to send a consistent signal from a few feet away.

Don't worry if single button presses are being broken up into multiple "clicks" because of signal drops; the remaining projects in this chapter use the remote as a single-click on/off button, which makes that less of an issue.

Making a Wireless Doorbell

Lighting up a tiny LED from across the room isn't particularly interesting. Next, you'll use your newly found wireless powers to make something a bit more useful: a wireless doorbell. This is a simple project that you can use outside your bedroom, dorm room, or office, without needing to punch a hole in your walls to wire up a traditional doorbell. You can get creative with the RF remote if you want; for example, you can take the PCB out of the case and mount it in a custom enclosure.

Wiring the Receiver

First, you need to get a speaker or piezo buzzer wired to your receiving end. Because this is easiest to do on a breadboard anyway, you can also move your RF receiver over to the breadboard. Wire it up so that it is getting 5V from the Arduino, and so that D2 (on the receiver) is connected to an input of your choice on the Arduino. If you connect it to pin 13, then the onboard LED illuminates whenever a signal is received, without you having to program explicit logic for that function. Connect your piezo buzzer or speaker through a 220Ω resistor, as you have done in previous projects. A 150Ω resistor will also work fine, and produce a slightly louder sound. Once your receiving end is wired up, it looks like Figure 15-7. Don't forget to make sure that 5V and ground are connected from the Arduino to the bus lines on the breadboard.

Programming the Receiver

Your receiving Arduino acts as your doorbell chime. With its RF receiver, it waits for a signal from your remote, and plays a song when one is received. To ensure it works reliably, you need to make sure that you only play the song once per click, that multiple rapid clicks don't interrupt the currently playing tune, and that holding the remote button down for an extended period doesn't make the tune play forever. Copy the `pitches.h` file from code that you used in Chapter 6, "Making Sounds and Music," into the folder of your doorbell sketch. You can use the same tune that was used in the code samples from Chapter 6. The file is also included as part of the code download for this chapter at exploringarduino.com/content2/ch15.

The code in Listing 15-2 waits for a button press to be received, plays a tune, and then starts waiting for another button press.

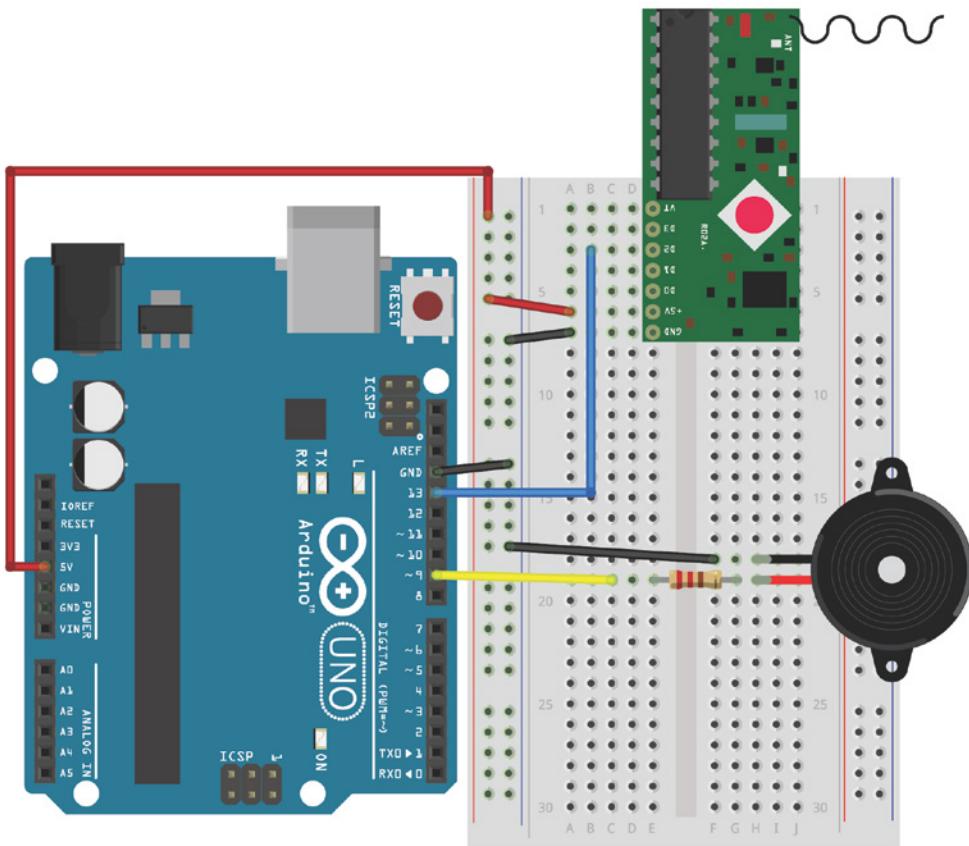


Figure 15-7: Wireless doorbell receiver

Created with Fritzing

Listing 15-2

A doorbell using the simple RF receiver–doorbell

//A Doorbell using the Simple RF Receiver

//Assumes that the MOMENTARY Type RF Receiver is being used!

//The M4 momentary type RF receiver acts like a push button.

//When the remote button is held down, the D2 pin on the module goes HIGH.

//When the remote button is released, the D2 pin on the module goes LOW.

```
#include "pitches.h" //Header file with pitch definitions
```

//The Arduino I/O Pin connected to pin labeled "D2" on the RF Module

```
const int TRIGGER_PIN = 13;
```

```
const int SPEAKER = 9; //Speaker Pin
```

```

//Note Array
int notes[] = {
    NOTE_A4, NOTE_E3, NOTE_A4, 0,
    NOTE_A4, NOTE_E3, NOTE_A4, 0,
    NOTE_E4, NOTE_D4, NOTE_C4, NOTE_B4, NOTE_A4, NOTE_B4, NOTE_C4, NOTE_D4,
    NOTE_E4, NOTE_E3, NOTE_A4, 0
};

//The Duration of each note (in ms)
int times[] = {
    250, 250, 250, 250,
    250, 250, 250, 250,
    125, 125, 125, 125, 125, 125, 125, 125,
    250, 250, 250, 250
};

void setup()
{
//No setup necessary
}

void loop()
{
    //While the button is held high, play the song once.
    if (digitalRead(TRIGGER_PIN))
    {
        for (int i = 0; i < 20; i++)
        {
            tone(SPEAKER, notes[i], times[i]);
            delay(times[i]);
        }
        //In case the button is still being held down after the song finishes,
        //wait here until it is released, before playing again
        while(digitalRead(TRIGGER_PIN));
    }
}

```

Inside the main `loop()`, the state of the trigger `TRIGGER_PIN` is checked by the `if()` statement. Once triggered, a `for()` loop plays through the `notes[]` array in order. Before exiting the `if()` statement, the single-line `while()` loop blocks further progression if the remote doorbell button is still being held down. This prevents the song from looping forever if the person pressing the doorbell is simply holding it down. The code does not resume the main `loop()` until the button is released. At that point, it starts waiting for another button press.

NOTE To watch a demo video of the RF doorbell, visit exploringarduino.com/content2/ch15.

The Start of Your Smart Home—Controlling a Lamp

Perhaps one of the most exciting opportunities introduced by wireless technology is the ability to seamlessly connect items in the physical world together, with minimal infrastructure changes. In the next two chapters, you learn about leveraging Bluetooth and Wi-Fi to interact with your Arduino. However, long before those technologies existed, RF links were used to enable “smart” devices.

Expanding on the doorbell receiver you’ve already built, you can add a controllable AC relay that can switch standard wall-voltage devices on or off! Pairing this technology with the RF remote, you can easily make a nightlight that you can turn on and off from bed, or make a reading lamp with a hard-to-reach inline switch easier to use. Think of a relay as a light switch that can be controlled by a low-voltage logic signal. Apply 5V and the switch “closes,” allowing current to flow through the connected appliance.

WARNING This section details how to control an AC appliance running off the 120V or 230V service in your home. These voltages can *kill you* if handled improperly. Only use certified products that are explicitly designed to safely switch these kinds of loads. The examples in this and later chapters use an enclosed relay box specially designed for this purpose, but it is designed only for use with North American 120V outlets. Do not use it in a 230V country (as similar products are available for those countries). It is possible to purchase relay shields for the Arduino that can switch these loads, but they are not enclosed. If you choose to use one of those shields, you do so at your own risk. Always ensure that high-voltage wire is *not* exposed when operating a device like this.

AC POWER TRANSMISSION

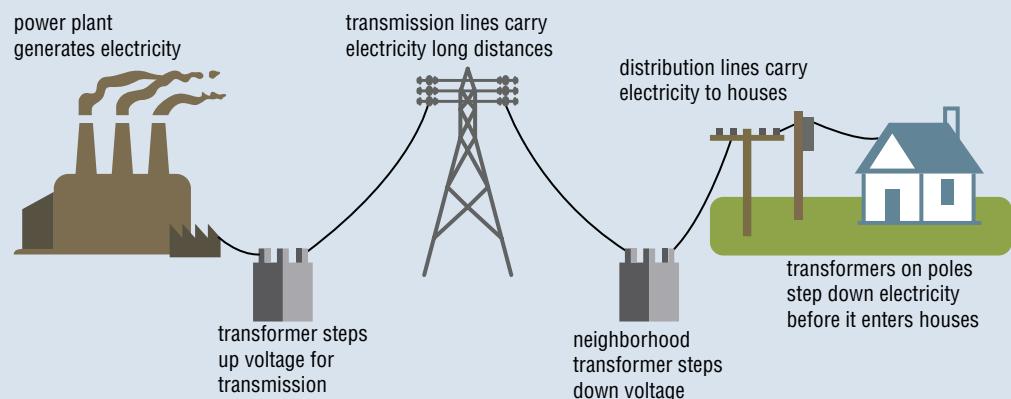
Your home uses AC (alternating current) power. Unlike DC (direct current), AC power oscillates 50 to 60 times a second (the exact rate depends on what country you are in). The actual voltage swings also vary from country to country. North and Central America operate between 110V and 127V (often just approximated to 120V), Japan operates at 100V, and most of the rest of the world operates between 220V and 240V (often just approximated to 230V). Some devices are designed to operate at universal voltages and frequencies (like your laptop or phone charger). Other devices must be designed for a specific voltage and/or frequency. For example, brushed AC motors must be designed for a specific voltage, because the frequency and amplitude of the AC voltage input directly correlates to the resulting rotational speed of the motor.

(Continued)

AC power is utilized in your home and for long-distance powerline transmission because it is far more efficient over long distances compared with DC, and because its voltage can be stepped up or down very easily. A transformer can step the voltage of an AC waveform up or down. The detailed operating physics of a transformer are beyond the scope of this book, but at a high level, the back-and-forth alternation of the direction of current flow enables electromagnetic energy transfer to occur at voltage ratios that can be determined by the number of windings in the transformers' internal coils. Power plants transform the electricity up to an extremely high voltage (more than 115,000 volts), and transmit the power over high-voltage lines; then, local substations and transformers step it back down to 120V or 230V before it enters your home.

Recall that overall power is equal to the voltage multiplied by the current. More current means more energy loss (in the form of heat) due to the impact of line resistance. By transmitting at a high voltage and low current, transmission lines can be kept to a smaller diameter and will lose less energy to heat over the length of the wire. Closer to your home, the step-down transformer decreases the voltage, thus increasing the current to recover the original power (minus efficiency losses). Figure 15-8 shows how power gets from a power plant to your home.

Electricity generation, transmission, and distribution



Source: Adapted from National Energy Education Development Project (public domain)

Figure 15-8: AC power transmission

Credit: U.S. Energy Information Administration (Public Domain), eia.gov

Your Home's AC Power

The AC power that arrives at your home or office is routed around to your outlets and light fixtures using three wires: Line, Neutral, and Earth. Just like with DC power, AC power must always flow in a loop. Compare Line to your positive DC voltage rail, and Neutral to your DC return path (what is commonly called *digital ground*). The Earth wire serves a protective purpose, and is literally connected to the Earth (a pole driven deep into the ground, or bonded to your water pipe) at your home's electrical junction box. Appliances bond the Earth wire to their metal enclosure.

If your appliance malfunctions, causing wall voltage to connect to the appliance's metal enclosure, the Earth wire is designed to provide a low-resistance path to the point of lowest electrical potential, the *literal* Earth. Without the Earth wire providing that path, your body would make a pretty good conductor, and the current would flow through your body, resulting in a painful electric shock when you touch a damaged piece of electrical equipment. Devices that do not have an Earth ground connection, must instead be *double insulated*, meaning that two layers of protective insulators must exist between the high-voltage components and any part that can be touched by a user. Your phone's AC USB adapter is an example of a product that is double insulated. A desktop computer, on the other hand, probably uses an "Earthed" cable, with the Earth wire bonded to its metal chassis for protection.

Different countries use different color codes for their AC wires. Because this book exclusively recommends the use of an enclosed AC relay product for safety reasons, you do not need to wire any AC cables yourself. In the United States, black wire insulation is used for the AC Line (also sometimes called "Hot" or "Live"), white wire insulation is used for the AC Neutral, and green wire insulation is used for the Earth. Single pole relays should always switch the Line wire, not the Neutral wire.

How a Relay Works

This project uses a mechanical relay to control an ordinary AC lamp. Relays are ubiquitous devices that leverage electromagnetic principles to aid in the isolated switching of a high-voltage and/or high-current load (like an AC lightbulb) using a low-voltage device (such as the microcontroller on your Arduino). Relays come in a variety of shapes, sizes, ratings, and configurations. Some are designed to "latch" to an on or off position. Some can switch multiple signals or loads at once. Some operate on a "make before break" principle where they connect a new load to a common input before breaking the connection to the previous load (these are often used in telecommunication systems). The relay you'll use for this project (the IoT Power Relay from Digital Loggers, Inc.) is already enclosed inside a protective, prewired case with appropriate

driving and isolation/protection electronics. Inside the protective case is a Single Pole Double Throw (SPDT) relay that operates as follows:

- When its control input is held at logic HIGH (5V), it allows current to flow from the “common” (input) pin to the outlets labelled “normally OFF.”
- When its control input goes LOW (0V), the relay switches, and instead connects the “normally ON” outlets to input power.

The SPDT relay’s “common” pin is connected to the AC Line wire that enters the box. Figure 15-9 shows a simplified wiring diagram of what the circuit board inside this IoT Power Relay looks like.

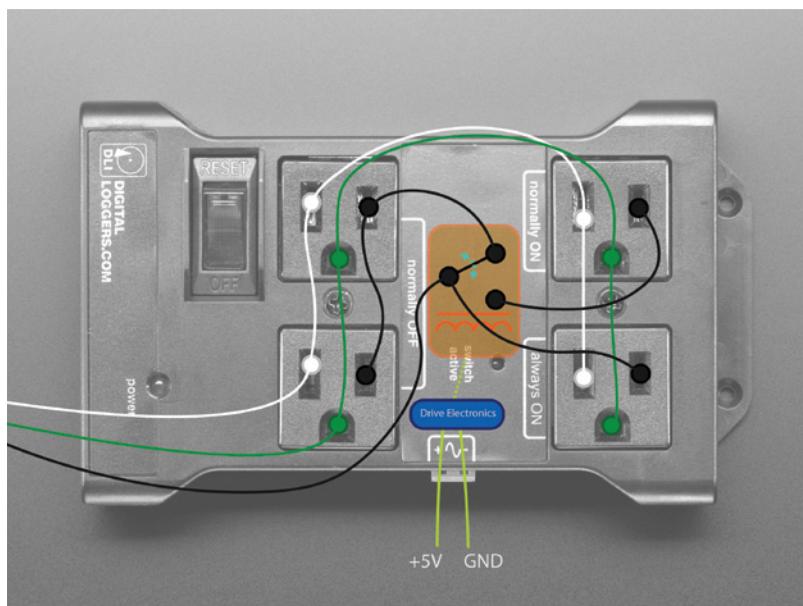


Figure 15-9: Simplified relay wiring

Credit: Adafruit, adafruit.com; wiring overlay by author

The three previously described wires all enter the box via the power inlet: white is Neutral, black is Line, and green is Earth. The Earth and Neutral wires connect to all the corresponding ports on the outlets. The Line wire connects to the “common” pin of the SPDT relay.

In Figure 15-9, 5V is applied to the control input of the IoT Relay. This input voltage causes current to flow through the orange coil (the squiggly line with the bar above it).

In practice, the control input is switching an optically isolated transistor, which is then allowing current to flow through the coil from a source voltage generated from the AC inlet.

When the current flows through a coil in a mechanical relay, the sudden change in current results in the generation of an electromagnetic field that throws the switch (denoted by the dotted blue line on top of the black switch “arm”). When the input is at 5V, the switch is held in that position, thus connecting the “common” pin to the pin that powers the “normally OFF” outlets. When 5V is removed from the input, the coil stops drawing current, and the lever flips back to the default position that connects the Line wire of the “normally ON” outlet to the “common” pin. It’s called “normally ON” because when the digital input is LOW (the default state), that outlet is on. Once you get it wired up, you can actually hear the relay arm click into place when you apply and remove the logic signal.

Programming the Relay Control

You can simply add a line that toggles the output of a digital pin to the previous sketch you wrote. However, if you track the state of the output, then you can play a different tune when the light turns on or off! In your software for the lamp control, try that approach. Add a Boolean state variable that inverts its value each time a remote press is detected. On each press, play a different tune using the piezo buzzer that you have hooked up, and toggle the output to the lamp control. The sketch in Listing 15-3 does just that. To keep things even simpler, it just plays the tune forward when the light turns on, and then plays the same notes in reverse order when the light turns off. This can be easily accomplished by using a `for()` loop with inverted counting logic. Whereas `for (int i = 0; i < 3; i++)` starts a counter at 0, and increments it by 1 until it hits 3, `for (int i = 2; i >= 0; i--)` starts a counter at 2 and decrements it by 1 until it is equal to 0. That allows you to traverse backwards through the `notes[]` array.

Listing 15-3

RF lamp controller-lamp_remote

```
//A Remote Control for a Lamp

//Assumes that the MOMENTARY Type RF Receiver is being used!
//The M4 momentary type RF receiver acts like a push button.
//When the remote button is held down, the D2 pin on the module goes HIGH.
//When the remote button is released, the D2 pin on the module goes LOW.

#include "pitches.h" //Header file with pitch definitions
```

```
//The Arduino I/O Pin connected to pin labeled "D2" on the RF Module
const int TRIGGER_PIN = 13; //Input from RF Module
const int SPEAKER = 9; //Speaker Pin
const int LAMP = 2; //Lamp Control

//Note Array
int notes[] = {NOTE_E3, NOTE_A4, NOTE_C5};

//The Duration of each note (in ms)
int times[] = {250, 250, 250};

//Default lamp to OFF
bool lamp_on = false;

void setup()
{
    pinMode(LAMP, OUTPUT); //Lamp Pin is an Output
    digitalWrite(LAMP, lamp_on); //Turn the Lamp off (this variable starts as
false)
}

void loop()
{
    //When the button is pressed, change the state of the lamp
    if (digitalRead(TRIGGER_PIN))
    {
        lamp_on = !lamp_on; //Invert the state of the lamp control variable
        digitalWrite(LAMP, lamp_on); // Set the lamp to its new state

        //Play a different sound depending on whether the lamp turned on or off
        if (lamp_on)
        {
            // Play a tune for turning the lamp on
            for (int i = 0; i < 3; i++)
            {
                tone(SPEAKER, notes[i], times[i]);
                delay(times[i]);
            }
        }
        else
        {
            // Play a tune for turning the lamp off (same song, backwards
            for (int i = 2; i >= 0; i--)
            {
                tone(SPEAKER, notes[i], times[i]);
                delay(times[i]);
            }
        }
    }
}
```

```
//In case the button is still being held down after the song finishes,  
//wait here. This effectively debounces the remote signal.  
while(digitalRead(TRIGGER_PIN));  
}  
}
```

Load this code onto your Arduino; then proceed to the next section, where you wire the relay into your existing circuit.

Hooking up Your Lamp and Relay to the Arduino

With the software ready to go, you just have to connect the relay box to power and to your Arduino. With the relay box unplugged, pull out the green terminal block. Loosen the screws on the terminal block and screw in two jumper wires. Plug the positive wire into pin 2 of your Arduino. Plug the negative wire into the ground on your Arduino.

Your Arduino needs power. Once it is programmed, unplug it from your computer and get a USB AC adapter. Plug your Arduino into the adapter, and plug that into the “always ON” outlet of the relay box. Then, plug the relay box into wall power, and switch it to the ON position. Finally, plug the lamp you want to control into one of the “normally OFF” outlets. If your lamp has a separate power switch, make sure it is switched ON. Once you have everything connected, it should look like Figure 15-10.

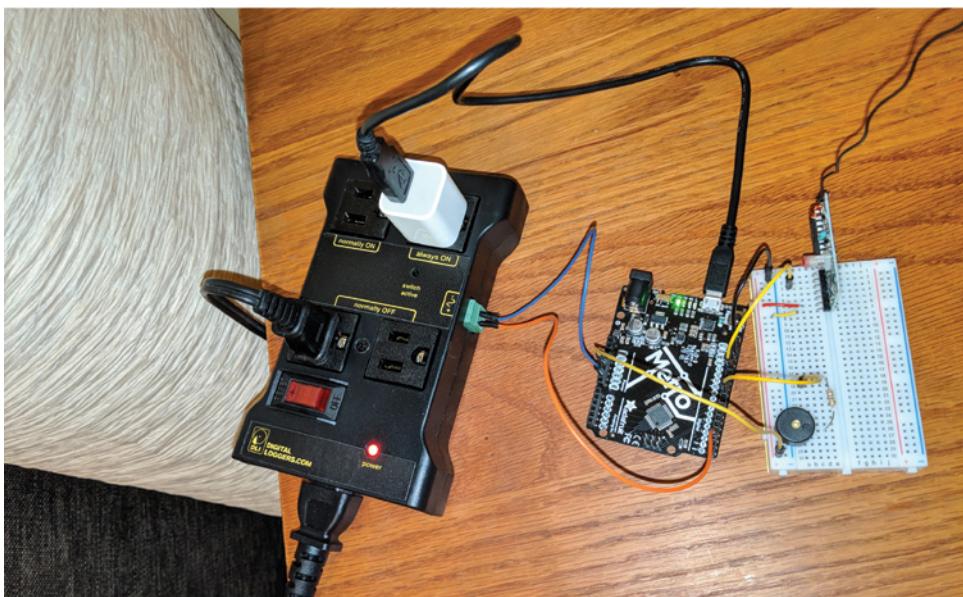


Figure 15-10: Lamp wired to Arduino with a relay

You're ready to control the lamp! Click your remote, and watch in amazement as your lamp turns on, and your Arduino plays a happy tune. When you click the remote again, another tune plays and the lamp turns off.

NOTE Watch a demo video of the RF Arduino AC lamp relay controller at exploringarduino.com/content2/ch15.

Summary

In this chapter, you learned the following:

- Radios transmit and receive information by modulating data at different frequencies on the electromagnetic spectrum.
- Only specific frequencies can be used for various types of transmissions, and they vary from country to country.
- You can control simple functions on your Arduino by wirelessly transmitting keypress data.
- You can make a doorbell by listening for RF key presses and triggering an audio response.
- Appliances in your home use AC power.
- You can use a relay to switch a high-power appliance on and off with a low-power microcontroller.
- You can wirelessly turn an AC lamp on or off by receiving RF signals with your Arduino and driving a relay.

16

Bluetooth Connectivity

Parts You'll Need for This Chapter

- Adafruit Feather 32u4 Bluefruit LE (pre-soldered)
- USB cable (Type A to Micro-B)
- Half-size or full-size breadboard
- Assorted jumper wires
- 220 Ω resistor
- 10k Ω trim potentiometer (or another analog sensor of your choice)
- 5 mm red LED
- 5V 1A USB port wall power supply
- Controllable power relay module (IoT Power Relay from Digital Loggers, Inc.)
- AC lamp
- Pocket screwdriver
- BTLE-capable smartphone (iPhone or Android)

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at: exploringarduino.com/content2/ch16

Code for this chapter can also be obtained from the Downloads tab on this book's Wiley web page:

wiley.com/go/exploringarduino2e

It probably didn't take long for you to grow bored of using simple RF communications to send a single bit of information to your Arduino wirelessly. As you know from using your laptop, smartphone, or IoT (Internet-of-Things) devices, it's

Exploring Arduino®: Tools and Techniques for Engineering Wizardry, Second Edition.

Jeremy Blum.

© 2020 John Wiley & Sons, Inc. Published 2020 by John Wiley & Sons, Inc.

possible to do a *lot* more with the right type of wireless technology. In this chapter, you'll learn about Bluetooth technology, one of the world's most popular wireless standards. Variants of Bluetooth technology are used in headsets, keyboards, mice, computers, smartphones, location beacons, and more.

Demystifying Bluetooth

“Bluetooth” has become an overloaded term that is often (incorrectly) used to describe any short-range, point-to-point wireless communication. Bluetooth, by design, is actually implemented differently from device to device, which is part of what often generates confusion when saying that something supports Bluetooth. Before you can understand how to implement software that leverages Bluetooth, it's worth understanding the various Bluetooth versions, profiles, and terminologies.

Bluetooth Standards and Versions

Originally standardized as IEEE 802.15.1 by the Institute of Electrical and Electronics Engineers, Bluetooth technology is now managed by the Bluetooth Special Interest Group, or SIG.

NOTE Bluetooth is one of many standards developed and maintained by the Institute of Electrical and Electronics Engineers (IEEE). Others that you may be familiar with include Wi-Fi (IEEE 802.11); Power over Ethernet (IEEE 802.3); POSIX, the underpinnings of compatibility between variants of Unix operating systems (IEEE 1003); FireWire (IEEE 1394); and many more.

Bluetooth specifications have evolved through several versions, with each version adding new features and enhancements to the technology. The first version was released in 1998 and was predominately aimed at providing a wireless replacement for serial links (the same kinds of serial links that you've used in previous chapters to print messages from your Arduino to your computer). In 2004, Bluetooth 2.0 added support for faster data rates, and paved the way for using Bluetooth as a digital audio link (probably the use case you are most familiar with). If you've ever struggled with pairing a set of Bluetooth headphones, you aren't alone. The original specification wasn't explicitly designed to stream audio, so integrating this functionality into the Bluetooth specifications, as well as the radios and processors, while maintaining backwards compatibility was not an easy task for the Bluetooth SIG. With the release of 2.0 in 2004 and 3.0 in 2009, hands-free headsets gained popularity, as more cellphones integrated Bluetooth radios.

Perhaps the most important Bluetooth revision so far was Bluetooth 4.0 in 2013. Bluetooth 4.0 introduced Bluetooth Low Energy, often abbreviated BTLE or BLE and

sometimes referenced by its marketing name, Bluetooth Smart. This introduction was particularly significant because it effectively broke the Bluetooth specification into two separate implementations: Bluetooth Classic and Bluetooth Low Energy.

Bluetooth Classic is backward compatible with all the previously established versions, supports higher data transfer rates, has support for various higher-bandwidth profiles (for streaming music and the like), and consumes more power. Bluetooth Low Energy consumes considerably less power, and is explicitly designed to support simple, battery-powered devices that only require minimal data transfer bandwidth. Examples include heart rate monitors, environmental sensors, smart home remotes, and more. These devices aren't streaming multimedia data; they are only communicating short bursts of simple data, and "sleeping" the rest of the time.

Both classic Bluetooth and BTLE operate on the 2.4-2.5 GHz ISM radio bands that you learned about in the previous chapter. Bluetooth chips can implement both Bluetooth Classic and BTLE, or just BTLE. Your smartphone supports both, allowing it to communicate with wireless headphones that use classic Bluetooth audio profiles, and simpler devices using BTLE only, like heart rate monitors. The Bluetooth chip that you'll use with the Arduino in this chapter implements BTLE only, using a popular chipset from Nordic Semiconductor.

Bluetooth Profiles and BTLE GATT Services

To help you understand the purpose of Bluetooth profiles, I'll draw parallels to the way you use the internet. Your computer connects to the internet, physically, through your internet service provider (ISP). Between their network and your computer is a modem, a router (in most cases), and either a wired Ethernet or a wireless Wi-Fi link between your computer and that router. That setup defines your physical connection to the internet—how you transfer data over that connection depends on what you're doing.

Your computer will use services, systems, and protocols that ride on top of that network to accomplish various tasks. For instance, DNS enables your computer to associate IP addresses with URLs, HTTPS allows you to securely load web pages in your browser, FTP allows you to download files from a remote server, and SMTP allows you to send emails. Bluetooth profiles are like these web services—they are different means of communicating over the same physical link. As long as both parties understand the same profile, they can talk to each other using that profile.

Which profiles are supported will depend on the device. For instance, Bluetooth headphones will likely support A2DP (Advanced Audio Distribution Profile) for streaming music, but probably won't support FTP (File Transfer Protocol). When you connect two Bluetooth devices together, they tell each other what profiles they support. That's why your phone knows that it can play music over your headphones, while also recognizing that it can't play music to your Bluetooth-enabled doorbell. BTLE

specifically focuses on supporting a particular profile called GATT (Generic Attribute). GATT is used to define a data structure for passing information between two BTLE-capable devices. While a profile like A2DP may be focused on streaming audio in one direction, GATT services focus on transmitting specifically formatted chunks of data. They are optimized for short, brief bursts of information.

There exists a pre-defined list of GATT services assigned by the Bluetooth SIG, or you are free to create your own GATT. Some examples of predefined GATT services include a blood-pressure monitoring service, a battery monitoring service, and an environmental monitoring service.

Nordic Semiconductor, the creator of the BTLE radio that you'll use in this chapter, has implemented a simple UART emulation GATT service that runs on top of BTLE. You'll use this pre-existing GATT service to transmit simple data between your phone and your Arduino. This will mimic some of the serial communication projects that you've already encountered in this book.

Communication between Your Arduino and Your Phone

With an understanding of Bluetooth under your belt, you are ready to establish a link between your BTLE-enabled Arduino and your smartphone. This chapter utilizes the Feather 32U4 Bluefruit LE board from Adafruit. There are a lot of Arduino-compatible BTLE devices on the market, but Adafruit provides some of the best libraries and tutorial support. You'll recognize the "32U4" moniker from when you first encountered the Arduino Leonardo.

The 32U4 is the same Atmel/Microchip microcontroller that is used on the Leonardo, and this board is largely hardware-compatible with the Leonardo. However, there is one key difference: the Feather board operates at 3.3V logic levels. That means that logic HIGH is 3.3V, not 5V on this board. If you want to connect it to 5V sensors or peripherals, you'll need to level-shift them, or use 3.3V peripherals instead. The Feather board is basically just a 32U4-based Arduino smashed together with a BTLE module on a single board—it's effectively no different from connecting a BTLE shield to an Arduino, but it achieves greater compactness by just including both of these elements on one board.

Reading a Sensor over BTLE

To get started, you'll use the simplest sensor of all, a potentiometer. Hook it up to your Feather board, connecting its outer two pins to 3.3V and GND, and the middle pin to A0, as shown in Figure 16-1. You'll program the Arduino to broadcast the value of this potentiometer over BTLE.

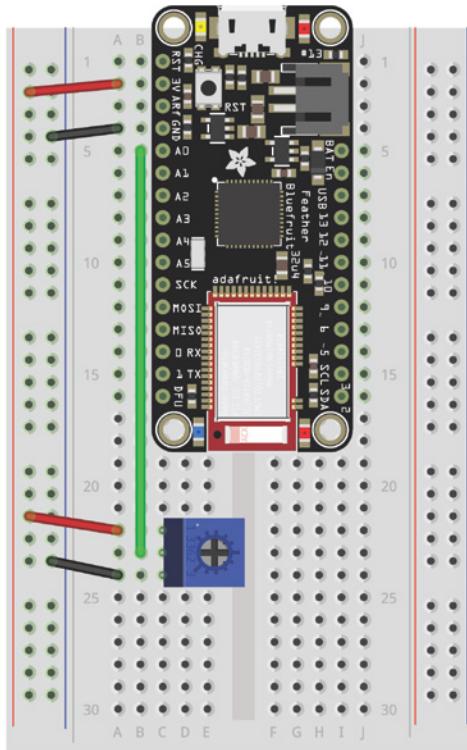


Figure 16-1: Feather BTLE board with potentiometer

Created with Fritzing

Adding Support for Third-Party Boards to the Arduino IDE

You've already learned how to add new libraries to the Arduino IDE. But, this is the first example in this book where you are using hardware that is neither explicitly designed by Arduino, nor a directly compatible clone of a first-party Arduino board. The Feather boards from Adafruit are "Arduino-compatible," meaning they are designed to be programmable with the Arduino IDE, use Arduino libraries, and generally work with any hardware that will work with a first-party Arduino board. You'll need to add support for this board to your Arduino IDE so that the IDE is able to compile code for it.

This process used to be far more difficult, but recent versions of the Arduino IDE have made it much easier to automatically add all the necessary files to support third-party hardware. In the Arduino IDE, go to File > Preferences. In the window that pops up, you see a text field at the bottom that says Additional Boards Manager URLs. In that text box, add this URL: <https://adafruit.github.io/arduino-board-index/>

package_adafruit_index.json. It should look like Figure 16-2 (note that the beginning of the URL is cutoff in the screenshot because of the length of the text box - you must paste this whole URL in).

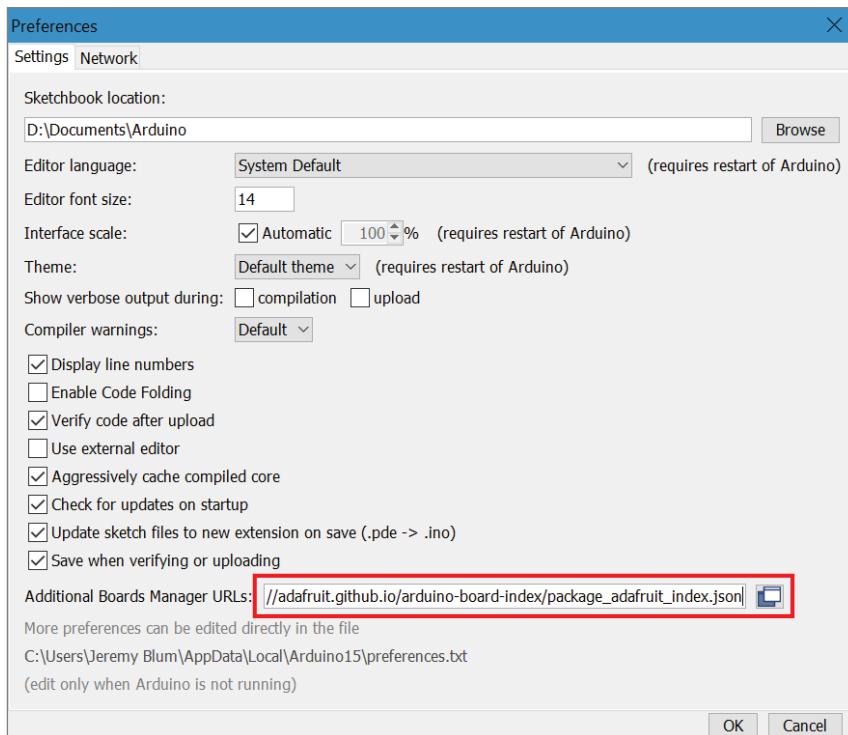


Figure 16-2: Arduino IDE Preferences window with added board URL

Click OK. This tells the Arduino IDE to query that URL and include the list of boards from that URL in the list of boards that you are able to install support for. Next, go to Tools > Board: “XXX” > Boards Manager (replacing “XXX” with whatever your currently selected board is). The Boards Manager window pops up. Change the Type drop-down menu to Contributed and type AVR into the search box. The result should look like Figure 16-3.

Click Install on the only entry. Then close that window and close the Arduino IDE.

Finally, you may need to install drivers for the Feather board if you are on Windows (you can skip this step if you are using a different operating system). Adafruit provides a convenient installer. Download the .exe file from blum.fyi/adafruit-windows-drivers. Follow the steps to complete the installation (you can install all the options). Then launch the Arduino IDE again. Once the IDE has opened, you should be able to navigate to Tools > Board and select Adafruit Feather 32U4 from the options.

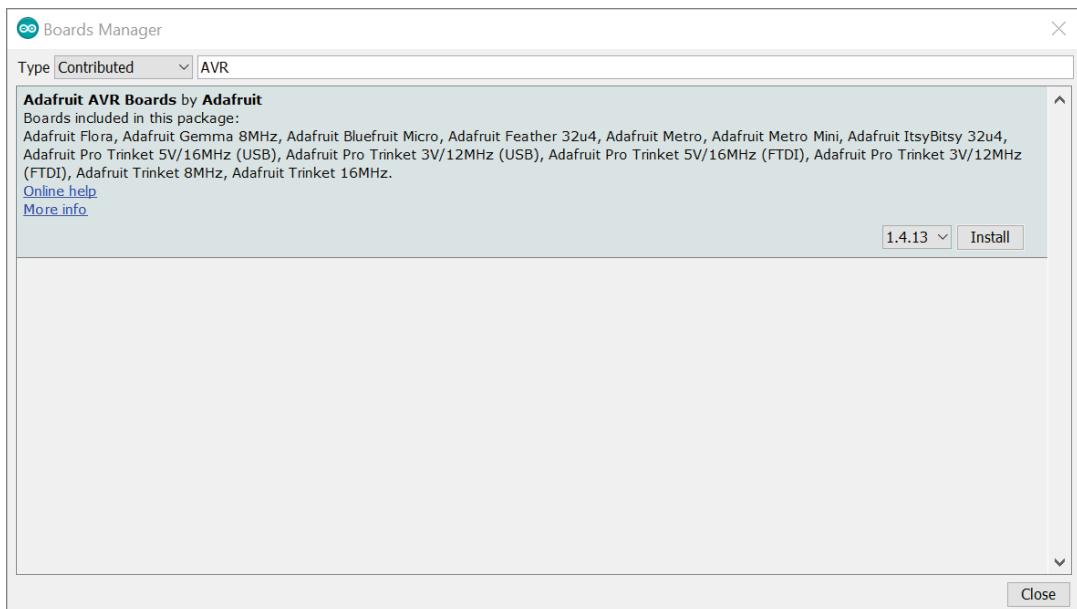


Figure 16-3: Arduino IDE Boards Manager

Installing the BTLE Module Library

Now you are ready to program your Feather board. Plug it into your computer via USB and confirm that it shows up under Tools > Port. Select it. The Feather board is basically two microcontrollers that are on one circuit board. The Nordic BTLE module (specifically an nRF51, called Bluefruit by Adafruit) is responsible for running the BTLE interface. It already contains its own firmware running its various BTLE services (including the serial emulation GATT service that you'll use momentarily). This BTLE module is connected to the 32U4 (effectively the “Arduino part” of this board) via an SPI interface. You'll use Adafruit-provided libraries to send commands back and forth between the 32U4 and the BTLE module.

Install the Adafruit nRF51 library using the process that you first learned about in Chapter 11, “The SPI Bus and Third-Party Libraries.” Navigate to Sketch > Include Library > Manage Libraries. Search for Adafruit_nRF51 and install the Adafruit BluefruitLE nRF51 library. Once it's installed, you can utilize its functions to easily talk to the nRF51 BTLE module that is already mounted to your Feather board.

Programming the Feather Board

As you program the Feather board, think about it as two discrete elements: the Arduino, and the BTLE module. You are only responsible for programming the Arduino's software. The BTLE module is already running firmware that is waiting to receive

commands and data via SPI from the connected Arduino. The objectives of your first Arduino program on the Feather board are as follows:

1. Connect to the BTLE module.
2. Confirm that the BTLE module is correctly receiving your commands.
3. Reset the BTLE module to its defaults and assign a custom broadcast name to the BTLE module (this is what you will see on your phone when you connect to it).
4. Wait for a smartphone to connect.
5. Once a phone has connected, transmit the value of the potentiometer to the phone as quickly as possible.

To achieve the first objective, connecting to the module, you need to import the Adafruit BTLE SPI library and create a BTLE object. In the case of the 32U4 Feather board, the BTLE module is connected to the Arduino's SPI pins, and its Chips Select, Interrupt, and Reset pins are respectively connected to pins 8, 7, and 4 from the 32U4. You must pass these pin numbers as arguments to the BTLE object upon its initialization. These pins are not hard-coded into the library code because the same library can be used for a standalone BTLE breakout module that Adafruit also sells. Because these pins are changeable parameters, users who may connect the BTLE module to different pins on a stand-alone Arduino can use this library without needing to modify its inner functionality. The import and object creation are done as follows:

```
// Include the nRF51 SPI Library
#include "Adafruit_BluefruitLE_SPI.h"

// On the 32U4 Feather board, the nRF51 chip is connected to the
// 32U4 hardware SPI pins. When we create the BTLE object, we tell
// it what pins are used for CS (8), IRQ (7), and RST (4):
Adafruit_BluefruitLE_SPI btle(8, 7, 4);
```

The second objective is to connect to the BTLE module and confirm that it is responding to commands. The BTLE library has a built-in function called `begin()` that sets up the module to receive commands over the SPI interface. Because you created an `Adafruit_BluefruitLE_SPI` object called `btle`, you call the `begin()` function on that object in your `setup()` function:

```
btle.begin();
```

The `begin()` function returns a Boolean representing whether the Arduino was able to successfully initialize the BTLE module. To take advantage of that, you can wrap it in an `if()` statement and halt the program execution if the module couldn't be initialized. Entering the wrong pin numbers during the object creation step is one reason why it may not initialize properly. The Arduino has no explicit function that

stops further code execution, so if you want to permanently halt a program, you can do so by putting it into an endless loop with `while(1);` as follows:

```
// Connect to the module.
Serial.print(F("Initializing BTLE Module..."));
if (!btle.begin())
{
    Serial.println("");
    Serial.println(F("Couldn't connect to nRF51 Module."));
    while(1);
}
Serial.println(F("Ready!"));
```

This code snippet assumes that you have already initialized the serial interface for printing debug messages to the attached computer. If the `begin()` function returns `false`, then a message is printed to the serial console and the program halts.

The third objective is to reset the BTLE module to its defaults, and assign it a new name. You don't actually have to do this each time—when you assign it a new name and restart the module, it records that name to non-volatile memory (meaning it is maintained across power cycles). However, there's no harm in setting it on each launch of the program. The same applies to the factory reset. It's really only necessary if you think you've managed to push a configuration change to the module that has caused it to stop working properly. Still, doing it on each run guarantees that the system always starts in the same state, which is useful for debugging the program if you choose to add more sophisticated features later. There is a built-in function to perform a factory reset of the module:

```
btle.factoryReset()
```

Setting the device name is a bit different; you can do this using an AT command. “AT” is short for *Attention* and it describes special commands sent to a modem device (a Bluetooth modem in this case) that should be interpreted differently from normal incoming data. Data received by a modem device is all sent over the same communication interface (SPI on the Feather board). By prefixing data that should be used to change internal settings on the Bluetooth modem with AT, you are telling the modem to interpret the remainder of that line as command instructions (instead of data to be transmitted to the remote device). The AT command string to change the Bluetooth modem’s broadcast name is `GAPDEVNAME`. So, to send this information to the modem, you simply do the following:

```
btle.print("AT+GAPDEVNAME=");
btle.println("Jeremy's Sensor");
```

The first line is the AT command. The second line is the parameter for the AT command. In this case, I'm naming the device “Jeremy's Sensor.” As you did with the

module initialization, you can execute both the reset and naming routines inside `if()` statements to ensure they were received and interpreted properly by the Bluetooth module:

```
// Reset to defaults
Serial.print(F("Resetting to Defaults..."));
if (!btle.factoryReset())
{
    Serial.println("");
    Serial.println(F("Couldn't reset module."));
    while(1);
}
Serial.println(F("Done!"));

// Set the name to be broadcast using an AT Command
Serial.print(F("Setting Device name..."));
btle.print("AT+GAPDEVNAME=");
btle.println("Jeremy's Sensor");
if (!btle.waitForOK())
{
    Serial.println(F("Could not set name."));
    while(1);
}
btle.reset(); // Restart the module for new name to take effect
Serial.println(F("Done!"));
```

Recall that you can wrap unchanging (static) strings in `F()` to save them to flash memory instead of RAM on the microcontroller. The `waitForOK()` function returns `true` only if the Bluetooth module acknowledges that it received the command and it was valid. If you have mistyped `GAPDEVNAME` as `GAPDIVNAME`, for example, it does not recognize the command and `waitForOK()` returns `false`. After setting the name, issuing the `reset()` command reboots the Bluetooth module, saving the new name to nonvolatile memory.

Now that the configuration of the module is complete, you can write the code that will go in the `loop()`. Each time through the loop, the software should confirm that a smartphone is connected to the module, take a reading from the analog sensor, and then transmit that reading to the connected phone. If a phone is not connected to the module, then the sketch should pause its readings and wait for a phone to reconnect before it continues looping. To check if the Bluetooth module is connected to a phone, use the `isConnected()` function. If that returns `false`, then enter a waiting loop. If it returns `true`, then the loop may proceed:

```
// Wait for a smartphone to connect if it isn't already
if (!btle.isConnected())
{
    Serial.print("Waiting to connect to a smartphone...");
    while (!btle.isConnected())
    {
        delay(1000);
```

```

        }
        Serial.println("Connected! ");
    }
}

```

The final part of the program is to transmit the data. Read the analog sensor as you've done many times before. Then, simply print that reading to the `btle` object instead of the `Serial` object to send the reading to the Bluetooth module. After each line that you send to the Bluetooth module, you can use the `waitForOK()` function that you used with the AT command to confirm that the data was received and transmitted:

```

// Get the Value of the Potentiometer
int val = analogRead(POT);

// Send the data to the BTLE module to be sent via BTLE Serial
btle.println(val);

// Wait for the BTLE module to acknowledge it received the data
btle.waitForOK();

```

Now you just need to put the above code snippets together to have a functional program that configures the Bluetooth module and starts sending the analog sensor readings. Putting it all together with some added serial debugging and LED blinking, you get something like Listing 16-1.

Listing 16-1

Transmitting sensor data over a BTLE link—BTLE_sensor

```

// Send sensor data over BTLE

// Include the nRF51 SPI Library
#include "Adafruit_BluefruitLE_SPI.h"

// On the 32U4 Feather board, the nRF51 chip is connected to the
// 32U4 hardware SPI pins. When we create the BTLE object, we tell
// it what pins are used for CS (8), IRQ (7), and RST (4):
Adafruit_BluefruitLE_SPI btle(8, 7, 4);

// Set this to true for one time configuration
// This performs a factory reset, then changes the broadcast name.
// There is no harm in redoing this at each boot (leave true).
// You can set this to false after you have programmed the module one time.
const bool PERFORM_CONFIGURATION = true;

// This is how the BTLE device will identify itself to your smartphone
const String BTLE_NAME = "Jeremy's Sensor";

```

```
// Potentiometer is connected to pin A0
const int POT = A0;

// On-Board LED is connected to Pin 13
const int STATUS_LED = 13;

void setup(void)
{
    // Set LED as output
    pinMode(STATUS_LED, OUTPUT);

    // We'll print debug info to the Serial console.
    Serial.begin(9600);

    // The 32U4 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
    // want the sketch to run without opening the serial console (or on battery).
    while(!Serial);

    // Connect to the module.
    Serial.print(F("Initializing BTLE Module..."));
    if (!btle.begin())
    {
        Serial.println("");
        Serial.println(F("Couldn't connect to nRF51 Module."));
        while(1);
    }
    Serial.println(F("Ready!"));

    // Reset the BTLE chip to factory defaults if specified.
    // You can trigger this to recover from any programming errors you
    // make that render the module unresponsive.
    // After doing factory reset of the module, it sets its broadcast name
    if (PERFORM_CONFIGURATION)
    {
        // Reset to defaults
        Serial.print(F("Resetting to Defaults..."));
        if (!btle.factoryReset())
        {
            Serial.println("");
            Serial.println(F("Couldn't reset module."));
            while(1);
        }
        Serial.println(F("Done!"));

        // Set the name to be broadcast using an AT Command
        Serial.print(F("Setting Device name..."));
        btle.print(F("AT+GAPDEVNAME="));
        btle.println(BTLE_NAME);
        if (!btle.waitForOK())

```

```
{  
    Serial.println(F("Could not set name."));  
    while(1);  
}  
btle.reset(); // Restart the module for new name to take effect  
Serial.println(F("Done!"));  
}  
  
//Switch to Data mode (from command mode)  
btle.setMode(BLUETOOTH_MODE_DATA);  
}  
  
void loop(void)  
{  
    // Wait for a smartphone to connect if it isn't already  
    if (!btle.isConnected())  
    {  
        Serial.print("Waiting to connect to a smartphone...");  
        while (!btle.isConnected())  
        {  
            delay(1000);  
        }  
        Serial.println("Connected!");  
    }  
  
    // Get the Value of the Potentiometer  
    int val = analogRead(POT);  
  
    // Print the value to the attached serial number  
    Serial.print(F("Analog Value: "));  
    Serial.print(val);  
    Serial.print(F("\tSending..."));  
  
    // Send the data to the BTLE module to be sent via BTLE Serial  
    // Blink the LED when we do this.  
    digitalWrite(STATUS_LED, HIGH);  
    btle.println(val);  
  
    // Wait for the BTLE module to acknowledge it received the data  
    btle.waitForOK();  
    Serial.println(F("OK!"));  
    digitalWrite(STATUS_LED, LOW);  
}
```

This program does everything that you've built over the last several pages. It adds a bit of serial debugging, and sets up a configuration variable so you can choose if you want to factory reset each time. Load this onto your Feather 32U4 Bluefruit LE. Because of the 32U4's native USB interface, you need to include the `while(!Serial)` line to prevent the sketch from starting until you open the serial monitor to watch its

output. The exact way the Feather board behaves with that line will depend on your computer. See the sidebar: “The 32U4’s USB Interface”.

THE 32U4’S USB INTERFACE

As you learned in Chapter 8, “Emulating USB Devices,” the ATmega 32U4 uses a direct USB interface instead of utilizing a separate USB-to-Serial chip like the ATmega 328P that is used on the Arduino Uno. For this reason, outgoing data is handled in a slightly different way. On the Uno, if you include `Serial.println()` statements and don’t open the serial monitor, the program happily continues, sending those print statements out into the void. The 32U4, however, may hold those messages in a transmit buffer if you start the sketch with the serial port active and then disconnect it. Unfortunately, the exact behavior can vary between operating systems. If you have issues with the sketch seeming to hang, you can either open the Arduino IDE serial monitor before starting the sketch (and keep it open), or try using the Feather board with a different computer or operating system, or both. If you don’t care about the serial data being printed out, then you can remove the `while(!Serial)`, `Serial.begin()`, `Serial.print()`, and `Serial.println()` commands. You may want to do that if you are operating the Feather board off a battery pack.

After you load the sketch onto the Feather board, open the serial monitor; you should see the window in Figure 16-4. (It is waiting to start its execution until you open the serial monitor because of the `while(!Serial)` line.)

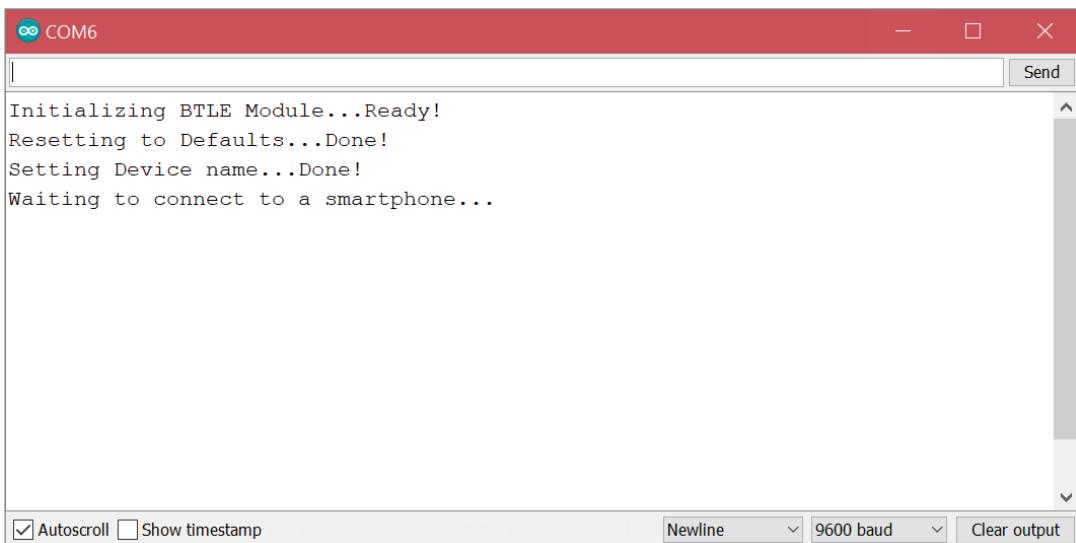


Figure 16-4: Serial monitor after module initialization

Connecting Your Smartphone to Your BTLE Transmitter

Adafruit has developed an app for Android and iPhone that makes it easy to perform basic tasks with your BTLE module. Writing your own mobile app is beyond the scope of this book, but Adafruit's apps are open-source and available on their GitHub profile if you'd like to see how they work. You can find the code at bluem.fyi/bluefruit-app-code.

Search the iPhone App Store or the Android Play Store for Adafruit Bluefruit LE Connect and install it on your smartphone. When you open the app for the first time, you may be prompted to give it a location and Bluetooth permissions. Agree to give it the permissions it requires, and you'll arrive at the main screen of the application, as shown in Figure 16-5.

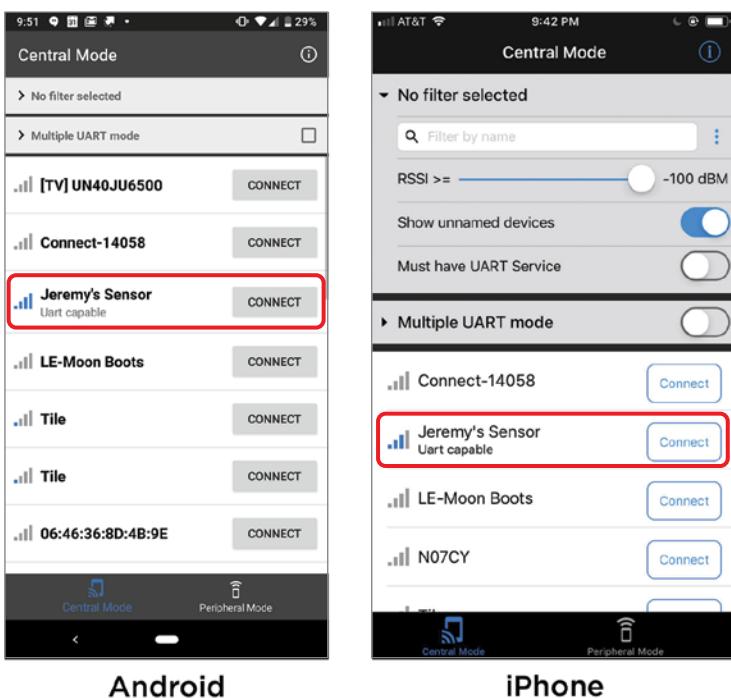


Figure 16-5: Adafruit Bluefruit LE Connect app on iPhone and Android

If your Feather board has been properly programmed and is showing “Waiting to connect to a smartphone ...” in the Arduino IDE serial monitor, then your device should show up with the custom name you’ve assigned it in the listing of BTLE devices. In

Figure 16-5, you can see that both the iPhone and Android devices have properly detected the BTLE module. Tap the Connect button next to your device. The application checks the firmware running on the module and may ask to upgrade it if a newer version is available, as shown in Figure 16-6.

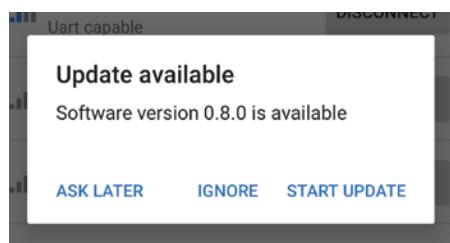


Figure 16-6: BTLE module firmware update alert

Start the update if prompted. This updates the firmware running on the BTLE module. Once that is complete, you may need to initiate the connection again. As soon as you connect, the blue LED on the Feather board illuminates, indicating that the BTLE module is connected to your smartphone. The red onboard LED should also illuminate to indicate that data transfer has begun, and your serial monitor should start to update, showing a stream of data from your Arduino to your phone, as shown in Figure 16-7.

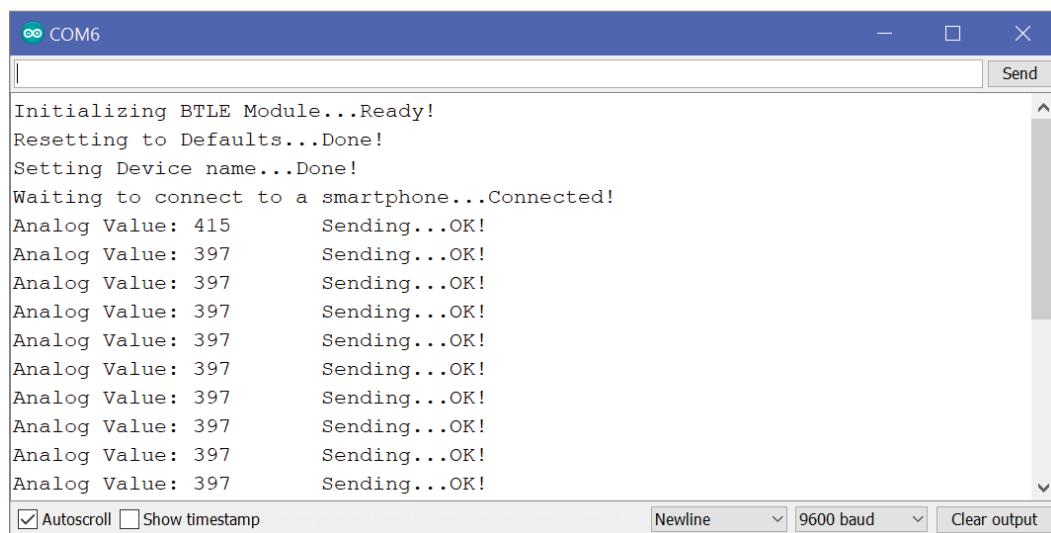


Figure 16-7: Serial monitor showing a data stream

The app has a built-in function for visualizing the data that you are sending to your phone. Click the Plotter button to see a live graph of the data. Turn the potentiometer while viewing the graph to see a live stream of data on the graph as you turn it, as shown in Figure 16-8.

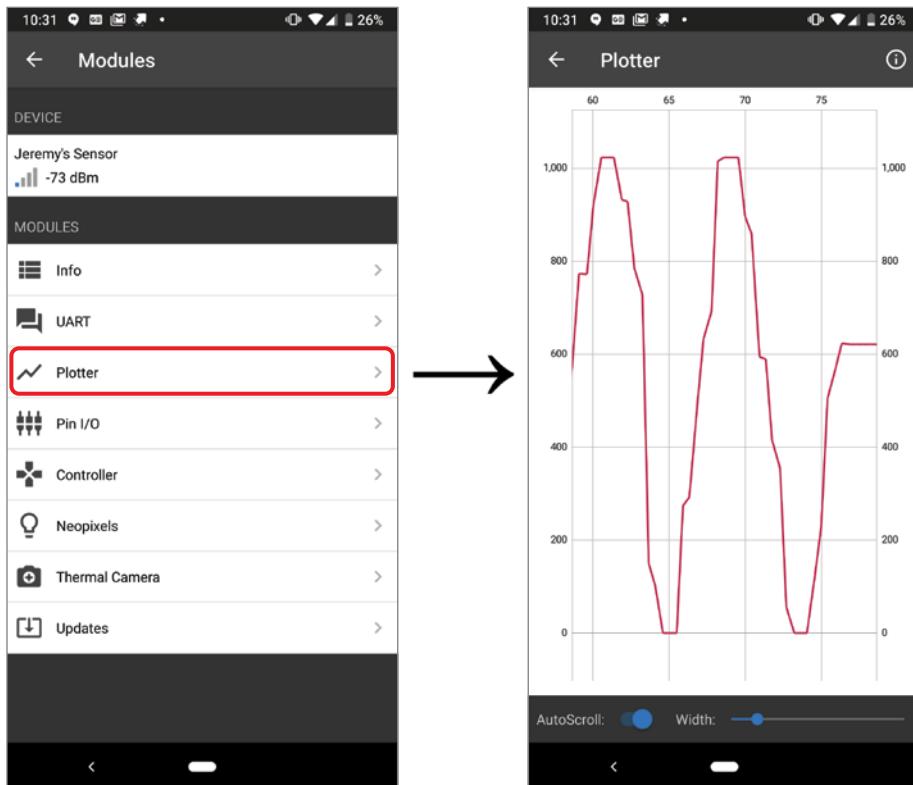


Figure 16-8: Using the plotting functionality

NOTE Watch a demo video of this BTLE sensor visualization at exploringarduino.com/content2/ch16.

Try connecting different analog sensors to pin A0. What happens when you connect a distance sensor? Or a photoresistor?

Sending Commands from Your Phone over BTLE

Now that you've learned how to transmit a data stream from your Arduino to your phone, what about the opposite direction? Can you use your phone to send

commands to your Arduino? Of course you can! Add an LED to your breadboard connected to pin 5 of the Arduino through a current-limiting resistor (220Ω or 150Ω will work fine), as shown in Figure 16-9. You can leave the potentiometer connected if you want.

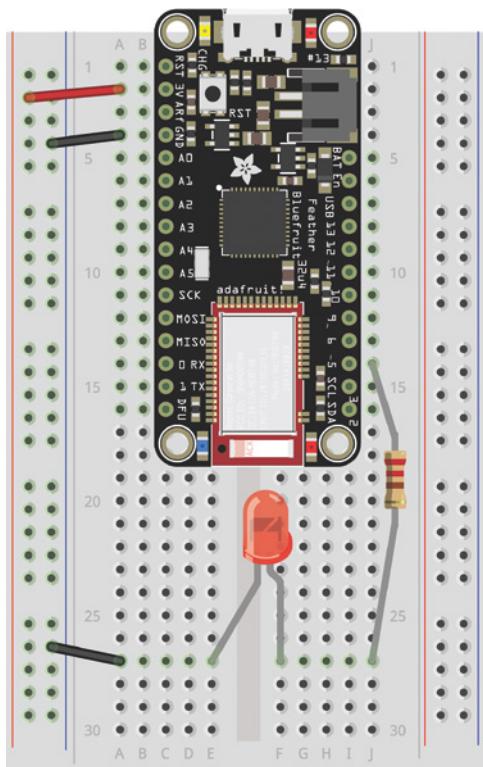


Figure 16-9: Feather BTLE board with LED

Created with Fritzing

Parsing Command Strings

The Bluefruit app provided by Adafruit also has a UART mode that allows you to easily send text commands to a connected BTLE module. You'll use that mode to transmit “natural language” commands that will turn a red LED on and off and query its status. By checking for some keywords in the incoming text strings on the Arduino, you can perform a rudimentary form of natural language processing.

Sophisticated natural language processing, similar to that done by the voice assistant app on your smartphone, leverages machine learning and complex language models that are beyond the scope of this book. By simply looking for certain keywords and not requiring exact command strings, you'll be able to send requests like "Turn on the LED" and "Turn off the LED now" to get your Arduino to do what you want.

In each pass through the main `loop()` of the program, you can use `btle.available()` to check if incoming data is available. If it is, `cmd = btle.readStringUntil('\n');` can then be called to read the entirety of the incoming string into the `cmd` variable. Transmissions from the Adafruit Bluefruit app automatically append a newline character (`\n`) to the end so that you can easily grab one command string at a time. With the incoming string stored, you can proceed with parsing. Start by converting the entire string to lowercase using `cmd.toLowerCase();`. This ensures that when you parse the string, you can just check for words that match in lowercase. If you send "Turn the LED on" and "Turn the led on," both are interpreted the same way.

```
// If there is incoming data available, read and parse it
while (btle.available() > 0)
{
    // Read the receive buffer until there is a newline
    cmd = btle.readStringUntil('\n');

    // Makes it lower case so we recognize the command regardless of caps
    cmd.toLowerCase();

    // PUT PARSING AND LED CONTROL CODE HERE

}
```

With the string received, you just need to parse it to know what action to take. You'll consider a few keywords. If the keyword "red" or "led" is present in the command string, then parsing proceeds to determine what to do with the LED. The code then looks for "on", "off", or "toggle". If one of those words is detected, then it takes the corresponding action on the LED and reports back the LED state. If none of those words are detected in the command string, then it just reports back the current state of the LED. If neither "red" nor "led" is present, then the BTLE module should respond that it doesn't know what to do. Consider the example strings and their corresponding actions in Table 16-1.

Table 16-1: Examples of Natural Language Commands

Command	Action	Why?
"Red light on"	Red LED turns on	<i>Red</i> keyword enters parsing. <i>On</i> keyword turns LED on.
"Toggle the LED"	Red LED changes state	<i>LED</i> keyword enters parsing. <i>Toggle</i> keyword toggles LED.
"Turn on that LED!"	Red LED turns on	<i>LED</i> keyword enters parsing. <i>On</i> keyword turns on LED.
"What's the LED state"	Reports the current LED state	<i>LED</i> keyword enters parsing. Lack of any other keyword defaults software to reporting state.
"What's up?"	Reports unknown command	Neither the <i>LED</i> nor <i>Red</i> keyword is present, so no further parsing occurs.
"Turn the light off"	Reports unknown command	Neither the <i>LED</i> nor <i>Red</i> keyword is present, so no further parsing occurs, even though the <i>off</i> keyword is present.

With that in mind, you can fill in the parsing section in the loop that receives the data:

```
// If there is incoming data available, read and parse it
while (btle.available() > 0)
{
    // Read the receive buffer until there is a newline
    cmd = btle.readStringUntil('\n');

    // Makes it lower case so we recognize the command regardless of caps
    cmd.toLowerCase();

    // Parse commands with the word "red" or "led"
    if (cmd.indexOf(F("red")) != -1 || cmd.indexOf(F("led")) != -1)
    {
        // Command contains "on"
        if (cmd.indexOf(F("on")) != -1)
        {
            led_state = HIGH;
            reply = F("OK! The LED has been turned on.");
        }

        // Command contains "off"
        else if (cmd.indexOf(F("off")) != -1)
        {
            led_state = LOW;
            reply = F("OK! The LED has been turned off.");
        }
    }
}
```

```

// Command contains "toggle"
else if (cmd.indexOf(F("toggle")) != -1)
{
    led_state = !led_state;
    if (led_state) reply = F("OK! The LED has been toggled on.");
    else reply = F("OK! The LED has been toggled off.");
}

// Command contained "red" or "led", but none of the other keywords
else
{
    if (led_state) reply = F("The LED is currently on.");
    else reply = F("The LED is currently off.");
}

// Set the LED state
digitalWrite(CTRL_LED, led_state);
}
else
{
    reply = F("Command not understood.");
}
}

```

This code snippet assumes that `reply` and `led_state` are global variables initialized at the top of the sketch. The crux of this code snippet is the `indexOf()` function, which acts on a string (`cmd` in this case), and returns the index of a particular string being searched for inside the string being searched. In a string, the index is the position of a particular character, with counting starting at 0. So, in the string "Hello Jeremy!", `indexOf("Jeremy")` would return 6, because the J in Jeremy is located at position 6 (with the H in Hello being at position 0). If the string being searched for does not occur anywhere in the string being searched, then the function returns a -1. Therefore, by confirming that the `indexOf()` function does not return a -1, you can be certain that the search string is present in the string being searched. Try to trace one of the example strings through the code snippet.

After this code sets the LED state and populates the `reply` variable, you simply need to communicate the `reply` back to your smartphone:

```

// Acknowledge Command
btle.println(reply);
Serial.print(F("Replied With: "));
Serial.println(reply);
btle.waitForOK();
digitalWrite(STATUS_LED, LOW);

```

Commanding Your BTLE Device with Natural Language

With the parsing completed, add the same setup and connection functionality that you used in Listing 16-1, and add the appropriate state variables to the top of the program so it can track the LED state and BTLE replies. You may also want to change the broadcast name of your device, to better communicate its new function. You should end up with something similar to Listing 16-2.

TIP The size of the user data buffer in the UART transmit service of the Nordic BTLE chip is only 20 bytes. That means that any strings over 20 characters in length will be truncated. If you see truncated strings in your serial monitor, then try using shorter command sentences. All the examples listed in Table 16-1 are 20 characters or less.

Listing 16-2

Controlling an LED with BTLE-BTLE_led

```
// Control an LED over BTLE

// Include the nRF51 SPI Library

#include "Adafruit_BluefruitLE_SPI.h"

// On the 32U4 Feather board, the nRF51 chip is connected to the
// 32U4 hardware SPI pins. When we create the BTLE object, we tell
// it what pins are used for CS (8), IRQ (7), and RST (4):
Adafruit_BluefruitLE_SPI btle(8, 7, 4);

// Set this to true for one time configuration
// This performs a factory reset, then changes the broadcast name.
// There is no harm in redoing this at each boot (leave true).
// You can set this to false after you have programmed the module one time.
const bool PERFORM_CONFIGURATION = true;

// This is how the BTLE device will identify itself to your smartphone
const String BTLE_NAME = "Jeremy's LED";

// On-Board LED is connected to Pin 13
const int STATUS_LED = 13;

// LED to be controlled is connected to Pin 5
const int CTRL_LED = 5;
```

```
// Variables to keep track of LED state
bool led_state = LOW;
String cmd = "";
String reply = "";

void setup(void)
{
    // Set LEDs as outputs and turn off
    pinMode(STATUS_LED, OUTPUT);
    digitalWrite(STATUS_LED, LOW);
    pinMode(CTRL_LED, OUTPUT);
    digitalWrite(CTRL_LED, led_state);

    // We'll print debug info to the Serial console.
    Serial.begin(9600);

    // The 32U4 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
    // want the sketch to run without opening the serial console (or on battery).
    while(!Serial);

    // Connect to the module.
    Serial.print(F("Initializing BTLE Module..."));
    if (!btle.begin())
    {
        Serial.println("");
        Serial.println(F("Couldn't connect to nRF51 Module."));
        while(1);
    }
    Serial.println(F("Ready!"));

    // Reset the BTLE chip to factory defaults if specified.
    // You can trigger this to recover from any programming errors you
    // make that render the module unresponsive.
    // After doing factory reset of the module, it sets its broadcast name
    if (PERFORM_CONFIGURATION)
    {
        // Reset to defaults
        Serial.print(F("Resetting to Defaults..."));
        if (!btle.factoryReset())
        {
            Serial.println("");
            Serial.println(F("Couldn't reset module."));
            while(1);
        }
        Serial.println(F("Done!"));
    }
}
```

```
// Set the name to be broadcast using an AT Command
Serial.print(F("Setting Device name..."));
btle.print(F("AT+GAPDEVNAME="));
btle.println(BTLE_NAME);
if (!btle.waitForOK())
{
    Serial.println(F("Could not set name."));
    while(1);
}
btle.reset(); // Restart the module for new name to take effect
Serial.println(F("Done!"));

}

//Switch to Data mode (from command mode)
btle.setMode(BLUEFRUIT_MODE_DATA);

}

void loop(void)
{
    // Wait for a smartphone to connect if it isn't already
    if (!btle.isConnected())
    {
        Serial.print("Waiting to connect to a smartphone...");
        while (!btle.isConnected())
        {
            delay(1000);
        }
        Serial.println("Connected!");
    }

    // If there is incoming data available, read and parse it
    while (btle.available() > 0)
    {
        // Blink the Status LED when we receive a request
        digitalWrite STATUS_LED, HIGH);

        // Read the receive buffer until there is a newline
        cmd = btle.readStringUntil('\n');
        Serial.print(F("Received Command: "));
        Serial.println(cmd);

        // Makes it lower case so we recognize the command regardless of caps
        cmd.toLowerCase();

        // Parse commands with the word "red" or "led"
        if (cmd.indexOf(F("red")) != -1 || cmd.indexOf(F("led")) != -1)
        {
```

```
// Command contains "on"
if (cmd.indexOf(F("on")) != -1)
{
    led_state = HIGH;
    reply = F("OK! The LED has been turned on.");
}

// Command contains "off"
else if (cmd.indexOf(F("off")) != -1)
{
    led_state = LOW;
    reply = F("OK! The LED has been turned off.");
}

// Command contains "toggle"
else if (cmd.indexOf(F("toggle")) != -1)
{
    led_state = !led_state;
    if (led_state) reply = F("OK! The LED has been toggled on.");
    else reply = F("OK! The LED has been toggled off.");
}

// Command contained "red" or "led", but none of the other keywords
else
{
    if (led_state) reply = F("The LED is currently on.");
    else reply = F("The LED is currently off.");
}

// Set the LED state
digitalWrite(CTRL_LED, led_state);
}
else
{
    reply = F("Command not understood.");
}

// Acknowledge Command
btle.println(reply);
Serial.print(F("Replied With: "));
Serial.println(reply);
btle.waitForOK();
digitalWrite(STATUS_LED, LOW);
}
```

Compile and flash this code onto your Feather board, and open the serial monitor. You should see the same window that you saw in Figure 16-4. Open the Adafruit

Bluefruit app on your phone, and connect to your newly named BTLE module. Confirm that the serial monitor shows that your phone has connected. Tap the UART option as shown in Figure 16-10, and you are brought to a chat-like interface where you can send and receive messages on your module. Type in commands similar to the ones shown in Table 16-1, and you should observe replies from your module. You should see the 5 mm red LED turn on and off when you command it to, and the red LED next to the USB connector on the Feather board should blink each time a command is received. You'll also see a listing of the commands and responses in the smartphone app's UART interface, as shown in Figure 16-10.

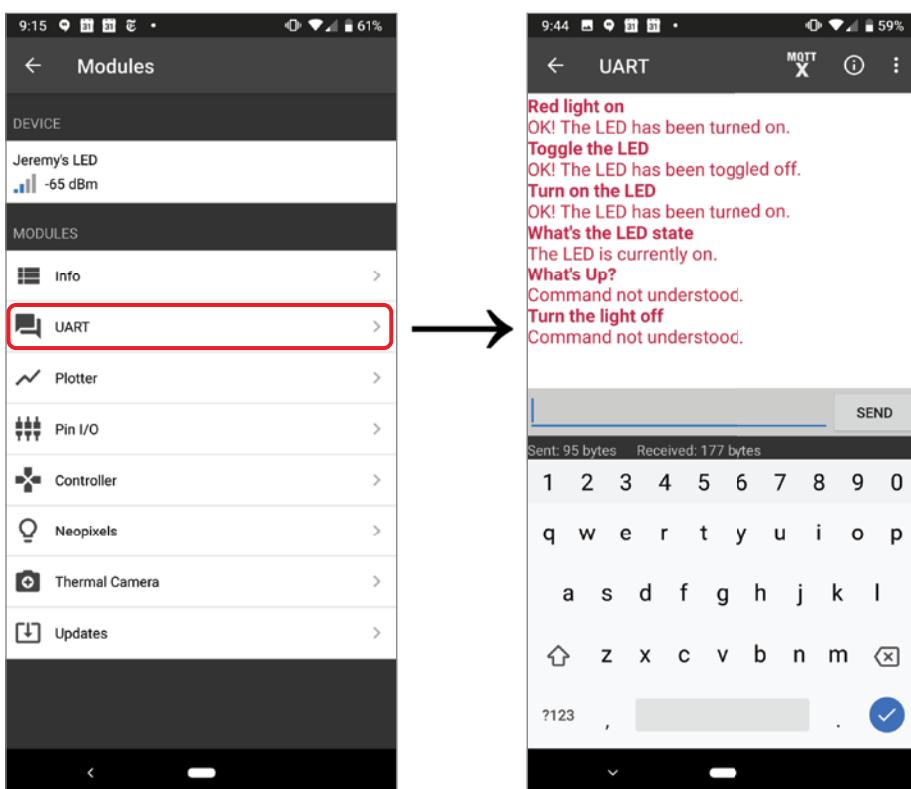


Figure 16-10: Sending commands over BTLE

Most smartphones allow you to use your voice to dictate to the keyboard. So without even writing any voice recognition software, you can turn this into a voice-controlled LED. Simply open the UART service in the Bluefruit app, and dictate your request to it!

NOTE Watch a demo video of this BTLE UART LED controller at exploring-arduino.com/content2/ch16.

Controlling an AC Lamp with Bluetooth

In the last chapter, you used an enclosed AC relay to control a lamp with your Arduino's RF link. Now, you can repeat that exercise using your new Bluetooth skills. Instead of using the UART app, though, you'll use a clever combination of AT commands and transmit-power trickery to build a lamp controller that will automatically turn your lamp on when you approach it, and turn it off when you leave.

How Your Phone "Pairs" to BTLE Devices

While sending chat-like messages to your Arduino is novel, it's not exactly the most convenient way of leveraging Bluetooth to control things. Imagine that your connected AC devices can be triggered automatically whenever you are in range, leveraging your smartphone as a proximity sensor. You can turn your light on while you're at home, trigger your nightlight when you go downstairs for a midnight snack, or turn off your TV whenever you leave the room. For this chapter's final project, you'll use the same relay box that you used in the last chapter to create a smart lamp that is only on when you are home.

Recall from earlier in the chapter that modern Bluetooth has two modes of operation: Low Energy and Classic. The module you are using is only a Bluetooth Low Energy device. The way smartphones handle these kinds of devices is a bit different. Bluetooth devices running classic profiles support "pairing." That's the process by which you tell your phone to remember a particular Bluetooth device, like your headphones. Whenever that device is powered and in range, your phone connects to it. BTLE devices act a bit differently by default. Each time a BTLE device connects to your phone, it provides a list of services that it supports. If both your phone and the BTLE device support the same service, then the BTLE device will "expose" an interface to that service so the phone can communicate with it. If the BTLE device isn't exposing a GATT service that requires pairing, then it may not be possible to pair your phone to it at the operating system level. iPhones, in particular, require a dedicated app, made by the BTLE device manufacturer, to handle the pairing process with a BTLE device if it doesn't expose certain services.

To make this setup process as simple as possible, however, there is a way to work around this restriction, so that you don't have to build a custom iPhone or Android app just so that your Arduino can detect your phone's proximity. iPhones and Androids

both enable options for pairing to BTLE devices if they are exposing an HID (Human Interface Device) profile. An example of an HID profile is a wireless keyboard. By enabling the HID Keyboard BTLE service, your Arduino announces itself as a wireless keyboard. Your phone's Bluetooth settings show this device and allow you to pair with it as you would with a classic Bluetooth device. Whenever your phone is within range of the paired device, it automatically connects to it—you don't even need to take your phone out of your pocket! You'll program the Arduino to wait for that connection and automatically switch the attached lamp on or off when a phone connects or disconnects from it.

Writing the Proximity Control Software

Before you actually hook your Arduino Feather board to a lamp, I suggest you get the basic functionality working with the circuit you've already been using: the single LED circuit shown in Figure 16-9. Once it's working, it's simple to connect the relay control wire to the same output pin (you can leave the red LED in place, too).

Working off the code you've already written, you want to make a few modifications. First, change the BTLE_NAME to one that is more fitting to its new function, such as "Smart Lamp." Next, you can optionally remove the PERFORM_CONFIGURATION constant and its affiliated if() statement. This circuit is rarely turned off, so it doesn't really matter if it reconfigures each time. Add a new constant called POWER_LEVEL:

```
// Set Power Level to change activation Range  
// Allowed options are: -40, -20, -16, -12, -8, -4, 0, 4  
// Higher numbers increase connection range  
const int POWER_LEVEL = -20;
```

This is used with a new AT command, AT+BLEPOWERLEVEL, to control the BTLE module's transmit power. If you pick a lower number, the connection range is smaller. If you pick a higher number, the connection range is larger. You probably want to test out a few different values to see what works best in your situation. The goal is to pick a number that roughly represents the turn-on distance for your lamp controller. In my home, setting the power to -20 is perfect for triggering a connection when my phone enters the front door of my apartment. If I wanted my lamp to turn on when I was on the staircase up to my apartment unit, then I might have picked a higher transmission power, like -12 or -16. After you set up the device name, use this new command to set the power level accordingly:

```
// Set the Power Level  
Serial.print(F("Setting Power level..."));
```

```
btle.print(F("AT+BLEPOWERLEVEL="));
btle.println(String(POWER_LEVEL));
if (!btle.waitForOK())
{
    Serial.println(F("Set Power Level."));
    while (1);
}
Serial.println(F("Done! "));
```

After the transmit power is configured, start the HID Keyboard service by using the BLEKEYBOARDEN AT command:

```
// Enable the HID Keyboard Profile
// (Necessary or iOS to Recognize it without app)
Serial.print(F("Enabling HID Keyboard..."));
btle.println(F("AT+BLEKEYBOARDEN=1"));
if (!btle.waitForOK())
{
    Serial.println(F("Could not enable HID Keyboard Profile."));
    while (1);
}
Serial.println(F("Done! "));
btle.reset(); // Restart the module for settings to take effect
```

With the transmission strength and keyboard profile configured in `setup()`, the `loop()` function just needs to set the control pin HIGH when a BTLE device is connected, and set the control pin LOW when a BTLE device is disconnected:

```
if (btle.isConnected())
{
    // Turn on the Lamp if connected
    digitalWrite(LAMP_PIN, HIGH);
}
if (!btle.isConnected())
{
    // Turn off the Lamp if disconnected
    digitalWrite(LAMP_PIN, LOW);
}
```

When your paired phone enters the connection range of your BTLE module, it automatically connects, triggering the LED to turn on. When your phone exits the transmission range of the BTLE module, your Arduino detects that it disconnected, and turns the attached LED off. Putting all the previous code snippets together, the functional software looks like Listing 16-3.

WARNING Do Not Leave Experimental, Unsecured Software Unattended! This software will control an AC appliance and may require debugging if you don't get the software working properly. Furthermore, this software does not implement security measures that may prevent a stranger from connecting to the Bluetooth device and controlling it. Never leave devices running experimental software unattended. Unplug this device when you aren't actively using it.

Listing 16-3

Controlling an LED with BTLE-BTLE_led

```
// Automatically turns on/off a lamp when a smartphone connects/disconnects

// Include the nRF51 SPI Library
#include "Adafruit_BluefruitLE_SPI.h"

// On the 32U4 Feather board, the nRF51 chip is connected to the
// 32U4 hardware SPI pins. When we create the BTLE object, we tell
// it what pins are used for CS (8), IRQ (7), and RST (4):
Adafruit_BluefruitLE_SPI btle(8, 7, 4);

// This is how the BTLE device will identify itself to your smartphone
const String BTLE_NAME = "Smart Lamp";

// Set Power Level to change activation Range
// Allowed options are: -40, -20, -16, -12, -8, -4, 0, 4
// Higher numbers increase connection range
const int POWER_LEVEL = -40;

// Lamp Control Pin
const int LAMP_PIN = 5;

void setup(void)
{
    // Set Lamp Control Pin as Output and turn off
    pinMode(LAMP_PIN, OUTPUT);
    digitalWrite(LAMP_PIN, LOW);

    // We'll print debug info to the Serial console.
    Serial.begin(9600);

    // The 32U4 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
```

```
// want the sketch to run without opening the serial console (or on battery).
//while (!Serial);

// Connect to the module.
Serial.print(F("Initializing BTLE Module..."));
if (!btle.begin())
{
    Serial.println("");
    Serial.println(F("Couldn't connect to nRF51 Module."));
    while (1);
}
Serial.println(F("Ready!"));

// Reset to defaults
Serial.print(F("Resetting to Defaults..."));
if (!btle.factoryReset())
{
    Serial.println("");
    Serial.println(F("Couldn't reset module."));
    while (1);
}
Serial.println(F("Done!"));

// Set the name to be broadcast using an AT Command
Serial.print(F("Setting Device name..."));
btle.print(F("AT+GAPDEVNAME="));
btle.println(BTLE_NAME);
if (!btle.waitForOK())
{
    Serial.println(F("Could not set name."));
    while (1);
}
Serial.println(F("Done!"));

// Set the Power Level
Serial.print(F("Setting Power level..."));
btle.print(F("AT+BLEPOWERLEVEL="));
btle.println(String(POWER_LEVEL));
if (!btle.waitForOK())
{
    Serial.println(F("Set Power Level."));
    while (1);
}
Serial.println(F("Done!"));

// Enable the HID Keyboard Profile
// (Necessary or iOS to Recognize it without app)
Serial.print(F("Enabling HID Keyboard..."));
```

```
btle.println(F("AT+BLEKEYBOARDEN=1"));
if (!btle.waitForOK())
{
    Serial.println(F("Could not enable HID Keyboard Profile."));
    while (1);
}
Serial.println(F("Done!"));

btle.reset(); // Restart the module for settings to take effect
}

void loop(void)
{
    if (btle.isConnected())
    {
        // Turn on the Lamp if connected
        digitalWrite(LAMP_PIN, HIGH);
    }
    if (!btle.isConnected())
    {
        // Turn off the Lamp if disconnected
        digitalWrite(LAMP_PIN, LOW);
    }
}
```

Note that I've commented out the `while (!Serial);` line in this sketch because you'll be using this code on an Arduino that will not be connected to a computer (once you connect the lamp relay). If you don't comment this line out, and the serial terminal isn't open, then your Arduino will just wait at that line forever!

Pairing Your Phone

The pairing process is similar for both iPhone and Android. Both operating systems change frequently, so these instructions are subject to change. Just do a web search for how to pair your phone with a Bluetooth device if these instructions don't work for you. Start by flashing Listing 16-3 onto your Arduino Feather board. You may want to have the serial monitor open to confirm that everything is working okay (uncomment the `while (!Serial);` line temporarily, or upload the code with the serial monitor already open).

Pairing an Android Phone

Open the Settings app on your phone, either by selecting it from the App Drawer, or by pressing the gear icon in the notification drop-down menu. Go to Connected Devices and tap Pair New Device. This turns on your Bluetooth radio if it isn't already. Under the list of available devices, your BTLE module should appear with the name you've

assigned it. Figure 16-11 shows this flow (the device is named Smart Lamp). Tap it. It should pair immediately, and you should see the red LED on your breadboard illuminate. The device should now show up under Currently Connected devices. If you toggle your phone's Bluetooth radio on and off manually from the notifications dropdown menu, you should see the LED turn on and off (your phone should automatically reconnect to your Arduino when it's in range). There will be a delay of a few seconds.

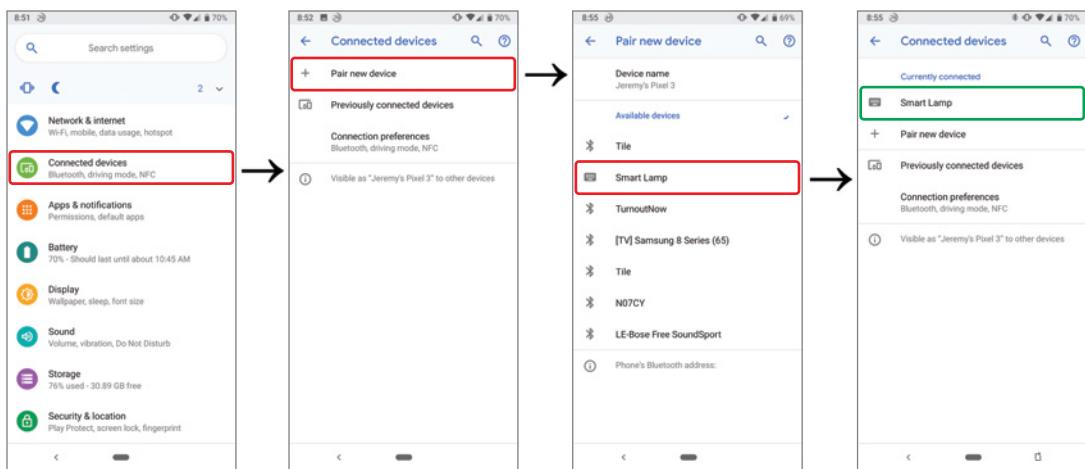


Figure 16-11: Pairing an Android phone

Pairing an iPhone

The process is similar on an iPhone. Go to the Settings app and tap Bluetooth. It automatically starts searching for nearby devices. You should now see your device appear under Other Devices. Tap it. Accept the pairing request that pops up. Your phone should now show that your device is Connected. Figure 16-12 shows this flow.

If your phone disconnects while trying to pair, just try again. Sometimes it takes two or three tries before it “sticks.” This happens with both iPhone and Android. The module does not connect to two devices at the same time, but it does “remember” and automatically pair with more than one device.

WARNING Listing 16-3 does not implement any form of pairing security or authentication. It is therefore possible for anybody within range to connect to your BTLE device (and control the connected lamp by doing so). This project is for experimentation purposes only—you should always implement some form of software security when building a system that can control things in your home or office. If you intend to leave this device running unsupervised, you do so at your own risk. If you want to improve the security of this project, one way to do so would be

to implement an application on the phone that handles bonding to the BTLE device, and communicates a passcode to it that confirms you are authorized to control it, before it starts toggling your lamp. Mobile software development is not covered by this book, but if you want to write an open source application that does that, let me know, and I'll link it from the book's website!

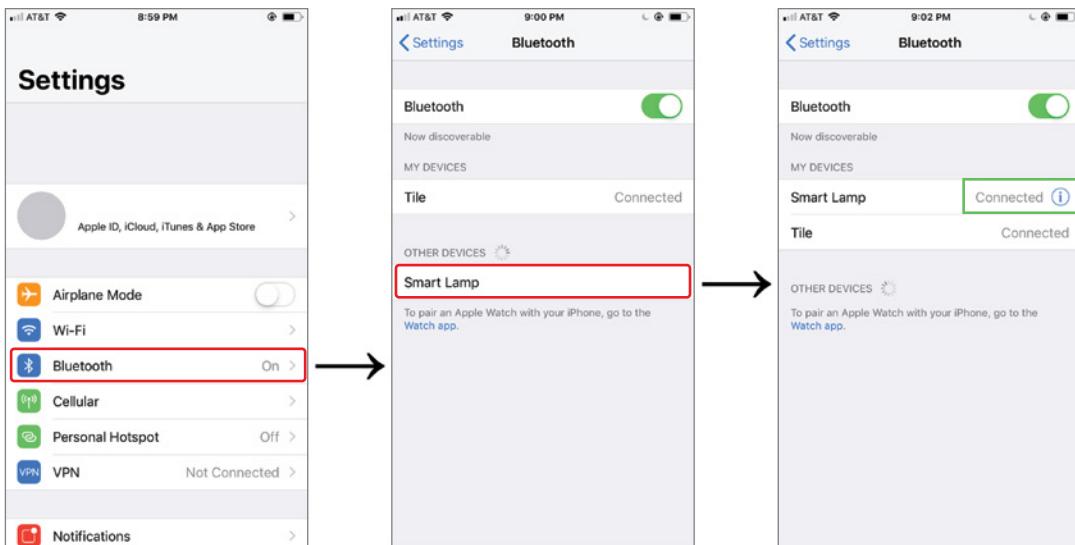


Figure 16-12: Pairing an iPhone

Make Your Lamp React to Your Presence

With your phone paired and your code working, it's time to hook up to an actual lamp. First, adjust the transmit power in the code by doing some tests. Figure out where you want the turn-on/off threshold to be and adjust the power until that is where the LED responds to the connection. Note that the connection and disconnection may not be instantaneous, and that electromagnetic interference may mean that the distance will not always be identical. Connect the relay control line to pin 5 of the Feather board (the same pin the red LED or resistor is currently connected to). Don't forget to connect the negative control line to the Arduino's ground pin. It should look like Figure 16-13.

That's it! Walk in and out of your smart home and marvel at your proximity-controlled lights!

NOTE Watch a demo video of this BTLE proximity lamp controller at exploringarduino.com/content2/ch16.

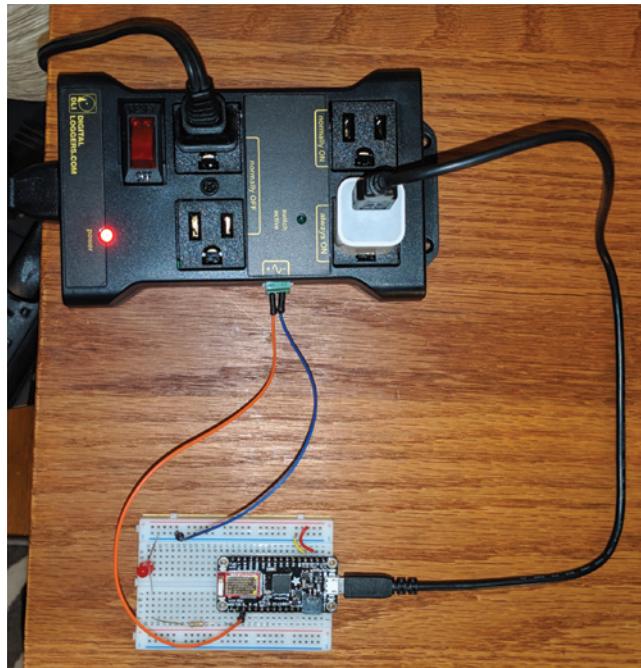


Figure 16-13: Feather Arduino connected to the relay controller

Summary

In this chapter, you learned the following:

- Bluetooth comprises different versions and operating modes that all leverage 2.4 GHz wireless communication channels.
- All modern phones and many other smart devices speak Bluetooth Classic protocols and/or Bluetooth Low Energy.
- It is possible to emulate a UART interface over BTLE.
- Data can be streamed via BTLE from a device to a smartphone.
- A smartphone can be used to send commands to a connected BTLE device.
- Basic natural processing can be achieved with a simple parsing algorithm.
- AT commands can be used to reconfigure modem devices (including Bluetooth modems).
- BTLE does not implement any security by default, so it is up to you to use common sense when building a wireless system that can be openly connected to by anybody in range.

17

Wi-Fi and the Cloud

Parts You'll Need for This Chapter

Adafruit Feather M0 Wi-Fi w/ATWINC1500 (soldered w/ PCB antenna)
USB cable (Type A to Micro-B)
Half-size or full-size breadboard
Assorted jumper wires
 220Ω resistors ($\times 4$)
 $4.7k\Omega$ resistors ($\times 2$)
5 mm common-anode RGB LED
Piezo buzzer
5V 1A USB port wall power supply (optional)
4-digit 7-segment display with I²C backpack (1.2 inch or 0.56 inch, any color)
Wi-Fi network credentials (and optionally, router administrator access)

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at: exploringarduino.com/content2/ch17

Code for this chapter can also be obtained from the “Downloads” tab on this book’s Wiley web page:
wiley.com/go/exploringarduino2e

This is it, the final frontier (and chapter). Short of launching your Arduino into space, connecting it to the internet is probably the closest that you will get to making the entire world your playground. Internet connectivity, in general, is an extremely

complex topic; you could easily write entire volumes of books about the best way to interface the Arduino with the Internet of Things, or IoT, as it is now often called. Because it is infeasible to cover the multitude of ways you can interface your Arduino with the web, this chapter focuses on imparting some knowledge with regard to how network connectivity works with your Arduino (or any IoT device) and how you can use a Wi-Fi-enabled Arduino to both serve up web pages and interact with data from the cloud. Specifically, you will learn about traversing your network topology, how a web page is served, and how to interface with a third-party web-based application programming interface, or API.

The Web, the Arduino, and You

Explaining all the workings of the web is a bit ambitious for one chapter in a book, so for this chapter, you can essentially think of your Arduino's relation to the internet using the diagram shown in Figure 17-1.

First, you will work only in the realm of your local network. When working within your local network, you can talk to your Arduino via a web browser only if they are both connected to the same router (either wired or by Wi-Fi). Then, you will explore ways in which you can traverse your router to access functionality from your Arduino anywhere in the world (or at least anywhere you can get an internet connection).

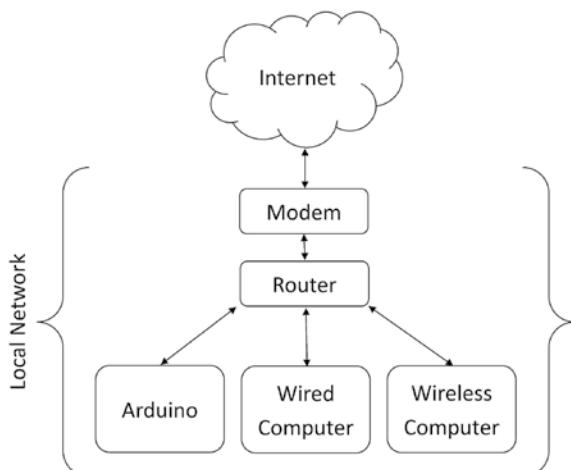


Figure 17-1: A simplified view of the web and your local network

Networking Lingo

Before you get your feet wet with networking your Arduino, let's get some lingo straight. Following are words, concepts, and abbreviations that you will need to understand as you work through this chapter.

The Internet vs. the World Wide Web vs. the Cloud

What we often refer to as *the web*, *the internet*, and *the cloud* are actually all slightly different things, so to start off, it's worth understanding the minor differences between them, as these terms are frequently used interchangeably.

The *internet* is the actual physical network of interconnected devices around the world that all speak to each other. Originally conceptualized in the 1960s, the ARPANET was a project of ARPA, the Advanced Research Projects Agency in the United States. Universities around the world as well as scientific and research agencies built upon the ARPANET, slowly developing it into the *internet* that we recognize today. While the *internet* defined the physical layer and the protocols used to transmit data from point to point, it didn't encapsulate the idea of "websites" until the 1990s. Researchers at CERN, the European Organization for Nuclear Research, introduced the concept of hyperlinked pages, creating the *World Wide Web* that you surf today. The *web* is just one of many applications that run on top of the *internet*.

The cloud is a relatively new term that generally refers to the movement of networked services from local networks (intranets) up to the public internet. In the case of a business, this could mean moving away from on-site network file storage to using a service like Google Drive or Dropbox.

It also applies to application software. For example, whereas most software that you interact with used to exist on your local computer (Microsoft Word is one example), high-speed internet connectivity has made it possible to move many applications to the *cloud*. Google Docs allows you edit a Word document that is actually stored on a Google server (*in the cloud*) instead of on your desktop. The main advantage of *cloud* services is that they reduce dependence on local computing resources, and instead rely on remote server farms that can achieve incredible levels of data throughput by combining a lot of computing power under one roof.

IP Address

An Internet Protocol (IP) address is a unique address that identifies each device that connects to the *internet*. In the case of your home network, there are actually two kinds of IP addresses that you need to worry about: the local IP address and the global IP address. If your home or office has a router (like the one in Figure 17-1), everything within your local network has a local IP address that is visible only to other devices

within your network. Your router/modem has one public-facing global IP address that is visible to the rest of the internet. If you want to move data between somewhere else on the internet and a device behind a router, you need to use network address translation (NAT).

Network Address Translation

There are not enough IP addresses to have one for every device in the world. Furthermore, users often do not want their computers and other networked devices to be visible to the rest of the world. For this reason, routers are used to create isolated networks of computers with local IP addresses. However, when you do want one of these machines to be accessible from the rest of the internet, you need to use NAT through the router. This allows a remote device to send a request to your router asking to talk to a device in your local network. When you connect your Arduino to the larger web later in this chapter, you will use a form of NAT.

NOTE For the purposes of this book, you'll be working with IPv4 addresses, which are of the format `xxx.xxx.xxx.xxx` (where each triplet is a number from 0 to 255). There are 4,294,967,296 IPv4 addresses (2^{32}). We have effectively run out of IPv4 addresses and are making the transition to IPv6 addresses, of which there are 2^{128} (3.4×10^{38}).

MAC Address

MAC addresses, unlike IP addresses, are globally unique. (Well, they're supposed to be, but in practice, they often are not.) MAC addresses are assigned to every physical network interface and do not change. For instance, when you buy a computer, the Wi-Fi module inside has a unique MAC address, and the Ethernet adapter has a unique MAC address. This makes MAC addresses useful for identifying physical systems on a network. Device manufacturers must work with IEEE to obtain a reserved block of MAC addresses to be assigned to the devices that they build.

HTML

Hypertext Markup Language, or HTML, is the language of the web. To display a web page from your Arduino, you will write some simple HTML that creates buttons and sliders for sending data.

HTTP and HTTPS

Hypertext Transfer Protocol, or HTTP, defines the protocol for communicating across the World Wide Web, and is most commonly used in browsers. HTTP defines a set of header information that must be sent as part of a message across the web. This header

defines how a web page will display in addition to whether the request was successfully received and acknowledged. HTTPS is HTTP over Secure Sockets Layer (or SSL); most of the web has moved to using this more secure standard that encrypts all data sent between clients and servers.

GET/POST

GET and POST define two ways for transferring information to a remote web server. If you've ever seen a URL that looks like `jeremyblum.com/?s=arduino`, you've seen a GET request. GET defines a series of variables following a question mark in the URL. In this case, the variable `s` is being set to Arduino. When the page receives this URL, it identifies this variable, performs the search, and returns the results page.

A POST is very similar, but the information is not transmitted in a visible medium through the URL. Instead, the same variables are transmitted transparently in the background. This is generally used to hide sensitive information or to ensure that a page cannot be linked to if it contains unique information.

DHCP

Dynamic Host Configuration Protocol, or DHCP, makes connecting devices to your local network a breeze. Odds are that whenever you've connected to a Wi-Fi (or wired) network, you haven't had to manually set an IP address at which the router can connect to you. So, how does the router know to route packets to you? When you connect to the network, a DHCP request is initiated with the router that allows the router to dynamically assign you an available IP address. This makes network setup much easier because you don't have to know about your network configuration to connect to it. However, it can make talking to your Arduino a bit tougher because you need to find out which IP address it was assigned.

DNS

DNS stands for Domain Name System. Every website that you access on the internet has a unique IP address that is the location of the server on the web. When you type in `google.com`, a DNS server looks at a table that informs it of the IP address associated with that “friendly” URL. It then reports that IP address back to your computer's browser, which can, in turn, talk to the Google server. DNS allows you to type in friendly names instead of remembering the IP addresses of all your favorite websites. DNS is to websites as your phone's contact list is to phone numbers.

Clients and Servers

In this chapter, you learn how to make a Wi-Fi-enabled Arduino act as either a client or a server. All devices connected to the internet are either clients or servers, though some actually fill both roles. A *server* does as the name implies: When information is

requested from it, it serves this information up to the requesting computer over the network. This information can come in many forms: as a web page, database information, an email, or a plethora of other things. A *client* is the device that requests data and obtains a response. When you browse the internet from your computer, your computer's web browser is acting as a client.

Your Wi-Fi-Enabled Arduino

For all the examples in this chapter, you will use an Adafruit Feather M0 Wi-Fi with ATWINC1500 (hereafter simply referred to as the *Feather board*, *Feather*, or *Arduino*). Because of the complexity involved in Wi-Fi connectivity and networking software, it is only feasible for this book to pick this one platform and focus on its use. However, many other Arduinos with Wi-Fi connectivity are available to buy. Because they all use slightly different Wi-Fi chipsets, they will not all work in an identical manner. This chapter will therefore try to explain general concepts in such a way that you can easily extrapolate them to similar hardware if you are not using this exact board.

This Feather board uses an Atmel Cortex M0+ microcontroller, in place of the AVR microcontrollers that you've used in all the previous chapters. The Cortex microarchitecture delivers considerably more horsepower than the AVR chips, and is generally more complex to work with. Thankfully, the Arduino IDE and compiler effectively mask that complexity by abstracting the hardware peripherals for you. You'll program it no differently from any other Arduino that you've worked with.

Controlling Your Arduino from the Web

First, you will configure your Arduino to act as a web server. Using some HTML forms, and the provided Wi-Fi libraries, you will have your Arduino connect to a Wi-Fi network and serve a web page that you can access to control some of its I/O pins. You will expose buttons to the web interface for toggling the colors in an RGB LED and controlling a speaker's frequency. The program that you write for this purpose is extensible, allowing you to add control of additional devices as you become more comfortable working with the Arduino.

Setting Up the I/O Control Hardware

If your Feather board came with its pins unsoldered, then solder them on and install it into a breadboard. You will set up some test hardware that is connected to your Arduino server so that you can control it from the web. For this example, you are connecting an RGB common-anode LED and a piezo buzzer or ordinary speaker. Wire it up as shown

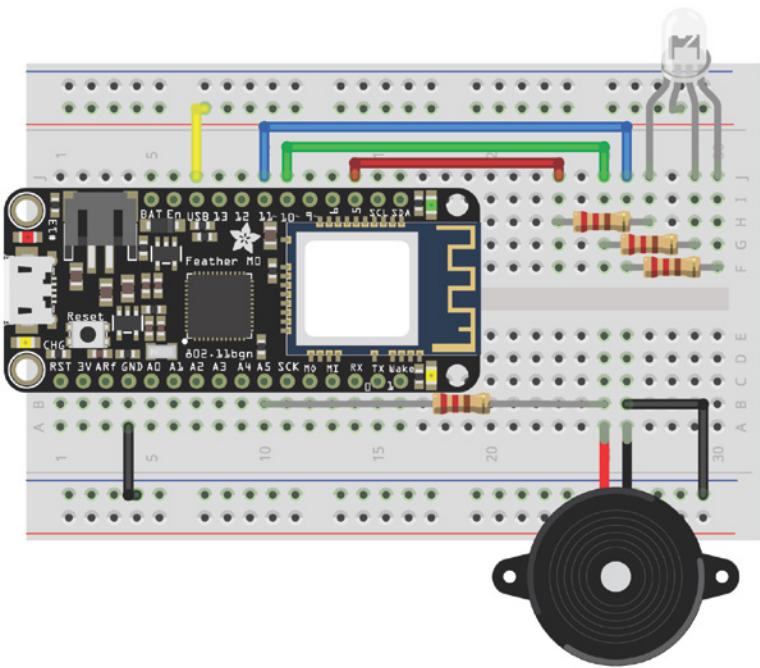


Figure 17-2: Arduino Wi-Fi “server” wired to RGB LED and piezo buzzer

Created with Fritzing

in Figure 17-2. You need to connect your RGB LED to pins 5, 10, and 11. The piezo buzzer or speaker should connect to pin A5 (analog inputs can also be used as digital outputs). Don’t forget to use current limiting resistors for both your piezo buzzer and your LED - 150 Ω or 220 Ω will work fine.

Recall that this Feather board operates at 3.3V logic levels. By connecting the LED’s common anode pin to the “USB” pin on the Feather, you are connecting to the 5V supply provided by the USB interface. This enables you to continue using the same current limiting resistor values that you have previously calculated as sufficient for an LED running off a 5V supply. If you were to run the LED off the Feather’s “3V” pin, you’d need to reduce the resistor values to achieve the same brightness levels.

It’s worth familiarizing yourself with the hardware design of this board — there are details on the Adafruit website, at blum.fyi/feather-wifi-pinout. Specifically, note that Digital pin 9 is also pin A7 in the Arduino software and is connected to a resistor divider for monitoring battery voltage. Hence, it may “float” at a voltage, and is therefore not ideal for driving the LED. pins 2, 4, 7, and 8, along with the hardware SPI pins, are used to communicate with the onboard Wi-Fi chipset.

Preparing the Arduino IDE for Use with the Feather Board

In the last chapter, you learned how to add support for third-party boards to the Arduino IDE. Recall that this involved two steps: first, adding the board URL to the IDE preferences, and second, searching for and adding the specific board support package from the Boards Manager window. See Figure 16-2 and Figure 16-3 if you need a refresher.

You've already added the Adafruit boards URL in Chapter 16, "Bluetooth Connectivity," so you do not need to do that again. If you skipped Chapter 16, go back to the section, "Adding Support for Third-Party Boards to the Arduino IDE," and follow the instructions to add a new boards URL. Then, go to Tools > Board > Boards Manager as you did before. This time, instead of searching for Adafruit AVR boards (which included the 32U4 that you used in the last chapter), you need to search for **SAMD**. Install *both* of the Arduino and Adafruit SAMD support packages, as highlighted in Figure 17-3. Then, restart the IDE. You should now be able to select Adafruit Feather M0 from the list of boards.

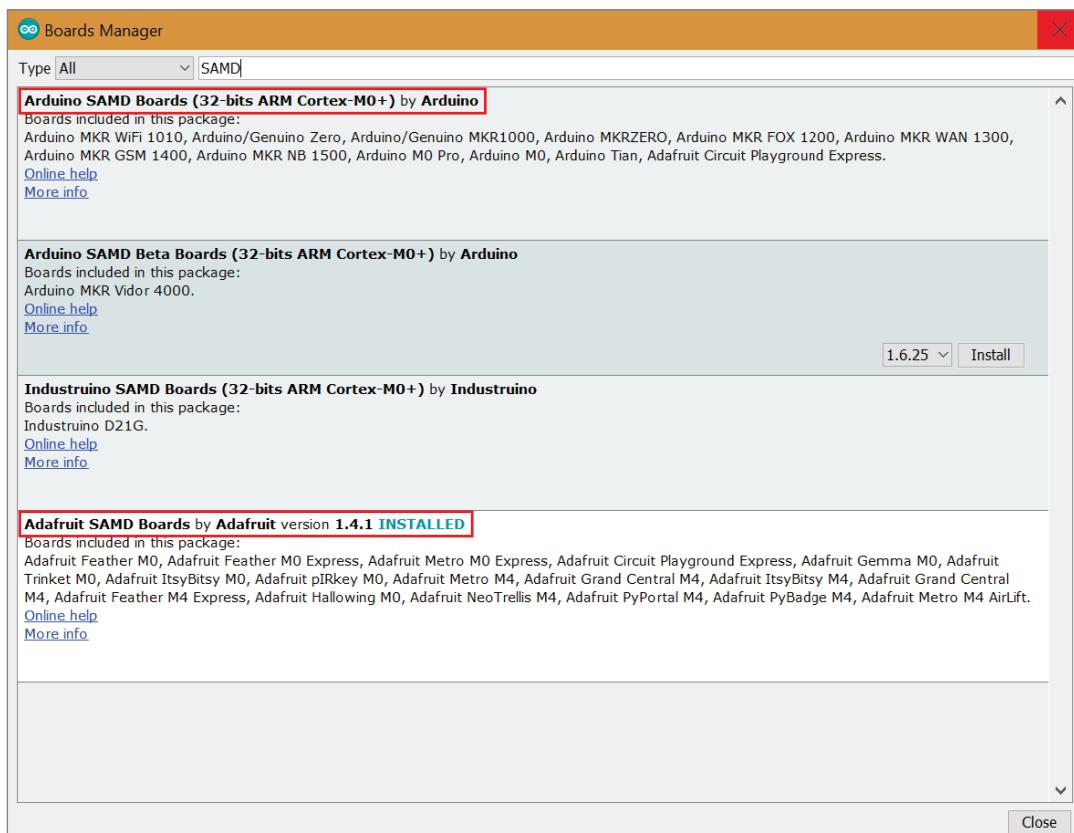


Figure 17-3: Arduino and Adafruit SAMD board support installation

This board may also require drivers to be installed on Windows. You should already be good to go if you installed the Adafruit drivers in the last chapter. If you didn't, just download and install them from blum.fyi/adafruit-windows-drivers.

Finally, you need to install the Arduino library for interacting with the WINC1500 that is integrated onto your Feather board. Go to *Sketch > Include Library > Manage Libraries* and search for **WINC1500**. Install the WiFi101library as shown in Figure 17-4.

Ensuring the Wi-Fi Library Is Matched to the Wi-Fi Module's Firmware

The WINC1500 library that is mounted onto your Feather board is its own little computer that manages all the heavy lifting related to Wi-Fi connectivity. It contains its own microcontroller running its own firmware to manage this task. The library that you just installed allows the main microcontroller (the M0+ that you will be programming with the IDE) to talk to the microcontroller inside the WINC1500. In order for that communication to work properly, the library running on the M0+ must be speaking the same language as the WINC1500. It's possible, therefore, for the firmware running on the WINC1500 to be incompatible with the version of the

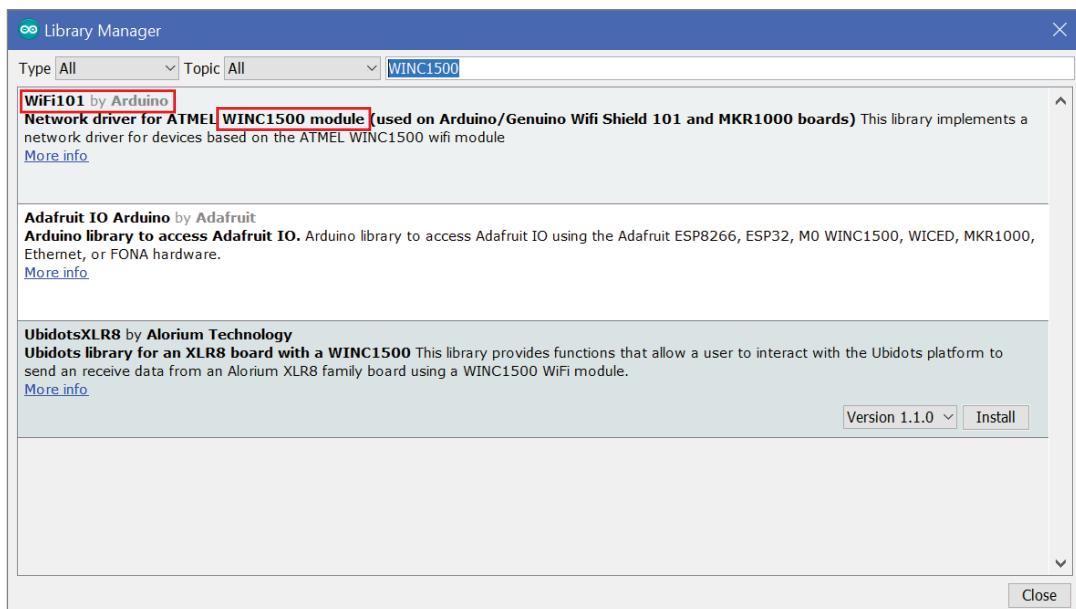


Figure 17-4: WiFi101 library installation

library that you've just installed. If you purchased this board a long time ago, but just installed the WiFi101 library, then the library may be expecting to talk to a WINC1500 with newer firmware.

Checking the WINC1500's Firmware Version

Before you proceed, it's worthwhile to run a simple test script that will connect to the WINC1500 and attempt to query its firmware version to see if it matches what the library is expecting. If the WINC1500 fails to reply, or reports an outdated firmware version, then you need to update its firmware.

Load the firmware-checking example sketch by going to File > Examples > WiFi101 > CheckWifi101FirmwareVersion. You need to add one line to the setup function in this example sketch to set the proper pins for the Wi-Fi module on the Feather board. Above the `serial.begin()` line, add the following:

```
WiFi.setpins(8,7,4,2);
```

This tells the library that the Chip Select, Interrupt, Reset, and Enable lines to the Wi-Fi module are connected to digital pins 8, 7, 4, and 2 on the M0+ microcontroller. Once you've added that line, upload the code to your Feather, and launch the serial monitor. If it tells you that the Library version matches the loaded firmware version, then you're all set! If you get a version mismatch like the one shown in Figure 17-5, then you need to update the firmware on the Wi-Fi module.

Updating the WINC1500's Firmware

If you received a firmware mismatch error, then you need to update the firmware that is running on the Feather's Wi-Fi module before you try to utilize it. Adafruit provides excellent step-by-step instructions on how to perform the upgrade, which you can find at blum.fyi/feather-wifi-update. Follow these instructions, then load the firmware-checking sketch again (with the added lines to set the right pins). This time, it should report a match with the required firmware version.

Writing an Arduino Server Sketch

You'll approach the challenge of building your Arduino web server code in four steps. First, you'll get your Feather to connect to your Wi-Fi network and obtain an IP address. Second, you'll develop the simplest web server possible, so that you can see what HTTP requests look like, how to parse them, and how to respond to them. Third, you'll design a simple webpage that you want your Arduino to display. Finally, you'll integrate the web page and some hardware control code into your web server sketch to make a fully functional project.

The screenshot shows the Arduino IDE interface. The top menu bar includes File, Edit, Sketch, Tools, Help, and a toolbar with icons for upload, refresh, and other functions. The main window has a teal header bar with the title "CheckWifi101FirmwareVersion | Arduino 1.8.9". Below the header is the code editor containing the following C++ code:

```

1 /*
2 * This example check if the firmware loaded on the WiFi101
3 * shield is updated.
4 *
5 * Circuit:
6 * - WiFi101 Shield attached
7 *
8 * Created 29 July 2015 by Cristian Maglie
9 * This code is in the public domain.
10 */
11 #include <SPI.h>
12 #include <WiFi101.h>
13 #include <driver/source/nmasic.h>
14
15 void setup() {
16
17   WiFi.setPins(8,7,4,2);
18
19   // Initialize serial
20   Serial.begin(9600);
21   while (!Serial) {
22     ; // wait for serial port to connect. Needed for native USB port only
23   }
24
25   // Print a welcome message
26   Serial.println("WiFi101 firmware check.");
27   Serial.println();
28

```

The line `WiFi.setPins(8,7,4,2);` is highlighted with a red rectangle. To the right, the串行监视器 (Serial Monitor) window is open, connected to COM15. It displays the following text:

```

WiFi101 firmware check.

WiFi101 shield: DETECTED
Firmware version installed: 19.5.2
Latest firmware version available : 19.6.1

Check result: NOT PASSED
- The firmware version on the shield do not match the
  version required by the library, you may experience
  issues or failures.


```

At the bottom of the Serial Monitor window, there are checkboxes for "Autoscroll" and "Show timestamp", and dropdown menus for "Newline" and "9600 baud".

Figure 17-5: Added pins and firmware mismatch

Connecting to the Network and Retrieving an IP Address via DHCP

Thanks to the wonders of DHCP, securely connecting to a Wi-Fi network with the Arduino Wi-Fi library is a snap. Before you look at the code, I'll explain what is going to happen. At the top of your program, you should include the serial Peripheral Interface (SPI) and Wi-Fi libraries for interfacing with the onboard Wi-Fi module. You'll create a global variable for tracking the Wi-Fi connection status, and constants for holding the Wi-Fi network name (also known as the network's SSID) and password. This assumes that you will be connecting to a Wi-Fi network with modern WPA or WPA2 security (nearly any home network you may encounter).

Within the `setup()`, you will set the pins for the Wi-Fi chipset and start the Wi-Fi connection with the specified network credentials. As you've done in previous chapters, you'll use `while(!serial);` to halt program execution until the USB serial monitor

is open. Once you open the serial monitor, the Feather will connect to the Wi-Fi network, and report the IP address that it was assigned via DHCP. Listing 17-1 shows this program.

Listing 17-1

Connect to Wi-Fi—connect_to_wifi.ino

```
// Connect a Feather M0 with ATWINC1500 to Wi-Fi

#include <SPI.h>
#include <WiFi101.h>

// Wi-Fi Info
const char WIFI_SSID[]      = "PUT NETWORK NAME HERE"; // Wi-Fi SSID
const char WIFI_PASSWORD[]   = "PUT NETWORK PASSWORD HERE"; // Wi-Fi Password

// Indicate connection status with the On-Board LED
const int ONBOARD_LED = 13;

// To keep track of whether we are associated with a Wi-Fi Access Point:
int wifi_status = WL_IDLE_STATUS;

void setup()
{
    // Configure the right pins for the Wi-Fi chip
    WiFi.setpins(8,7,4,2);

    // Setup the pins
    pinMode(ONBOARD_LED, OUTPUT);
    digitalWrite(ONBOARD_LED, LOW);

    // Start the Serial Interface
    Serial.begin(9600);

    // The M0 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
    // want the sketch to run without opening the serial console (or on battery).
    while(!Serial);

    Serial.print("Connecting to: ");
    Serial.println(WIFI_SSID);
    WiFi.setTimeout(5000); // Allow up to 5 seconds for Wi-Fi to connect
    while (wifi_status != WL_CONNECTED)
    {
```

```
wifi_status = WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}
Serial.println("Connected!\n");
digitalWrite(ONBOARD_LED, HIGH); // Turn on the Onboard LED when we connect

// Print the IP that was received via DHCP
IPAddress ip = WiFi.localIP();
Serial.print("This Arduino's IP is: ");
Serial.println(ip);
Serial.println("");
}

void loop()
{
    // Do Nothing
}
```

NOTE If you have trouble getting the Arduino IDE to upload code to your Feather M0 board, it could be because it isn't automatically entering its bootloader. To force it into its bootloader, double-tap the reset button on the Feather board. You'll see the Red LED start to pulse, indicating that the board is now in bootloader mode. Reselect the board's port in the Arduino IDE, and try to upload the code again. You may need to change the port again once the upload has completed.

Load Listing 17-1 onto your Feather, being sure to enter your network credentials at the top of the sketch. Note that the sketch is also set up to control the onboard red LED. When the connection to the Wi-Fi network succeeds, the red LED will illuminate. Open the serial monitor. You should see the connection occur, and an IP address should be printed as shown in Figure 17-6. My Wi-Fi network is called “Exploring Arduino” with a password of “voltaire” (an 18th-century philosopher who advocated for free speech—an important tenet of the modern World Wide Web!).

Once your Arduino is connected, you can ping it to confirm that it is responding to local network requests. On a computer that is on the same network as your Arduino, open your command prompt or terminal application and type `ping XXX.XXX.XXX.XXX` (replacing the X's with your Arduino's reported IP address) to confirm that the Arduino replies to the ping request. This command is the same on all modern operating systems. Figure 17-6 shows the Arduino replying to the ping with a latency of 2 to 3 milliseconds.

The image shows the Arduino IDE interface. On the left, the code for 'connect_to_wifi' is displayed. Lines 18 and 19 are highlighted with red boxes:
`18 const char WIFI_SSID[] = "Exploring Arduino"; // Wi-Fi SSID
19 const char WIFI_PASSWORD[] = "voltaire"; // Wi-Fi Password`

On the right, a terminal window titled 'COM5' shows the output of the Arduino sketch. It displays the connection status and the IP address of the Arduino (192.168.0.141). Below the terminal is a command prompt window showing the results of a 'ping' command to the Arduino's IP address.
`Connecting to: Exploring Arduino
Connected!
This Arduino's IP is: 192.168.0.141
Autoscroll Show timestamp Newline
Command Prompt
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.
C:\Users\Jeremy Blum>ping 192.168.0.141
Pinging 192.168.0.141 with 32 bytes of data:
Reply from 192.168.0.141: bytes=32 time=3ms TTL=128
Reply from 192.168.0.141: bytes=32 time=3ms TTL=128
Reply from 192.168.0.141: bytes=32 time=2ms TTL=128
Reply from 192.168.0.141: bytes=32 time=2ms TTL=128
Ping statistics for 192.168.0.141:
 Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
 Approximate round trip times in milli-seconds:
 Minimum = 2ms, Maximum = 3ms, Average = 2ms
C:\Users\Jeremy Blum>`

Figure 17-6: Arduino connected to Wi-Fi and responding to ping requests

WARNING If you do not see a reply indicating “0% loss” as shown in Figure 17-6, but your Arduino confirms that it is connected to your network, then your network may be configured to block client-to-client pings or connections. This is a common security configuration on “guest” networks where the network administrator wants to allow clients to individually access the web, but doesn’t want those clients to be able to talk to each other over the local network. You will need to speak to your network administrator or set up a home router to continue onto the development of the Arduino-hosted web page.

Writing the Code for a Bare-Minimum Web Server

Now that your Arduino is connecting to Wi-Fi, you can implement a very simple HTTP server that can listen for incoming requests and reply with an acknowledgement.

At a bare minimum, a server just needs to listen for incoming requests, and send back a reply once the full request has been received. To start, the server doesn't even have to understand these requests, or parse them. It just needs to know that they've been received and send back an empty page.

Inside the main loop(), the Arduino waits for a client to connect to its server. Once a client is connected (a browser visiting the web page causes this connection), the Arduino web server reads incoming data until the HTTP request has been fully received (indicated by the receipt of an empty line). To better understand what the request sent by your browser looks like, you'll print out this request to the serial monitor as it comes in. Once the HTTP request is received in full, the Arduino replies with a "200 response" to the browser to indicate that the request was successful.

HTTP RESPONSE CODES

The Hypertext Transfer Protocol outlines a variety of response codes to be used by the server when replying to a request from a client. All responses must always include one of these codes so that the client knows how to interpret the data that is returned. The response code that you are probably most familiar with is code 404. A website will return a 404 response code when the requested resource is not available. For example, if you visit exploringarduino.com/bad-page, you'll get a 404 response and will be redirected to an error page. If the requested page is valid and the server is able to handle the request, then you'll get a 200 response along with the data to render the requested page in your browser.

In addition to the response code, the server also needs to confirm to the browser that it is speaking the same HTTP "language" and in what format the returned data will be provided. The complete response header looks like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

This header must be followed by a blank line, and then the content of an HTML page. For this bare-minimum test, you can just return the bare header, and the browser will show a blank page. Listing 17-2 shows this bare-minimum server code.

Listing 17-2

Bare-minimum server—bare_minimum_server.ino

```
// Arduino Bare Minimum Web Server
// Some code adapted from Arduino Example Code written by Tom Igoe
```

```
#include <SPI.h>
#include <WiFi101.h>

// Wi-Fi Info
const char WIFI_SSID[]      = "PUT NETWORK NAME HERE"; // Wi-Fi SSID
const char WIFI_PASSWORD[]   = "PUT NETWORK PASSWORD HERE"; // Wi-Fi Password

// Indicate connection status with the On-Board LED
const int ONBOARD_LED = 13;

// The server will listen on port 80 (the standard HTTP Port)
WiFiServer server(80);

// To keep track of whether we are associated with a Wi-Fi Access Point:
int wifi_status = WL_IDLE_STATUS;

void setup()
{
    // Configure the right pins for the Wi-Fi chip
    WiFi.setpins(8,7,4,2);

    // Setup the pins
    pinMode(ONBOARD_LED, OUTPUT);
    digitalWrite(ONBOARD_LED, LOW);

    // Start the Serial Interface
    Serial.begin(9600);

    // The M0 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
    // want the sketch to run without opening the serial console (or on battery).
    while(!Serial);

    Serial.print("Connecting to: ");
    Serial.println(WIFI_SSID);
    WiFi.setTimeout(5000); // Allow up to 5 seconds for Wi-Fi to connect
    while (wifi_status != WL_CONNECTED)
    {
        wifi_status = WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
    }
    Serial.println("Connected!\n");
    digitalWrite(ONBOARD_LED, HIGH); // Turn on the Onboard LED when we connect

    // Start the server
    server.begin();
    Serial.println("Server Started!");
}
```

```
// Print the IP that was received via DHCP
IPAddress ip = WiFi.localIP();
Serial.print("Control this Arduino at: http://");
Serial.println(ip);
Serial.println("");
}

void loop()
{
    // Start a server that listens for incoming client connections
    WiFiClient client = server.available();

    // Has a client (browser) connected?
    if(client)
    {
        // While the connection is alive, loop through the incoming lines
        while (client.connected())
        {
            // We'll read in one line of incoming data at a time
            String incoming_line = "";
            // Use a do-while loop so that we don't start checking line formatting
            // until the String contains its first character
            do
            {
                while(!client.available()); // Wait for the next byte to come in
                char c = client.read(); // Once it does, read it in
                incoming_line += c; // And append it to the current line
            } while (!incoming_line.endsWith("\r\n"));

            Serial.print(incoming_line); // Print line that just arrived

            // If last line was empty (only had the carriage return and newline)
            // Then, that means we've received the entire incoming request.
            if (incoming_line == "\r\n")
            {
                // We must acknowledge that the request was received with a valid code
                client.println("HTTP/1.1 200 OK");
                client.println("Content-type:text/html");
                client.println();

                // We can now close the connection
                delay(50);
                client.stop();
            }
        }
    }
}
```

As with Listing 17-1, you need to fill in your Wi-Fi credentials. The main loop works in the way it was described earlier. Each time your browser sends a request to the Arduino, it is parsed in the `while (client.connected())` loop. You read the incoming data one line at a time. Note the use of a `do...while()` loop in place of the more standard `while()` loops that you've used up to this point. The only difference between these two loops is that `do...while()` loops always do one iteration through the loop body before checking the conditions for loop continuation. This is useful in this scenario because you are checking if the current line of received data has been *fully* received (indicated by the presence of a carriage return and newline character at the end). If you checked the given condition at the beginning of the loop, the loop would never execute because an empty string doesn't end with those characters. You first need to ensure that at least one incoming character makes it into the string being checked.

As each full line is received, it is printed out to the serial monitor. Once an empty line is received (`incoming_line == "\r\n"`), you know the complete request has come in and you can reply to it with the 200 response code header described earlier in this section.

Load Listing 17-2 onto your Feather and open the serial monitor. Then, open a web browser and navigate to the URL provided by your Arduino's serial monitor. Note that the computer must be on the same network (either wired or connected via Wi-Fi). It should load a blank white page (because you only sent back a 200 response code with no data). You should see the requests come into your Arduino's serial monitor. You may receive more than one request per page load, as your browser will likely try to request the favicon for the webpage (the tiny icon that is used when you save a bookmark) from the standard location of `/favicon.ico`. Figure 17-7 shows an example of incoming data to your Arduino server.

When you're just loading the "root" page, the GET request shows a path of `/`, which is highlighted in Figure 17-7. Once you add a form to this page, clicking elements on the page will pass GET arguments that will transform that line in the HTTP data to look like this: `GET /?L=10 HTTP/1.1`. By parsing that line, you'll be able to tell what was clicked, and take an action. For example, `L=10` will tell your Arduino to toggle the LED on pin 10. Next, you'll construct the HTML form that will enable this functionality.

DESIGNING A SIMPLE WEB PAGE

It's useful to design a simple web page separately from the Arduino before trying to get the Arduino to serve up the page so that you can ensure that it looks the way you want. Your web page will have simple buttons for toggling each LED, and a slider for adjusting the frequency at which a speaker is playing. It will use HTML form elements to render these components, and it will use the HTTP GET protocol to send commands from the browser to the server as described in the last section. As you

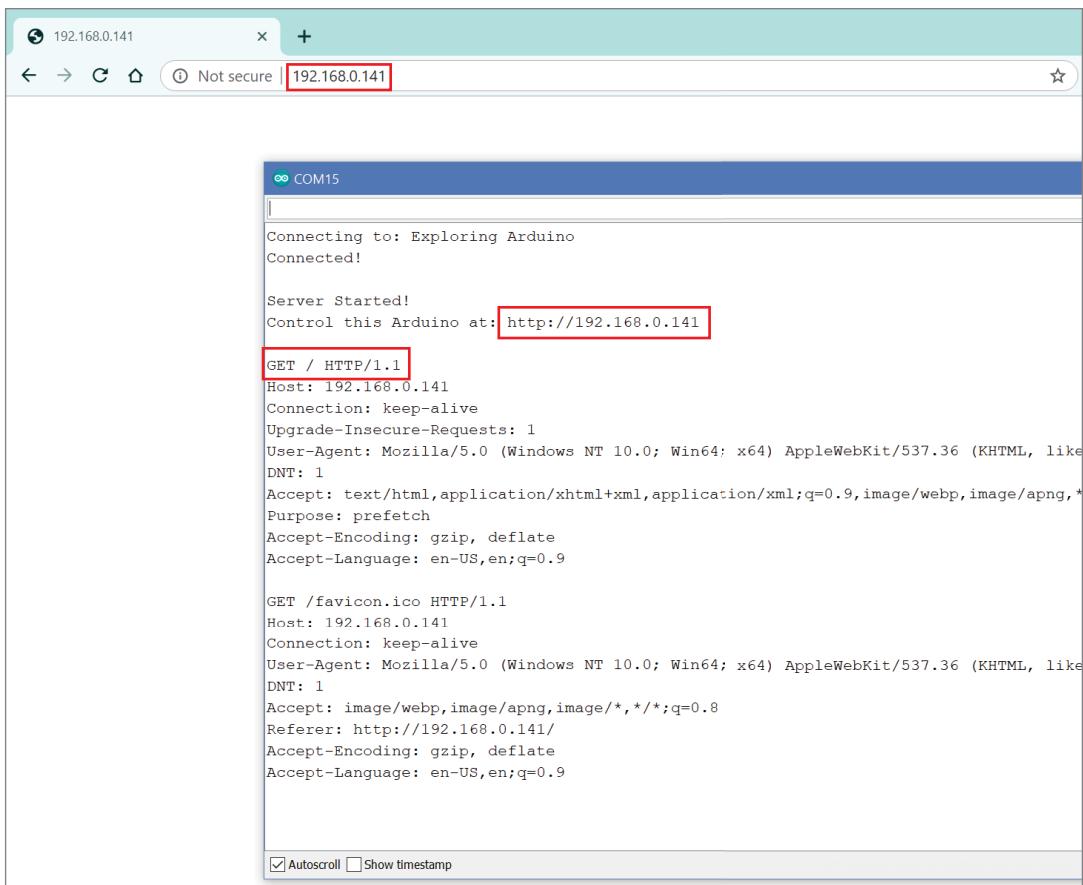


Figure 17-7: Arduino running a server and receiving requests

design the website, it won't actually be hooked up to a server, so interacting with it will not elicit any action from the Arduino, but it will allow you to confirm that the form commands are properly being passed as GET commands in the URL of the page.

Open up your favorite text editor (I recommend Sublime Text—it is available for all OS platforms, and will highlight and color-code your HTML) and create a new file with a .html extension. It doesn't matter what you name the file; test.html will work fine. This will be a very bare-bones website, so don't worry about making this a fully “compliant” HTML website; it will be missing some tags that are normally used, such as <body> and <head>. These missing tags will not affect how the page is rendered in the browser. In your new HTML file, enter the markup from Listing 17-3.

Listing 17-3

HTML form page-server_form.html

```
<form action=' ' method='get'>
  <input type='hidden' name='L' value='5' />
  <input type='submit' value='Toggle Red' />
</form>

<form action=' ' method='get'>
  <input type='hidden' name='L' value='10' />
  <input type='submit' value='Toggle Green' />
</form>

<form action=' ' method='get'>
  <input type='hidden' name='L' value='11' />
  <input type='submit' value='Toggle Blue' />
</form>

<form action=' ' method='get'>
  <input type='range' name='S' min='0' max='1000' step='100' value='0' />
  <input type='submit' value='Set Frequency' />
</form>
```

This HTML page includes four form elements (the HTML between each `<form ...>` and `</form>` tag). `<form>` specifies the beginning of a form, and `</form>` specifies the end. Within each form are `<input />` tags that specify what data will be passed to the server when the form is submitted. In the case of the LED toggle buttons, a variable called `L` will be passed to the server via a GET method with a value equivalent to the I/O pin number that you will be toggling. Once you copy this HTML code snippet into your Arduino sketch, you can replace those hard-coded pins with pin constants. The `action` element set to `' '` (an empty string) in the `form` tag indicates that the same page should be reloaded when the variable is passed to the server. The `hidden input` specifies that this value will just be passed when the Submit button is pressed.

For the frequency slider, you are using an HTML5 input element called `range`. This will make a range slider. You can move the slider (in increments of 100) to select a frequency that will be transmitted as the value of a variable called `S`. In older browsers, this slider may render as an input box rather than a slider, if they don't support the `range` element. To see what the page will look like, open it up with your favorite browser (I recommend Google Chrome). In Chrome, you need to press `Ctrl+O` (Windows) or `Cmd+O` (OS X) to display an Open dialog box. The rendered HTML file should look similar to Figure 17-8.

If you press any of the buttons, you should see a GET statement appended to the address in your browser's URL bar. In Figure 17-8, the GET statement in the URL bar shows that I just pressed the Toggle Blue button because the `L` variable is set to the blue LED pin, 11.

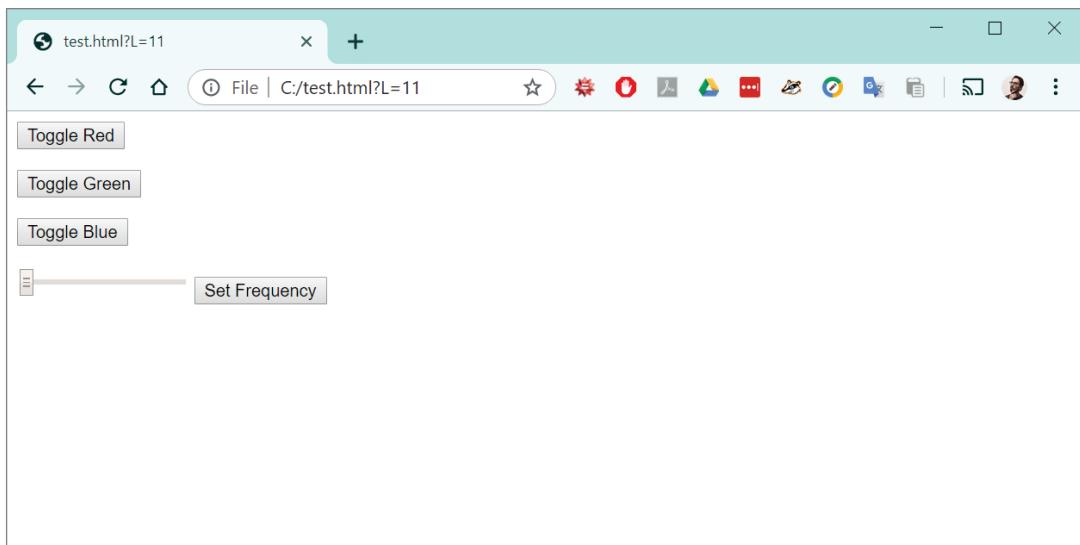


Figure 17-8: Web page content test in Chrome

PUTTING IT TOGETHER: WEB SERVER SKETCH

Now, you need to take the HTML snippet you've developed, and integrate it into a larger server sketch that will handle connecting to the Wi-Fi network, obtaining an IP address, responding to client requests with the page you designed, and performing hardware actions based on GET statements from the page forms.

Given all the requirements listed in the previous sections, you can now construct a server program for the Arduino. The sketch in Listing 17-4 works very well for accomplishing the tasks of controlling an RGB LED and speaker. If you want to add extra functionality with more GET variables, it should be fairly straightforward to do so. The areas where you can insert this extra functionality are called out in the code comments.

Listing 17-4

Web server code—`web_control_server.ino`

```
// Arduino Web Control Server for LEDs and Piezo Buzzer
// Some code adapted from Arduino Example Code written by Tom Igoe

#include <SPI.h>
#include <WiFi101.h>

// Wi-Fi Info
```

```
const char WIFI_SSID[]      = "PUT NETWORK NAME HERE"; // Wi-Fi SSID
const char WIFI_PASSWORD[] = "PUT NETWORK PASSWORD HERE"; // Wi-Fi Password

// Indicate connection status with the On-Board LED
const int ONBOARD_LED = 13;

// pins that the HTML Form will Control
const int RED      = 5;
const int GREEN    = 10;
const int BLUE     = 11;
const int SPEAKER = A5;

// The server will listen on port 80 (the standard HTTP Port)
WiFiServer server(80);

// To keep track of whether we are associated with a Wi-Fi Access Point:
int wifi_status = WL_IDLE_STATUS;

void setup()
{
    // Configure the right pins for the Wi-Fi chip
    WiFi.setpins(8,7,4,2);

    // Setup the pins
    pinMode(ONBOARD_LED, OUTPUT);
    digitalWrite(ONBOARD_LED, LOW);
    pinMode(RED, OUTPUT);
    digitalWrite(RED, HIGH);    // Common Anode RGB LED is Off when set HIGH
    pinMode(GREEN, OUTPUT);
    digitalWrite(GREEN, HIGH);  // Common Anode RGB LED is Off when set HIGH
    pinMode(BLUE, OUTPUT);
    digitalWrite(BLUE, HIGH);   // Common Anode RGB LED is Off when set HIGH

    // Start the Serial Interface
    Serial.begin(9600);

    // The M0 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
    // want the sketch to run without opening the serial console (or on battery).
    while(!serial);

    Serial.print("Connecting to: ");
    Serial.println(WIFI_SSID);
    WiFi.setTimeout(5000); // Allow up to 5 seconds for Wi-Fi to connect
    while (wifi_status != WL_CONNECTED)
    {
        wifi_status = WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
    }
}
```

```
Serial.println("Connected!\n");
digitalWrite(ONBOARD_LED, HIGH); // Turn on the Onboard LED when we connect

// Start the server
server.begin();
Serial.println("Server Started!");

// Print the IP that was received via DHCP
IPAddress ip = WiFi.localIP();
Serial.print("Control this Arduino at: http://");
Serial.println(ip);
Serial.println("");
}

void loop()
{
    // Start a server that listens for incoming client connections
    WiFiClient client = server.available();

    // Has a client (browser) connected?
    if(client)
    {
        // While the connection is alive, loop through the incoming lines
        String command = "";
        while (client.connected())
        {
            // We'll read in one line of incoming data at a time
            String incoming_line = "";
            // Use a do-while loop so that we don't start checking line formatting
            // until the String contains its first character
            do
            {
                while(!client.available()); // Wait for the next byte to come in
                char c = client.read(); // Once it does, read it in
                incoming_line += c; // And append it to the current line
            } while (!incoming_line.endsWith("\r\n"));

            Serial.print(incoming_line); // Print line that just arrived

            // Perform the action requested by "GET" requests
            // Parsing out data from lines that look like: "GET /?L=10 HTTP/1.1"
            if (incoming_line.startsWith("GET /?"))
            {
                // command will look like "L=10"
                command = incoming_line.substring(6,incoming_line.indexOf(" HTTP/1.1"));
            }

            // If last line was empty (only had the carriage return and newline)
            // Then, that means we've received the entire incoming request.
            if (incoming_line == "\r\n")
```

```
{  
    // Reply to all incoming complete requests with our form page  
    // Response Code 200: Request for a page was received and understood  
    client.println("HTTP/1.1 200 OK");  
    client.println("Content-type:text/html");  
    client.println();  
  
    // Red toggle button  
    client.print("<form action='' method='get'>");  
    client.print("<input type='hidden' name='L' value='" + String(RED) + "  
" />");  
    client.print("<input type='submit' value='Toggle Red' />");  
    client.print("</form>");  
  
    // Green toggle button  
    client.print("<form action='' method='get'>");  
    client.print("<input type='hidden' name='L' value='" + String(GREEN) + "  
" />");  
    client.print("<input type='submit' value='Toggle Green' />");  
    client.print("</form>");  
  
    // Blue toggle button  
    client.print("<form action='' method='get'>");  
    client.print("<input type='hidden' name='L' value='" + String(BLUE) + "  
" />");  
    client.print("<input type='submit' value='Toggle Blue' />");  
    client.print("</form>");  
  
    // Speaker frequency slider  
    client.print("<form action='' method='get'>");  
    client.print("<input type='range' name='S' min='0' max='1000' step='100' value='0' /  
" >");  
    client.print("<input type='submit' value='Set Frequency' />");  
    client.print("</form>");  
  
    // You can add more form elements to control more things here  
  
    // End with a blank line  
    client.println();  
  
    // We can now close the connection  
    delay(50);  
    client.stop();  
  
    // Execute the command if one was received  
    if (command.startsWith("L="))  
    {  
        int led_pin = command.substring(2).toInt();  
    }  
}
```

```
    Serial.print("TOGGLING PIN: ");
    Serial.println(led_pin);
    Serial.println("");
    digitalWrite(led_pin, !digitalRead(led_pin));
}
else if (command.startsWith("S="))
{
    int speaker_freq = command.substring(2).toInt();
    Serial.print("SETTING SPEAKER FREQUENCY TO: ");
    Serial.println(speaker_freq);
    Serial.println("");
    if (speaker_freq == 0) noTone(SPEAKER);
    else tone(SPEAKER, speaker_freq);
}
// You can add additional 'else if' statements to handle other commands
}
}
}
```

This code executes all the functionality that was described in the previous sections. Be sure to change the Wi-Fi credentials address listed in this code to match your network. For simplicity, the Arduino responds to every incoming request with the page that you designed in Listing 17-3. Note that the hard-coded pin numbers have been replaced by the pin variables concatenated into the relevant strings. As each line from the client is read in, `if (incoming_line.startsWith("GET /?"))` grabs the ones that may contain commands for the Arduino and strips out the relevant command elements. After the full request has been read in and the reply is sent, the command string is checked for its contents by stripping out the command character (L for the LEDs or S for the speaker) and performing an action based on the command's value. Load Listing 17-4 on to your Arduino and launch the serial monitor.

Controlling Your Arduino from Inside and Outside Your Local Network

Now that the server code is running, and your Arduino is connected to the network with a valid IP address, you can access it with a browser and control it. First, you will do so over your local network, and then you'll learn how you can take advantage of port forwarding in your router to access it from outside of your local network.

Controlling Your Arduino over the Local Network

To confirm that the web interface is working properly, ensure that your computer is attached to the same network as your Arduino (via Wi-Fi or Ethernet). Open your

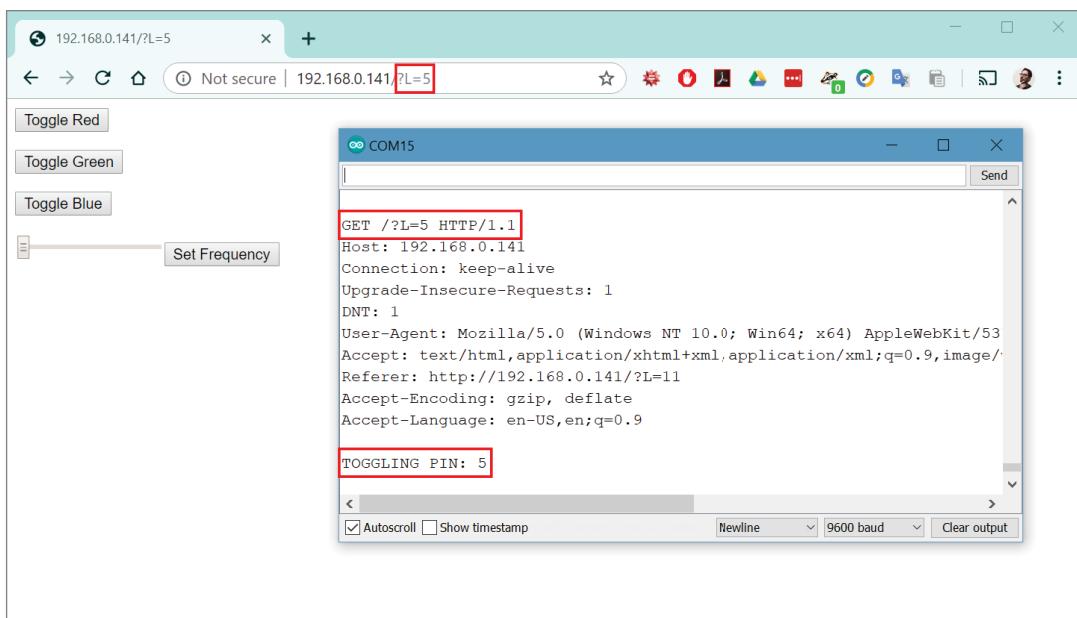


Figure 17-9: Arduino control web page and serial debugging

favorite browser, and enter the IP address from the previous section into the URL bar. This should open an interface that looks just like the HTML page you created earlier. Try pressing the buttons to toggle the various LED colors on and off. Move the slider and hit the frequency adjustment button to set the frequency of the speaker. You should see and hear the Arduino responding. The serial monitor will show the incoming requests as they are received. Notice the GET commands being passed to the Arduino server through the browser's URL bar (see Figure 17-9).

HOW MANY WAYS CAN YOU CONTROL A LAMP?

In the last two chapters, you learned how to control an AC lamp via RF remote control and via Bluetooth proximity. Try expanding your Arduino Wi-Fi server project to control that same lamp using the AC relay that you used for the last two chapters.

After you're satisfied with controlling the lights and sounds over the local network, you can follow the steps in the next section to enable control from anywhere in the world.

NOTE To watch a demo video of the Arduino being controlled over a local network, check out exploringarduino.com/content2/ch17.

Using Port Forwarding to Control Your Arduino from Anywhere

The steps in the previous section enabled you to control your Arduino from anywhere within your local network. This is because the IP address that you are connecting to is a local address that sits behind your router. If you want to control your Arduino from computers outside of your local network, you need to take advantage of advanced technologies that will allow you to *tunnel* to your device through your router from the outside world. To do this, you need to implement three steps:

1. Reserve the local DHCP address used by your Arduino.
2. Forward an external port on your router to an internal port pointing at your Arduino.
3. Connect your router to a dynamic DNS updating service.

WARNING The steps in this section are advanced and will differ (maybe drastically) depending on what kind of router you have. I will generalize, but I also assume that you have some knowledge of router administration. I recommend searching the web for instructions specific to your router for each of the steps listed. If this is your first time logging in to your router's administration (or admin) panel, I don't suggest following these steps; you could potentially mess up your network setup. Some routers may not even support all the functions required to enable port forwarding and dynamic DNS updating. If you are not at all familiar with network administration, stick to local web access for now.

Logging In to Your Router First, log in to your router's administration panel. The admin panel URL is the gateway IP address for your network. In almost all home network configurations, this consists of the first three decimal-separated values of your Arduino's local IP address, followed by a 1. If, for example, your Arduino's IP address were 192.168.0.141, then your gateway address would probably (but not necessarily) be 192.168.0.1. Try typing that address into your browser to see whether you get a login screen. Enter the login credentials for your router admin page; these are not the same as your wireless login credentials. (If you never changed them from the default values, you may be able to find them in your router's setup manual or on a sticker attached to your router.)

If that IP address does not work, you need to determine it manually. On Windows, you can open a command prompt and type `ipconfig`. You want to use the Default Gateway address for your active network connection. If you are on a Mac, access System Preferences, go to Network, click the Advanced button, go to the TCP/IP tab, and use the Router Address. If you are in Linux, open a terminal, type `route -n`, and use the last Gateway Address listing that is nonzero.

Reserving Your Arduino's DHCP Address Once you're in your router's admin console, look for an option to reserve DHCP addresses. By reserving a DHCP address, you are ensuring that every time a device with a particular MAC address connects to the router, it will be assigned the same local IP address. Reserved IP addresses are never given to a client with a MAC address other than the specified address, even if that reserved client is not presently connected to the router. By reserving your Arduino's DHCP IP address, you ensure that you'll always be able to forward web traffic to it in the next step.

Once you find the option, reserve whatever IP address your Arduino is currently using by assigning it to the MAC address that is printed on the sticker attached to the top of the Wi-Fi module on your Feather (see Figure 17-10). Be sure to apply the setting, which may require restarting your router. You can confirm that this works by restarting both your router and the Arduino and seeing if your Arduino gets the same IP address when it reconnects.

Forwarding Port 80 to Your Arduino Now that you have an unchanging local IP address for your Arduino, you need to pipe incoming web traffic to that internal IP address. Port forwarding is the act of listening for traffic on a certain externally facing port of a router and always forwarding that traffic to a specific internal IP address. Port 80 is the default port for HTTP communication, so that is what you will use. Locate the right option in your router administration panel and forward external port 80 to internal port 80 on the IP address that you just assigned to your Arduino. If the router specifies a range for the ports, just make the range 80–80. Now, all traffic to your router on port 80 will go to your Arduino.

Using a Dynamic DNS Updating Service The last step is to figure out how to access your router from elsewhere in the world. If you are working on a commercial network (or you pay a lot for your home's internet connection), you may have a static



Figure 17-10: The MAC address is printed on the Feather's Wi-Fi module

global IP address. This is rare for residential internet connections, but still possible; check with your internet service provider (ISP). If that is the case, just type what is my ip into Google, and it will tell you what your global IP address is. If you know you have a static IP address, you can access that IP address from anywhere in the world, and traffic on it should forward to your Arduino. If you want, you can even buy a domain name and set up your domain name's DNS servers to point to that IP address.

However, the odds are good that you have a dynamic global IP address. Your ISP probably changes your IP address once every few days or weeks. So, even if you figure out what your global IP address is today, and access your Arduino via this IP address, it may stop working tomorrow. There is a clever way around this, which is to use dynamic IP services. These services run a small program on your router that periodically checks your global IP address and reports it back to a remote web server. This remote web server then updates a subdomain that you own (such as `myarduino.ddns.net`) to always point to your global IP address, even when it changes.

Many modern routers have built-in support for certain Dynamic DNS services — you should pick one that your router supports. Some are free, while others charge a nominal yearly fee. You can follow the setup instructions in your router's admin panel to create an account with one of these services and connect it to your router. After doing this, you can access your Arduino remotely, even with a dynamically changing global IP address. In case your router does not support any dynamic DNS services, remember that some also offer clients that will run on computers within your network rather than on the router directly.

Once you have determined your public IP address (or obtained a dynamically updating URL), you can enter it into your browser, and you should be able to connect to your Arduino. Give the address to a friend so they can test it remotely!

WARNING **NETWORK SECURITY** The server sketch you developed for your Arduino has *no security*. If you open up a port on your network to the outside world, then anybody with your public IP address or Dynamic DNS URL can conceivably connect to your Arduino and start fiddling with it. Don't use this approach for mission-critical applications. Whenever you connect a project to the internet, you do so at your own risk.

Interfacing with Web APIs

In the preceding section, you learned how to turn your Arduino into a web server that exposed a web interface for controlling its I/O pins over the local network or the internet. However, an equally common reason for connecting your Arduino to the web

is to interface with application programming interfaces (or APIs). APIs are interfaces exposed by service providers to allow computing systems to programmatically access and/or supply data to or from their services. Here are some examples of APIs from companies and organizations you may be familiar with:

- The Google Maps API allows application developers to embed Google Maps data into their apps.
- The GitHub API allows programmatic access to software projects stored on GitHub. (I use this feature to automatically publish packaged code downloads on exploringarduino.com when I push software updates to the Exploring Arduino GitHub repository.)
- The Phillips Hue API allows you to write software that controls web-connected Philips lightbulbs in your home.
- The NASA API provides a programmatic way to search and download space-related imagery.
- The Facebook API is used by web developers to enable you to log into their websites using your Facebook credentials.

Using a Weather API

For this final project, you'll use an open weather API provided by OpenWeatherMap.org to create a live temperature display for your location. The OpenWeatherMap project is one of many websites that provide a freely accessible weather API with live data for any given location. Its free API offers real-time data and is rate-limited to a maximum of 60 API requests per minute—more than sufficient for this project, which will only update once per minute.

WHY WOULD ANYBODY NEED TO GET WEATHER DATA MORE THAN 60 TIMES PER MINUTE?

Companies often build APIs so that other companies will integrate them into their own applications. If you were developing a weather application for iPhones, for example, you might connect your app to the OpenWeatherMap project and have it use their API to show weather data. That's fine when one person is using that app, but what happens if ten thousand people are using the app? All those requests to the weather service will be registered to the app developer's API key (which you'll learn about in a moment). Distributed among all those users, the application will be making well over 60 requests per minute, and the app developer will need to pay for a higher quantity of API requests. Because you'll just be experimenting and not distributing an app with this API key, you'll be able to keep your usage in the free service range.

APIs are constantly changing, and it is possible that the exact procedure for communicating with the OpenWeatherMap API will be different by the time you pick up this book. The remainder of this chapter should teach you a general approach for interacting with an API, and should be applicable to any API that enables data access with an API key and returns data in a structured format. Good APIs are “versioned” and will continue to work the same way for a long period of time if you continue to specify the particular version.

Creating an Account with the API Service Provider

To start, you’ll probably need an account with the API provider. In the case of the OpenWeatherMap API, just navigate to openweathermap.org and click the Sign Up link. If you’ve found a different API provider that you plan to use for this project, sign up on their website. Once you have signed up and your account has been activated, log into it.

After you sign in, you should see an API Keys section in your account page. Click it to go to a page that lists your API key. An API key is automatically created for you when you set up the account, so there is no need to create another one. Figure 17-11 shows the API Keys page. Keep this page open; you’ll need to copy the listed API key into your program. This API key is unique to you and will authenticate you to the OpenWeatherMap servers.

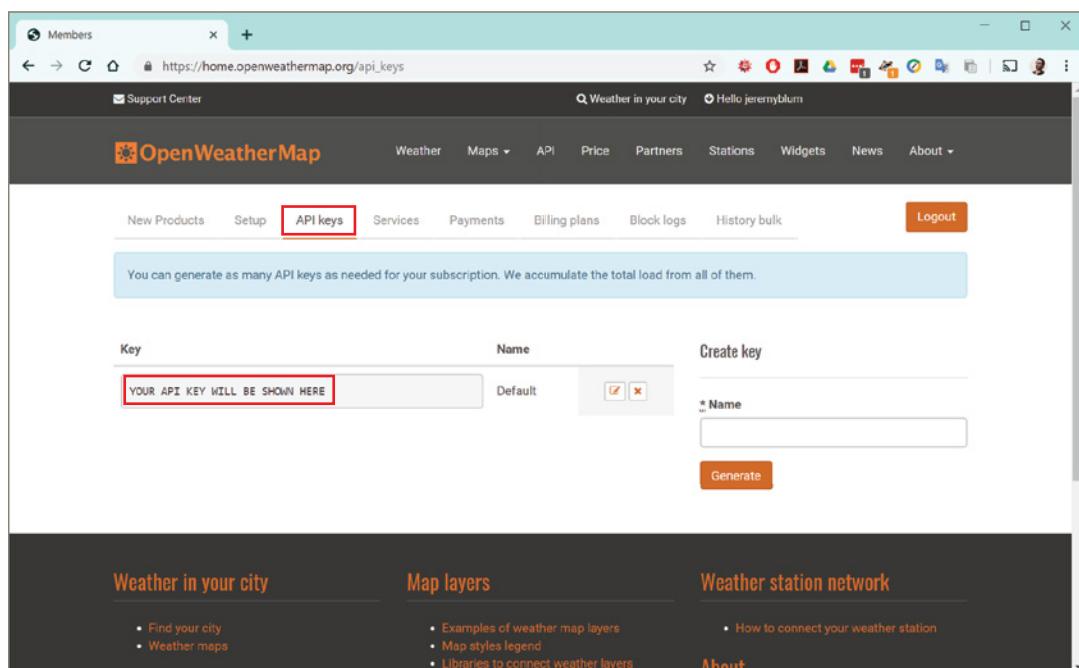


Figure 17-11: OpenWeatherMap.org API key management page

Understanding How APIs Are Structured

APIs are designed with the express purpose of enabling two services, potentially owned or developed by different companies, to communicate with each other and exchange data in an efficient manner. To do this, a few requirements must be met:

1. The API must be able to authenticate the service that is requesting or uploading data.
2. The API should return data in a consistent and easily machine-readable format that uses as little storage and bandwidth as possible to convey the necessary information.
3. The API should not be changed in non-backwards-compatible ways, to ensure that services that rely on it can continue to function.
4. The authenticated API users should only be able to access information to which they have permissions.

To accomplish these design goals, APIs generally employ two important technologies: serializable data formats and key-based or token-based authentication systems. You've already created an account and received your API key. Just like a key for your house, you shouldn't share it with strangers! Keep it private, and use it only for your own projects.

JSON-Formatted Data and Your Arduino

One of the most popular serializable data formats is JSON (pronounced Jay-Sahn), which stands for JavaScript Object Notation. As the name implies, JSON was derived from the JavaScript programming language, but it is now used universally in most programming languages. It is a particularly popular way for formatting data returned by APIs because it maintains human readability while still being easily machine-parseable.

So, what is “serializable”? This just means that a complex, multi-layer data structure encoded in JSON can be easily converted back and forth between a string-type representation that can be easily transmitted from servers to clients with no special protocol requirements. Consider this simple object, which contains information about this book, shown in JSON format:

```
json_object = { "title" : "Exploring Arduino",
                "author_first" : "Jeremy",
                "author_last" : "Blum",
                "edition_list" : [1,2],
                "num_chapters" : 17}
```

This json_object contains five key-index values. Three of them are strings, one of them is a list of numbers, and one is a single number. Individual items in this object can be accessed like this: `book_title = json_object["title"]`. JSON objects can include nested objects inside of them, allowing for a vast amount of organized data to be easily stored.

To send this data over an HTTP connection, you “serialize” it into a string representation that looks like this: `'{"title":"Exploring Arduino","author_first":"Jeremy","author_last":"Blum","edition_list":[1,2],"num_chapters":17}'`. On the receiving end, it can be unpacked and the relevant data can be easily extracted into the required variables. You will use a JSON Arduino library to unpack data returned from the weather API.

Fetching and Parsing Weather Data

To request weather data from the API, you first need to issue a request to the desired endpoint on the API server, while providing the right arguments in the URL. API servers offer a variety of endpoints that each serve up different information. For instance, a `/user` endpoint may return information about the requested user account, and a `/data` endpoint may return whatever kind of data the service provides.

Like the requests you learned about earlier, this one will be a GET request. For this project, you’ll be getting the current weather for a city of your choice. Each endpoint will be appended to the end of the base URL, and then parameters will be passed to the API in the form of GET arguments in the URL. Per the API documentation provided at openweathermap.org/current, the endpoint for getting the current weather is `api.openweathermap.org/data/2.5/weather?q={city name}`, where `{city_name}` will be replaced by your city of choice. Some URL GET parameters are endpoint-specific, while others apply to all API requests. For example, the optional `units` parameter can be included in any API request to set whether the returned data is in degrees Celsius or degrees Fahrenheit. If no unit parameter is specified, the temperature is returned in Kelvin.

All queries must also include an `appid` parameter, set to the API key that you obtained earlier. This allows the API provider to track usage of the API so that they can correctly charge their paying customers, and instruct their servers to ignore unauthorized data requests. Putting that all together, you end up with a complete GET request URL that breaks down as shown in Figure 17-12.

Before you program your Arduino to send a GET request to that URL, it’s worth trying it in your browser to see what the response JSON object will look like. Copy the API request shown in Figure 17-12 into your browser’s URL bar, being sure to insert your personal API key that you received for OpenWeatherMap.org. Optionally, replace San Francisco with the city of your choice. You should get a reply that looks like Figure 17-13.

http://api.openweathermap.org/data/2.5/weather?units=metric&q=San%20Francisco&appid=YOUR_KEY

Protocol API URL Service, Version & Endpoint URL Parameters: Units, Location Query, and AppID (API Key)

Figure 17-12: API request analysis



Figure 17-13: API response in a browser

This confirms that your API key is functional, and your query is valid. But, it's pretty hard to read in this form. Do a web search for **JSON Pretty Printer** and copy the contents of your API reply into it. It should return a prettier version of the API response that looks like this:

```
{
  "coord": {
    "lon": -122.42,
    "lat": 37.78
  },
  "weather": [
    {
      "id": 721,
      "main": "Haze",
      "description": "haze",
      "icon": "50n"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 13.1,
    "pressure": 1011,
    "humidity": 81,
    "temp_min": 11.11,
    "temp_max": 15
  },
  "visibility": 4828,
  "wind": {
    "speed": 2.32,
    "deg": 249.424
  },
  "clouds": {"all": 90},
  "dt": 1558316288,
  "sys": {"type": 1, "id": 4322, "message": 0.0097, "country": "US", "sunrise": 1558270612, "sunset": 1558322141},
  "id": 5391959,
  "name": "San Francisco",
  "cod": 200
}
```

```
  },
  "clouds": {
    "all": 90
  },
  "dt": 1558316288,
  "sys": {
    "type": 1,
    "id": 4322,
    "message": 0.0097,
    "country": "US",
    "sunrise": 1558270612,
    "sunset": 1558322141
  },
  "id": 5391959,
  "name": "San Francisco",
  "cod": 200
}
```

Looking at the response in this format, it's clear that the current temperature will be in the `["main"]["temp"]` variable. There's also a lot of other useful weather information that you can use to expand on this project! Now that you know how to issue an API GET request and how to find the relevant data in the output, it's time to make the Arduino Feather do the heavy lifting.

Getting the Local Temperature from the Web on Your Arduino

Programming your Arduino to issue the GET request that you just issued from your browser is very similar to the process you used to launch the server on your Arduino earlier. You'll continue to use the WiFi101 library, but you will now initialize the Arduino as a client, instead of as the server. You'll format your URL request, send it to the API, and wait to receive a reply back, which you will parse into a serialized JSON string. Once you have that string, you can use the Arduino JSON library to turn the JSON string into an object from which you can extract the relevant data (current temperature).

First, install the Arduino JSON library. Open the Library Manager panel in the Arduino IDE, set the Type to Arduino, and search for **Arduino JSON**. Install the official Arduino_JSON library as shown in Figure 17-14.

As you build your Arduino sketch, start with the same Wi-Fi connection logic that you implemented earlier in this chapter. Import the Arduino JSON library with `#include <Arduino_JSON.h>`. Add some new constants for holding the data that you'll use for constructing your API request:

```
const char SERVER[] = "api.openweathermap.org";
const char HOST_STRING[] = "HOST: api.openweathermap.org";
```

```
const String API_KEY = "PUT YOUR API KEY HERE";
const String CITY = "San Francisco"; // Replace with your City
const String UNITS = "F"; // Set to F or C
```

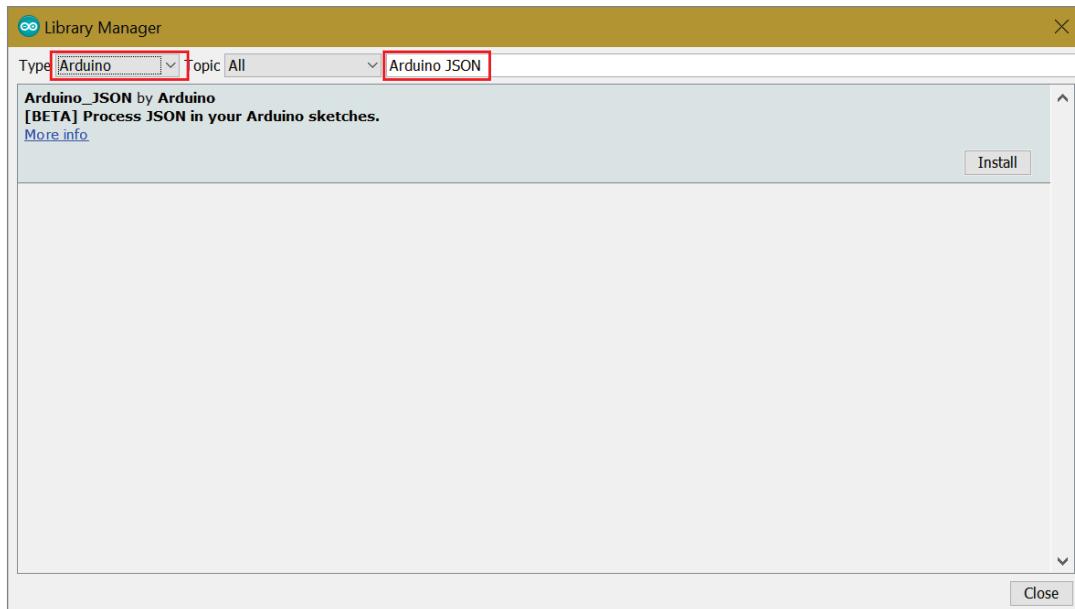


Figure 17-14: Installing the Arduino JSON library

After connecting to Wi-Fi in the `setup()` function, you can initialize a client connection to the API server as follows:

```
String api_units = "metric";
if (UNITS == "F")
{
    api_units = "imperial";
}
String request = "GET /data/2.5/weather?units=" +
    api_units +
    "&q=" +
    CITY +
    "&appid=" +
    API_KEY +
    " HTTP/1.1";
```

```
// Connect to Server and issue a Request
if (client.connect(SERVER, 80))
{
    client.println(request);
    client.println(HOST_STRING);
    client.println("Connection: close");
    client.println();
}
```

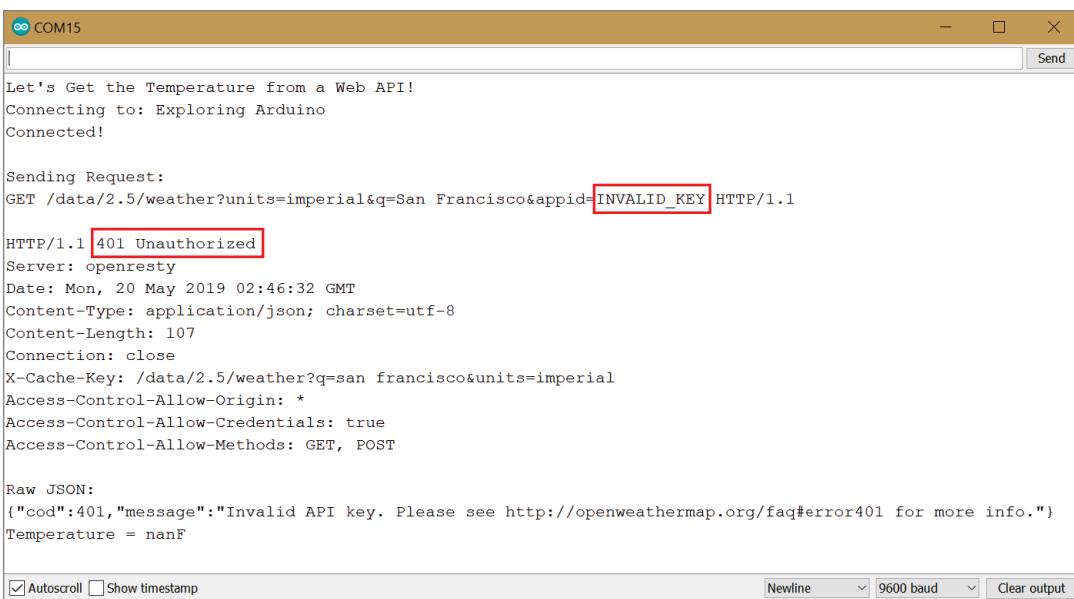
This code constructs the request URL from the variables you've assigned, and connects to the server on port 80 (the standard port for HTTP connections). Finally, you wait for a reply and parse out the JSON string:

```
// Wait for available reply
while (!client.available());

// Throw data out until we get to the JSON object that starts with '{'
// Print the header info so issues can be debugged
while(true)
{
    char h = client.read();
    if (h == '{') break;
    Serial.print(h);
}

// Once we hit the JSON data, read it into a String
String json = "{";
do
{
    char c = client.read();
    json += c;
} while (client.connected());
client.stop();
JSONVar api_object = JSON.parse(json);
Serial.println("Raw JSON:");
Serial.println(api_object);
double temp = (double) api_object["main"]["temp"];
Serial.print("Temperature = ");
Serial.print(temp);
Serial.println(UNITS);
```

Recall that a `while` loop with no contents will effectively halt the program until its condition is true. This means that the program will wait until data is returned to the client. Once it is, you can throw out the header HTTP data by reading until the first open bracket of the JSON reply. Although this code isn't explicitly parsing out potential HTTP error codes, it does print out the header information to the serial console so that you can debug issues with your program. For example, if you use an invalid API



The screenshot shows the Arduino Serial Monitor window titled "COM15". The text output is as follows:

```

Let's Get the Temperature from a Web API!
Connecting to: Exploring Arduino
Connected!

Sending Request:
GET /data/2.5/weather?units=imperial&q=San Francisco&appid=INVALID_KEY HTTP/1.1

HTTP/1.1 401 Unauthorized
Server: openresty
Date: Mon, 20 May 2019 02:46:32 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 107
Connection: close
X-Cache-Key: /data/2.5/weather?q=san francisco&units=imperial
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST

Raw JSON:
{"cod":401,"message":"Invalid API key. Please see http://openweathermap.org/faq#error401 for more info."}
Temperature = nanF

```

At the bottom of the window, there are checkboxes for "Autoscroll" and "Show timestamp", and dropdown menus for "Newline" and "9600 baud". There is also a "Clear output" button.

Figure 17-15: Error resulting from use of an invalid API key

key, the remote server will reject your request, resulting in a 401 error as shown in the serial monitor in Figure 17-15.

Similar to the do...while() loop that you used earlier in this chapter, incoming data is appended into a string until it is complete. Finally, `JSON.parse()` parses the JSON-serialized string into a data structure whose elements can be accessed. `double temp = (double) api_object["main"]["temp"];` creates a double-precision floating point variable called `temp` that is set equal to the value of the temperature that was returned from the API. Finally, this is printed to the serial monitor.

Putting all that together and adding some serial debugging strings, you end up with the simple sketch in Listing 17-5. This one talks to an API and extracts your local temperature data!

Listing 17-5

Get live weather from the web-web_weather.ino

```

// Gets Live Weather Data from the Web

#include <SPI.h>
#include <WiFi101.h>
#include <Arduino_JSON.h>

```

```
// Wi-Fi Info
const char WIFI_SSID[]      = " PUT NETWORK NAME HERE "; // Wi-Fi SSID
const char WIFI_PASSWORD[] = " PUT NETWORK PASSWORD HERE"; // Wi-Fi Password

// API Info
const char SERVER[] = "api.openweathermap.org";
const char HOST_STRING[] = "HOST: api.openweathermap.org";
const String API_KEY = " PUT YOUR API KEY HERE ";
const String CITY = "San Francisco"; // Replace with your City
const String UNITS = "F"; // Set to F or C

// Indicate connection status with the On-Board LED
const int ONBOARD_LED = 13;

// The Arduino is the Client
WiFiClient client;

// To keep track of whether we are associated with a Wi-Fi Access Point:
int wifi_status = WL_IDLE_STATUS;

void setup()
{
    // Configure the right pins for the Wi-Fi chip
    WiFi.setPins(8,7,4,2);

    // Setup the Pins
    pinMode(ONBOARD_LED, OUTPUT);
    digitalWrite(ONBOARD_LED, LOW);

    // Start the Serial Interface
    Serial.begin(9600);

    // The M0 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
    // want the sketch to run without opening the serial console (or on battery).
    while(!serial);

    Serial.println("Let's Get the Temperature from a Web API!");

    Serial.print("Connecting to: ");
    Serial.println(WIFI_SSID);
    WiFi.setTimeout(5000); // Allow up to 5 seconds for Wi-Fi to connect
    while (wifi_status != WL_CONNECTED)
    {
        wifi_status = WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
    }
}
```

```
Serial.println("Connected!\n");
digitalWrite(ONBOARD_LED, HIGH); // Turn on the Onboard LED when we connect

// Prepare the API Request
String api_units = "metric";
if (UNITS == "F")
{
    api_units = "imperial";
}
String request = "GET /data/2.5/weather?units=" +
    api_units +
    "&q=" +
    CITY +
    "&appid=" +
    API_KEY +
    " HTTP/1.1";

// Connect to Server and issue a Request
if (client.connect(SERVER, 80))
{
    Serial.println("Sending Request: ");
    Serial.println(request);
    Serial.println("");
    client.println(request);
    client.println(HOST_STRING);
    client.println("Connection: close");
    client.println();
}

// Wait for available reply
while (!client.available());

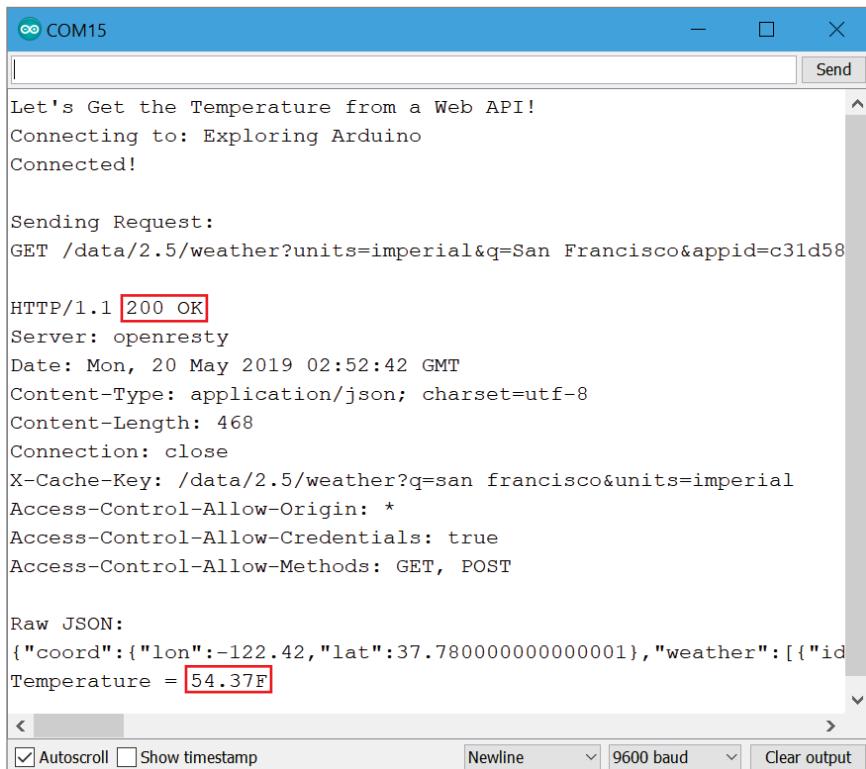
// Throw data out until we get to the JSON object that starts with '{'
// Print the header info so issues can be debugged
while(true)
{
    char h = client.read();
    if (h == '{') break;
    Serial.print(h);
}

// Once we hit the JSON data, read it into a String
String json = "{";
do
{
    char c = client.read();
    json += c;
} while (client.connected());
client.stop();
```

```
JSONVar api_object = JSON.parse(json);
Serial.println("Raw JSON:");
Serial.println(api_object);
double temp = (double) api_object["main"]["temp"];
Serial.print("Temperature = ");
Serial.print(temp);
Serial.println(UNITS);
}

void loop()
{
    // Nothing! We're just getting the data one time in setup
}
```

Load this sketch onto your Feather (you don't need anything to be attached to it other than the USB cable to your computer). Don't forget to fill in your Wi-Fi credentials and your API key where indicated. Optionally, you can change the units from imperial to metric. After the sketch is loaded on, open your serial monitor; you should see a result like Figure 17-16.



The screenshot shows the Arduino Serial Monitor window titled "COM15". The window displays the following text:

```
Let's Get the Temperature from a Web API!
Connecting to: Exploring Arduino
Connected!

Sending Request:
GET /data/2.5/weather?units=imperial&q=San Francisco&appid=c31d58

HTTP/1.1 200 OK
Server: openresty
Date: Mon, 20 May 2019 02:52:42 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 468
Connection: close
X-Cache-Key: /data/2.5/weather?q=san francisco&units=imperial
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST

Raw JSON:
{"coord":{"lon":-122.42,"lat":37.780000000000001},"weather":[{"id":800,"main":"Clear","description":"few clouds","icon":"01d"}]}
Temperature = 54.37F
```

At the bottom of the window, there are several control buttons: "Autoscroll" (checked), "Show timestamp" (unchecked), "Newline" (unchecked), "9600 baud" (selected), and "Clear output".

Figure 17-16: Successful API call from the Arduino

Completing the Live Temperature Display

Now that your Arduino is successfully communicating with a web API, pulling down data, and parsing JSON, you can add a little bit of hardware flair. Showing the temperature on the serial monitor isn't particularly useful, because your computer is already connected to the web! Connect an LED seven-segment readout to your Arduino Feather so that you can display your city's current temperature on it (without needing to be tethered to your computer). Adafruit sells a super-sized 1.2-inch, four-digit, seven-segment readout with an I²C "backpack" that will let you connect it to your Feather with only a few wires. The details for this part are linked from the website for this chapter—it is also available in several colors.

With a simple diffusing piece of thin plastic or paper, you can make an aesthetically pleasing, Wi-Fi-connected temperature readout that will make it easy to decide if you need to grab your jacket on the way out of your home for the day! Figure 17-17 shows an example of the finished project.

Wiring up the LED Readout Display

Before you make the requisite sketch updates to control the display, get it wired up to your Arduino Feather. The suggested large Adafruit four-digit, seven-segment display includes an I²C driver chip. You need to connect the pins to your Feather as follows:

- The pin labeled 'IO' connects the 3.3V. This is the logic voltage to be used for the I²C communications. The voltage provided to this pin sets the HIGH logic level to be used for communications by determining what voltage the onboard pull-up resistors are connected to. Since the Feather is a 3.3V device, and because additional pull-up resistors are necessary to achieve reliable performance (see the "Tuning I²C Buses over Wires" sidebar), this pin must be connected to the Feather's 3.3V supply.



Figure 17-17: Completed live temperature display

- The + pin connects to 5V (the pin labelled USB). This is the positive voltage supply for the LEDs. They require 5V (as opposed to 3.3V) because each segment on the display comprises two LEDs in series. Their combined forward voltage exceeds 3.3V, thus requiring a high voltage supply.
- The - pin connects to Ground.
- The D pin connects to the Feather's I²C Data Pin (SDA).
- The C pin connects to the Feather's I²C Clock Pin (SCL).

TUNING I²C BUSES OVER WIRES

The LED seven-segment display already includes 10kΩ pull-up resistors on the I²C lines. Recall from Chapter 10, "The I²C Bus," that I²C uses an open-drain drive structure and requires pull-up resistors on the data and clock lines. I²C is intended to be used over a short distance, and generally not over wires unless careful design attention is paid to the drive strength of the I/O pins, the values of the pull-up resistors, and bus capacitance. Furthermore, the Arm Cortex (the architecture used in the M0+ microcontroller on your Feather board) I²C peripheral can be less forgiving to data glitches and timing problems than that of the AVR processors that you've used in previous chapters.

Figure 17-18 shows the data and clock signals that were captured with the display connected to the Feather as described previously (blue lines) and with 4.7kΩ pull-up resistors added to the data and clock signals (orange lines).

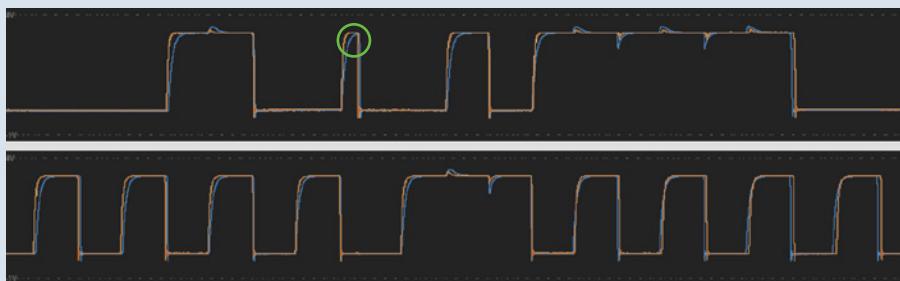


Figure 17-18: I²C signals with and without stronger pull-up resistors

Note the area circled in green. The blue line (without the added pull-up resistors) has a slower rise time. During a short enough pulse, it may not reach the voltage necessary to register as a logical HIGH before it is pulled LOW again. On the orange line, note how the added, lower-value (stronger), pull-up resistors cause the rise time to decrease, giving the signal ample time at the HIGH voltage level to be registered as a valid bit of data by the microcontroller.

The exact performance will vary from board to board and will depend on wire length, breadboard/wire capacitance, and other factors. However, the built-in $10\text{k}\Omega$ resistors are likely to be too weak to interface the LED display with your Feather directly. You should add stronger pull-up resistors (as illustrated in Figure 17-19) to combat increased bus capacitance caused by wires, and to ensure that the signals achieve sufficiently fast rise times. A value of $4.7\text{k}\Omega$ was found to work reliably for this setup. When in doubt, keep the I²C wires short.

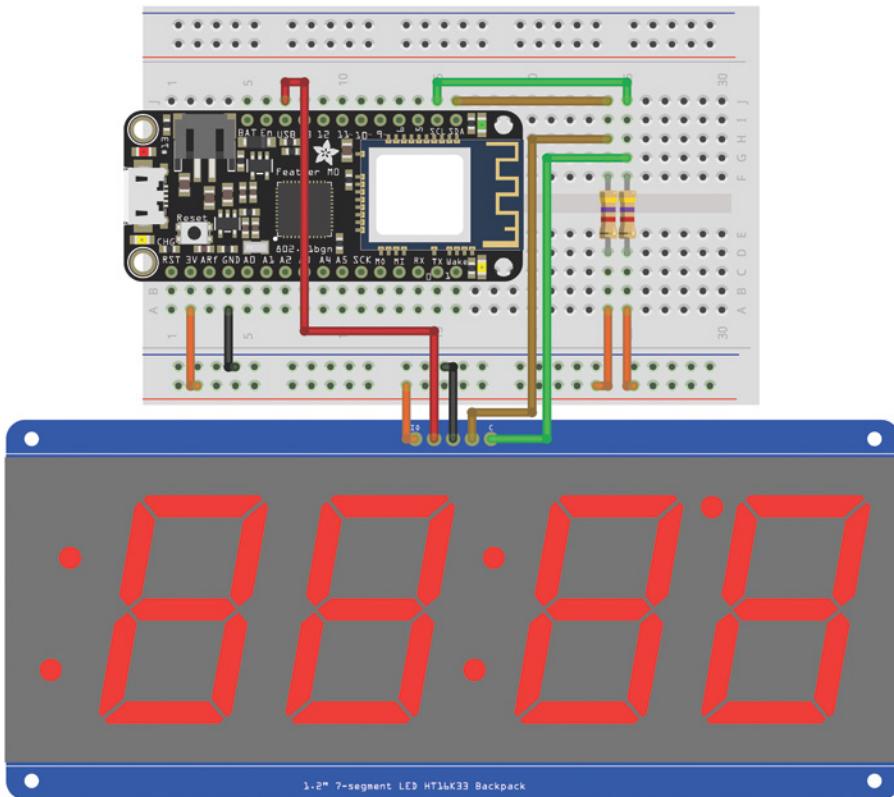


Figure 17-19: Feather wired to an LED display

Created with Fritzing

With your Feather wired as described here, and the stronger pull-up resistors added as described in the previous sidebar, your setup should look like Figure 17-19.

Driving the Display with Temperature Data

Starting with the sketch you've already written to grab temperature data from the web, you now need to add libraries to support the LED display, and to parse the data that will generate the digits for the display.

Open the Arduino IDE's Library Manager and install the two required libraries by searching for both **Adafruit GFX** and **Adafruit LED Backpack**. Install the libraries as shown in Figure 17-20 and Figure 17-21.

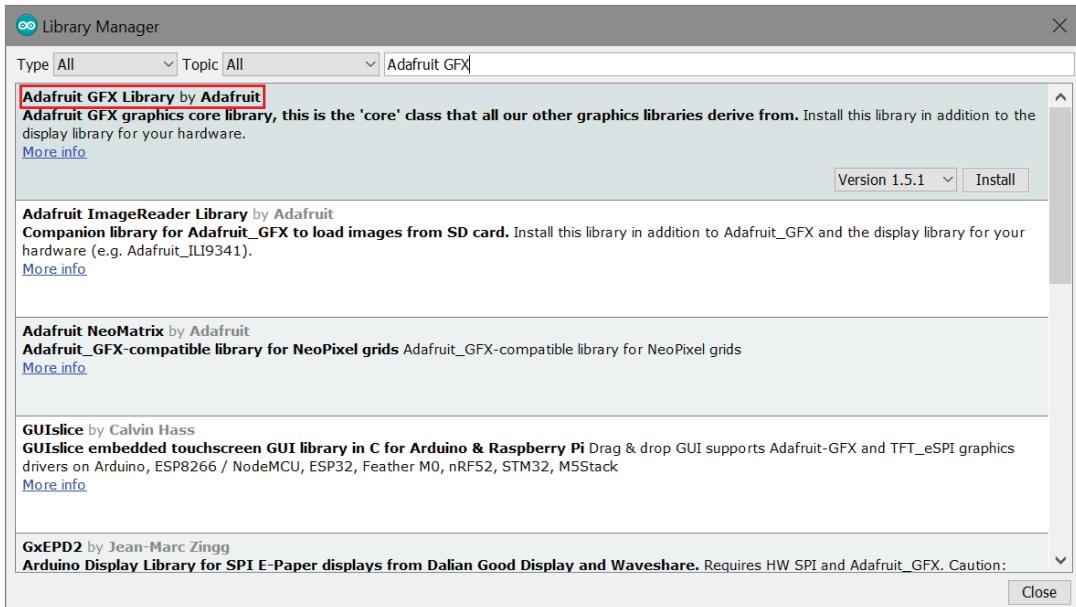


Figure 17-20: Install the Adafruit GFX Library

Include the libraries at the top of your sketch using the following code:

```
#include <Adafruit_GFX.h>
#include <Adafruit_LEDBackpack.h>
```

You also need to create an object for writing data to the display:

```
Adafruit_7segment seven_seg_display = Adafruit_7segment();
```

The temperature that you received from the web needs to be formatted before you can display it on the readout. Use the first three digits of the four-digit display to show

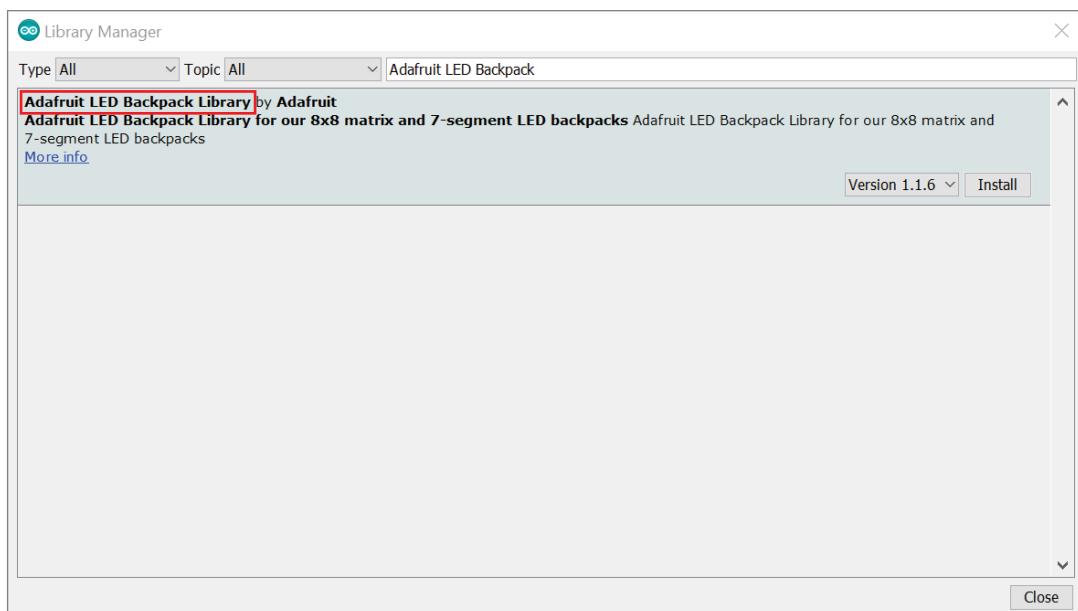


Figure 17-21: Install the Adafruit LED Backpack Library

the temperature value, rounded to the nearest whole number. Use the last digit to show a “C” or an “F” for the temperature unit. Allowing for negative numbers (the negative sign occupies the first digit if present), you’ll be able to show whole numbers from -99 to 999 (let’s hope it doesn’t actually get that hot or cold). So, the first thing you should do is round the data you received to a whole number, and constrain it to that range:

```
int temp_round = constrain(round(temp), -99, 999);
```

The provided functions for printing to the display right-justify the number. However, you want to reserve the rightmost digit for printing the unit. An efficient way to deal with this is to just multiply the temperature by 10. This effectively appends a 0 to the end, which you can then overwrite with the unit before committing the string to the LED driver chip:

```
seven_seg_display.print(temp_round*10);
```

Next, print the degree indicator (there is a small dot on the LED display that is perfectly positioned for this):

```
seven_seg_display.writeDigitRaw(2, 0x10);
```

This snippet is writing a raw hex value to the second digit on the display. Digit 0 is the first number, Digit 1 is the second number, Digit 2 represents the five dots on the LED readout, Digit 3 is the third number, and Digit 4 is the fourth number. By writing 0x10 to the second digit, you ensure that the degree dot is turned on.

Finally, write an "F" or a "C" to the final position on the display. You can also flip the units each time so that on each update, the temperature is shown in a different format:

```
if (UNITS == "F")
{
    seven_seg_display.writeDigitRaw(4,0x71); // Print a "F"
    UNITS = "C"; // Show the opposite unit on the next update
}
else
{
    seven_seg_display.writeDigitRaw(4,0x39); // Print a "C"
    UNITS = "F"; // Show the opposite unit on the next update
}
```

The hex representations of the "F" and the "C" are present in the number table in this library's source code. You can see the exact location at blum.fyi/led-backpack-number-codes.

Add a one-minute delay to the end of the loop so that you do not constantly poll the API server. If you don't do this, you will eventually exceed your free allocation of API calls, and your IP address will be blocked by the server! Putting everything together, you end up with the sketch in Listing 17-6.

Listing 17-6

Get live weather from the web and show it on a readout-web_weather_display.ino

```
// Gets Live Weather Data and Displays it on a big 7-seg Readout

#include <SPI.h>
#include <WiFi101.h>
#include <Arduino_JSON.h>
#include <Adafruit_GFX.h>
#include <Adafruit_LEDBackpack.h>

// Wi-Fi Info
const char WIFI_SSID[]      = " PUT NETWORK NAME HERE ";      // Wi-Fi SSID
const char WIFI_PASSWORD[] = " PUT NETWORK PASSWORD HERE "; // Wi-Fi Password

// API Info
const char SERVER[] = "api.openweathermap.org";
const char HOST_STRING[] = "HOST: api.openweathermap.org";
const String API_KEY = " PUT YOUR API KEY HERE ";
const String CITY = "San Francisco"; // Replace with your City
String UNITS = "F"; // Set to F or C
```

```
// Indicate connection status with the On-Board LED
const int ONBOARD_LED = 13;

// Make the 7-Seg Display Object
Adafruit_7segment seven_seg_display = Adafruit_7segment();

// The Arduino is the Client
WiFiClient client;

// To keep track of whether we are associated with a Wi-Fi Access Point:
int wifi_status = WL_IDLE_STATUS;

void setup()
{
    // Configure the right pins for the Wi-Fi chip
    WiFi.setPins(8,7,4,2);

    // Setup the Pins
    pinMode(ONBOARD_LED, OUTPUT);
    digitalWrite(ONBOARD_LED, LOW);

    // Initialize the display on its default I2C Address
    seven_seg_display.begin(0x70);

    // Start the Serial Interface
    Serial.begin(9600);

    // The M0 has a hardware USB interface, so you should leave the following
    // line uncommented if you want it to wait to start initializing until
    // you open the serial monitor. Comment out the following line if you
    // want the sketch to run without opening the serial console (or on battery).
    // while(!serial);

    Serial.println("Web-Connected Temperature Display");

    Serial.print("Connecting to: ");
    Serial.println(WIFI_SSID);
    WiFi.setTimeout(5000); // Allow up to 5 seconds for Wi-Fi to connect
    while (wifi_status != WL_CONNECTED)
    {
        wifi_status = WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
    }
    Serial.println("Connected!\n");
    digitalWrite(ONBOARD_LED, HIGH); // Turn on the Onboard LED when we connect
}
```

```
void loop()
{
    // Prepare the API Request
    String api_units = "metric";
    if (UNITS == "F")
    {
        api_units = "imperial";
    }
    String request = "GET /data/2.5/weather?units=" +
                    api_units +
                    "&q=" +
                    CITY +
                    "&appid=" +
                    API_KEY +
                    " HTTP/1.1";

    // Connect to Server and issue a Request
    if (client.connect(SERVER, 80))
    {
        Serial.println("Sending Request: ");
        Serial.println(request);
        Serial.println("");
        client.println(request);
        client.println(HOST_STRING);
        client.println("Connection: close");
        client.println();
    }

    // Wait for available reply
    while (!client.available());

    // Throw data out until we get to the JSON object that starts with '{'
    // Print the header info so issues can be debugged
    while(true)
    {
        char h = client.read();
        if (h == '{') break;
        Serial.print(h);
    }

    // Once we hit the JSON data, read it into a String
    String json = "{";
    do
    {
        char c = client.read();
        json += c;
    } while (client.connected());
    client.stop();
    JSONVar api_object = JSON.parse(json);
    Serial.println("Raw JSON:");
}
```

```
Serial.println(api_object);
double temp = (double) api_object["main"]["temp"];
Serial.print("Temperature = ");
Serial.print(temp);
Serial.println(UNITS);

// Show the temperature on the display
int temp_round = constrain(round(temp), -99, 999);
Serial.print("Displaying: ");
Serial.print(temp_round);
Serial.println(UNITS);

// Prints right justified, so multiplying by 10 moves it left one digit
// This makes room for the "C" or "F" unit to the right
seven_seg_display.print(temp_round*10);

// This prints the dot that will serve as the degree sign
seven_seg_display.writeDigitRaw(2, 0x10);

// Print the units
if (UNITS == "F")
{
    seven_seg_display.writeDigitRaw(4,0x71); // Print a "F"
    UNITS = "C"; // Show the opposite unit on the next update
}
else
{
    seven_seg_display.writeDigitRaw(4,0x39); // Print a "C"
    UNITS = "F"; // Show the opposite unit on the next update
}

//Write to the display
seven_seg_display.writeDisplay();

// Wait about one minute before checking again
Serial.print("Waiting one minute before next check.");
for (int i = 0; i <60; i++)
{
    Serial.print(".");
    delay(1000); // Delay 1 second
}
Serial.println("");
Serial.println("");

}
```

Replace the network credentials and API key with your own. Also be sure to set your city accordingly. In the listing, I've commented out the serial wait loop so that it

will not wait for a serial interface before it starts working. This will allow you to just connect to power (potentially using a USB power brick) and deploy your live temperature display anywhere in your home where you get a Wi-Fi signal.

NOTE To watch a demo video of the Arduino receiving temperature data from the web and displaying it on an LED readout, check out exploringarduino.com/content2/ch17.

Once you've got this project working, how else can you envision using the data from this API? Perhaps consider connecting a speaker to alert you when thunderstorms are approaching, or connect a light bulb that will turn on at the official sunset time and off at the official sunrise time.

Summary

In this chapter, you learned the following:

- The internet has a lot of acronyms. You learned the meanings of IP, DHCP, DNS, MAC, and more.
- Internet-connected devices can act as a clients and/or server.
- HTML can be used to render a form for controlling your Arduino over the web.
- Your Arduino can act as a simple web server.
- APIs can be leveraged to communicate with third-party data services.
- Your Arduino can query an API and receive data back in JSON format.
- Your Arduino can de-serialized JSON strings and extract relevant data from them.
- An I²C bus's pull-ups must be properly tuned to ensure that it performs reliably.
- Using third-party libraries, an Arduino can control an I²C-based, four-digit, seven-segment LED display.

Appendix A: Deciphering Datasheets and Schematics

At the heart of all Arduinos is a microcontroller (or MCU for short). This appendix does not summarize the features of every microcontroller in every Arduino, but it does provide a brief guide to reading and understanding datasheets. Specifically, it examines elements of the Microchip (previously Atmel) ATmega328P (the MCU used in an Arduino Uno). In addition to understanding component datasheets, learning how to read the key parts of a schematic is a critical skill. This appendix investigates the schematic for the Arduino Uno, so that you can get a better idea of how an Arduino actually works.

Reading Datasheets

One of the most important skills that you can develop as an engineer is the ability to read datasheets. Just about any electronic component that you can buy has an associated datasheet that contains information about the technical limits of the part, instructions on how to use its capabilities, and so forth.

Breaking Down a Datasheet

Consider the datasheet for the Microchip/Atmel ATmega328P, for instance. Recall that the ATmega328P is the MCU used in the Arduino Uno and many Arduino clones. To find the datasheet for most parts, you can just perform a Google search. For example, to find the datasheet for the ATmega328P, just search for **ATmega328P datasheet** on Google and look for the first PDF link from the manufacturer's website (microchip.com in this case). The datasheets for the MCUs used in official Arduino boards can also be found on the hardware page for each board on the [Arduino.cc](http://arduino.cc) website. When you have the datasheet in hand, start by reviewing the first few pages (see Figure A-1). In most cases, the first few pages will provide a high-level overview of the features of that MCU.

 **MICROCHIP**

ATmega48A/PA/88A/PA/168A/PA/328/P

megaAVR® Data Sheet

Introduction

The ATmega48A/PA/88A/PA/168A/PA/328/P is a low power, CMOS 8-bit microcontrollers based on the AVR® enhanced RISC architecture. By executing instructions in a single clock cycle, the devices achieve CPU throughput approaching one million instructions per second (MIPS) per megahertz, allowing the system designer to optimize power consumption versus processing speed.

Features

- High Performance, Low Power AVR® 8-Bit Microcontroller Family
- Advanced RISC Architecture
 - 131 Powerful Instructions – Most Single Clock Cycle Execution
 - 32 x 8 General Purpose Working Registers
 - Fully Static Operation
 - Up to 20 MIPS Throughput at 20MHz
 - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory Segments
 - 4/8/16/32KBytes of In-System Self-Programmable Flash program memory
 - 256/512/512/1KBytes EEPROM
 - 512/1K/1K/2KBytes Internal SRAM
 - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
 - Data retention: 20 years at 85°C/100 years at 25°C⁽¹⁾
 - Optional Boot Code Section with Independent Lock Bits
 - In-System Programming by On-chip Boot Program
 - True Read-While-Write Operation
 - Programming Lock for Software Security
- QTouch® library support
 - Capacitive touch buttons, sliders and wheels
 - QTouch and QMatrix™ acquisition
 - Up to 64 sense channels
- Peripheral Features
 - Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
 - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode

Figure A-1: The first two pages of the ATmega328P datasheet*Credit: © Microchip Technology Incorporated. Used with permission.*

ATmega48A/PA/88A/PA/168A/PA/328/P

- Real Time Counter with Separate Oscillator
- Six PWM Channels
- 8-channel 10-bit ADC in TQFP and QFN/MLF package
 - Temperature Measurement
- 6-channel 10-bit ADC in PDIP Package
 - Temperature Measurement
- Programmable Serial USART
- Master/Slave SPI Serial Interface
- Byte-oriented 2-wire Serial Interface (Philips I²C compatible)
- Programmable Watchdog Timer with Separate On-chip Oscillator
- On-chip Analog Comparator
- Interrupt and Wake-up on Pin Change
- Special Microcontroller Features
 - Power-on Reset and Programmable Brown-out Detection
 - Internal Calibrated Oscillator
 - External and Internal Interrupt Sources
 - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby, and Extended Standby
- I/O and Packages
 - 23 Programmable I/O Lines
 - 28-pin PDIP, 32-lead TQFP, 28-pad QFN/MLF and 32-pad QFN/MLF
- Operating Voltage:
 - 1.8 - 5.5V
- Temperature Range:
 - -40°C to 85°C
- Speed Grade:
 - 0 - 4MHz@1.8 - 5.5V, 0 - 10MHz@2.7 - 5.5V, 0 - 20MHz @ 4.5 - 5.5V
- Power Consumption at 1MHz, 1.8V, 25°C
 - Active Mode: 0.2mA
 - Power-down Mode: 0.1µA
 - Power-save Mode: 0.75µA (Including 32kHz RTC)

Figure A-1: (continued)

From a quick glance at the datasheet, you can learn a considerable amount about the microcontroller. You can ascertain that it can be reprogrammed about 10,000 times, and that it can operate from 1.8V to 5.5V (5V in the case of the Arduino). You can also learn how many inputs and outputs (I/Os) it has, what special functions it has built in (like hardware Serial Peripheral Interface [SPI] and I²C interfaces), and the resolution of its analog-to-digital converter (ADC).

It's common for a single datasheet to be shared by several products that are in the same class. The datasheet for the 328P also serves as the datasheet for the ATmega 48, 88, and 168 class MCUs. These MCUs are largely identical, with the entirety of their differences explained in Table 2-1 of the datasheet (duplicated in Figure A-2).

Table 2-1. Memory Size Summary

Device	Flash	EEPROM	RAM	Interrupt Vector Size
ATmega48A	4KBytes	256Bytes	512Bytes	1 instruction word/vector
ATmega48PA	4KBytes	256Bytes	512Bytes	1 instruction word/vector
ATmega88A	8KBytes	512Bytes	1KBytes	1 instruction word/vector
ATmega88PA	8KBytes	512Bytes	1KBytes	1 instruction word/vector
ATmega168A	16KBytes	512Bytes	1KBytes	2 instruction words/vector
ATmega168PA	16KBytes	512Bytes	1KBytes	2 instruction words/vector
ATmega328	32KBytes	1KBytes	2KBytes	2 instruction words/vector
ATmega328P	32KBytes	1KBytes	2KBytes	2 instruction words/vector

Figure A-2: Differences between chips defined in this datasheet

Credit: © Microchip Technology Incorporated. Used with permission.

NOTE This datasheet is actually hundreds of pages long, and an entire book could probably be dedicated just to interpreting it, so I won't go much further here. However, throughout the remainder of this appendix, I do point out several important topics to look out for.

Datasheets as long as this one generally have PDF bookmarks built in that make it easier to find what you're looking for. Of particular interest for your Arduino adventures may be information about I/O ports, the timers, and the various hardware serial interfaces. As one more example, consider Figure 14-1 from the datasheet's I/O section in the PDF, which is reproduced in Figure A-3.

Diagrams like this one can be found throughout the datasheet, and can give you a deeper insight into how your Arduino is actually working. In this example, you can

see that the I/O pins all have protection diodes to shield them from excessively high or negative voltages. It's also important to observe that there is a known pin capacitance, which could have significant implications if you're trying to determine the rise and fall times when switching the value of a pin.

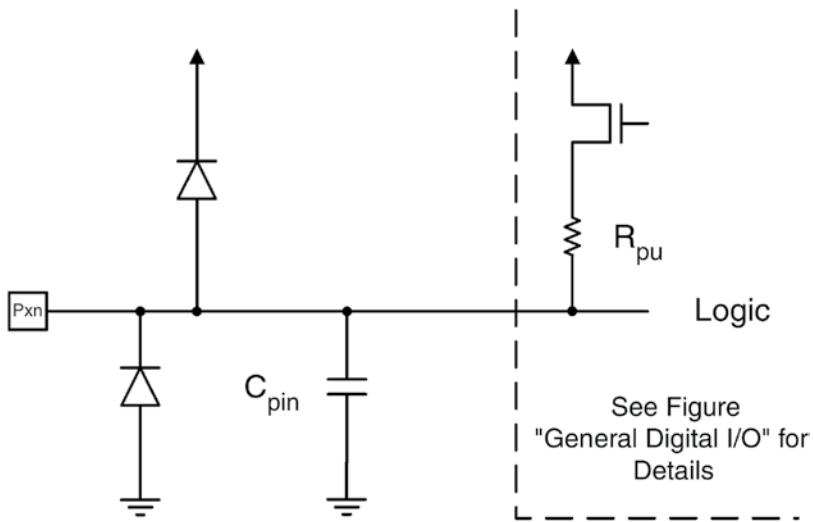


Figure A-3: I/O pins diagram

Credit: © Microchip Technology Incorporated. Used with permission.

Understanding Component Pin-Outs

All datasheets include the pin-out for the device in question, which clearly illustrates the functions of each pin. Particularly for microcontrollers, pins may have multiple functions, so understanding the pin-out can be critical for grasping what each pin can and cannot do. Consider the pin-out of the ATmega328P (see Figure A-4). Understanding the pin-out of the microcontroller at the heart of the Arduino Uno will make it easier to understand the Uno's schematic, which you'll look at in the next section.

Many microcontrollers are available in a variety of package sizes, as you can see in Figure A-4. All electronic components used to be made in dual inline packages, or DIPs (with pins that were soldered through holes). Over the last couple of decades, the industry has transitioned almost entirely to surface-mount device (SMD) packages, which can be easily placed on boards with robotic “pick-and-place” machines and

soldered down in an oven. These SMD packages enable faster and cheaper electronics manufacturing. To learn more about the advantages of surface-mount devices, read the “Device Miniaturization and SMT” sidebar in Chapter 11, “The SPI Bus and Third-Party Libraries.”

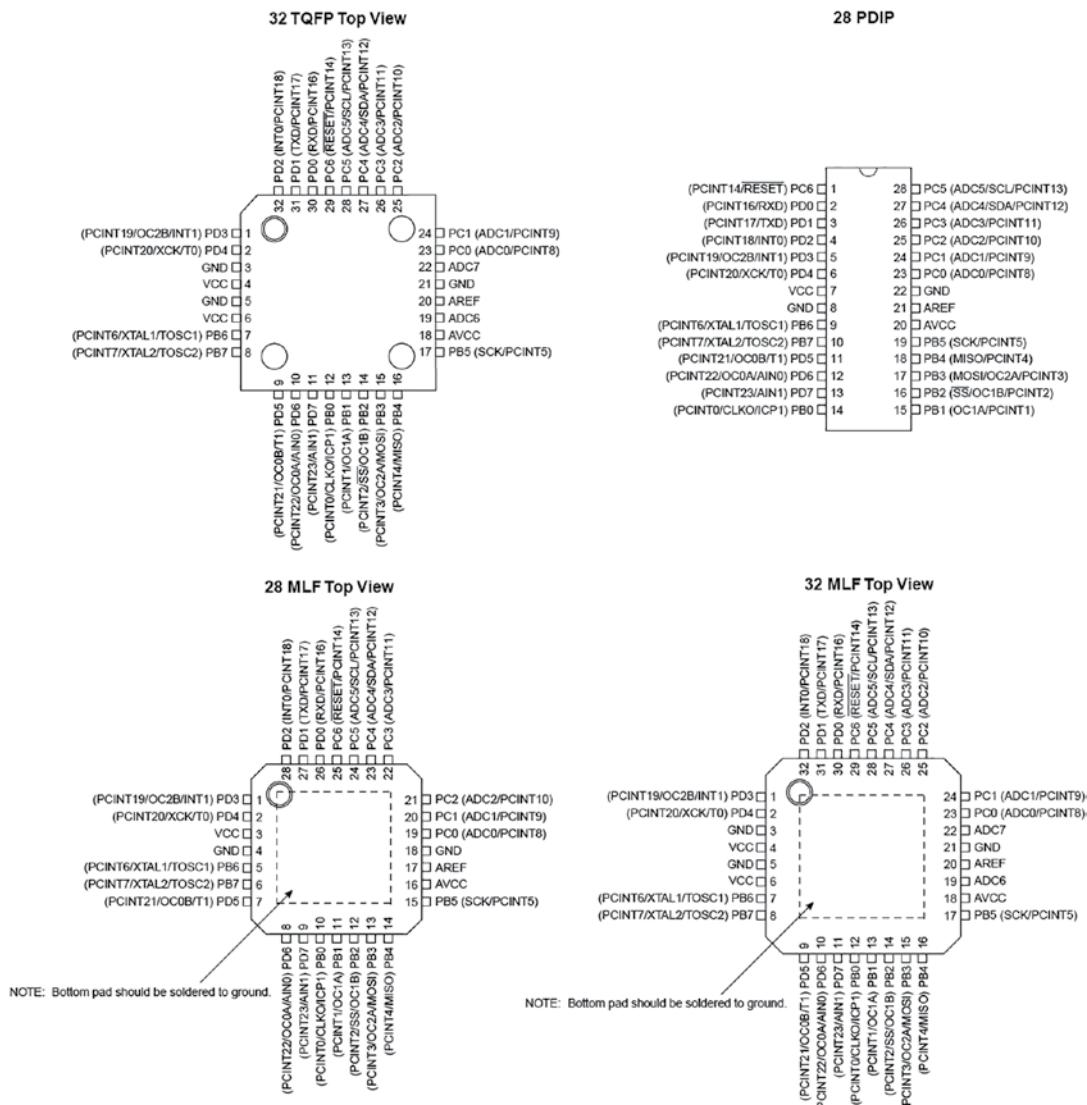


Figure A-4: ATmega328P pin-outs

Credit: © Microchip Technology Incorporated. Used with permission.

Your Arduino Uno may have a DIP-style 328P or one of the SMD variants. Either way, the internal silicon is the same—it's only packaged differently. On DIP packages, the half circle at the top of the pin-out corresponds to a similar half circle on the actual chip. Look at the chip in your Arduino Uno, and you'll see this half circle; now you know that the pin immediately to its left is pin 1. If you are using an Arduino Uno variant or clone that has a surface-mount MCU package instead of a DIP package, then pin 1 will be indicated with a dot on the corner of the package.

You'll also probably notice some abbreviations that you may not be familiar with. They are defined here:

- VCC refers to voltage supply to the chip. In the case of the Arduino Uno, VCC is 5V.
- AVCC is a separate supply voltage for the ADC. For the Arduino Uno, it is also 5V.
- AREF is broken out to a pin. As a result, you can choose an arbitrary voltage below 5V to act as the reference for the ADC if you desire.
- GND is, of course, the ground connection.

The rest of the pins are all general-purpose I/O. Each is mapped to a unique pin number in the Arduino software so that you don't have to worry about the port letter and number.

The labels in parentheses represent alternative functions for each pin. For example, pins PD0 and PD1 are also the Universal Synchronous/Asynchronous Receiver and Transmitter (USART) Receive (RX) and Transmit (TX) pins, respectively. Pins PB6 and PB7 are the crystal connection pins (XTAL). In the case of the Arduino Uno, an external 16 MHz ceramic resonator or crystal is connected to these pins, so you cannot use them for general-purpose I/O.

If you have trouble deciphering the pin labels, you can usually learn more about what they mean by searching the rest of the datasheet for those terms. The Arduino website has a diagram illustrating how the ATmega pins are connected to numbered pins on the Arduino board. You can find it at blum.fyi/arduino-uno-pin-map, and it is also shown in Figure A-5.

ATmega168/328P-Arduino Pin Mapping

Note that this chart is for the DIP-package chip. The Arduino Mini is based upon a smaller physical IC package that includes two extra ADC pins, which are not available in the DIP-package Arduino implementations.

Atmega168 Pin Mapping				
Arduino function		Arduino function		
reset	(PCINT14/RESET) PC6	1	28	PC5 (ADC5/SCL/PCINT13)
digital pin 0 (RX)	(PCINT16/RXD) PD0	2	27	PC4 (ADC4/SDA/PCINT12)
digital pin 1 (TX)	(PCINT17/TXD) PD1	3	26	PC3 (ADC3/PCINT11)
digital pin 2	(PCINT18/INT0) PD2	4	25	PC2 (ADC2/PCINT10)
digital pin 3 (PWM)	(PCINT19/OC2B/INT1) PD3	5	24	PC1 (ADC1/PCINT9)
digital pin 4	(PCINT20/XCK/T0) PD4	6	23	PC0 (ADC0/PCINT8)
VCC	VCC	7	22	GND
GND	GND	8	21	AREF
crystal	(PCINT6/XTAL1/TOSC1) PB6	9	20	AVCC
crystal	(PCINT7/XTAL2/TOSC2) PB7	10	19	PB5 (SCK/PCINT5)
digital pin 5 (PWM)	(PCINT21/OC0B/T1) PD5	11	18	PB4 (MISO/PCINT4)
digital pin 6 (PWM)	(PCINT22/OC0A/AIN0) PD6	12	17	PB3 (MOSI/OC2A/PCINT3)
digital pin 7	(PCINT23/AIN1) PD7	13	16	PB2 (SS/OC1B/PCINT2)
digital pin 8	(PCINT0/CLK0/ICP1) PB0	14	15	PB1 (OC1A/PCINT1)

Digital Pins 11, 12 & 13 are used by the ICSP header for MOSI, MISO, SCK connections (Atmega168 pins 17, 18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

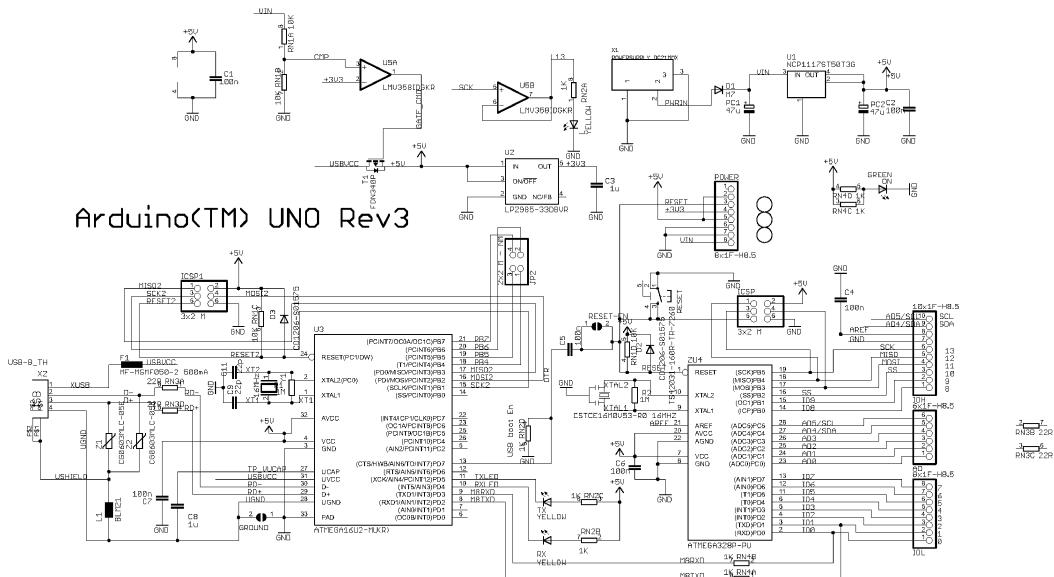
Figure A-5: Arduino ATmega pin mapping

Credit: Arduino, arduino.cc

Understanding the Arduino Schematic

Perhaps one of the best ways to learn about electrical design is to analyze the schematics of existing products, such as the Arduino. Figure A-6 shows the schematic for the Arduino Uno.

Can you match all the parts to the parts that you can see on your Arduino Uno? Start with the main ATmega328P MCU (Part ZU4 in the schematic) and all the breakout pins. Here, you can easily identify which ATmega ports or pins map to the pins that are available to you in the integrated development environment (IDE).



Reference Designs ARE PROVIDED "AS IS" AND "WITH ALL FAULTS. Arduino DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, REGARDING PRODUCTS, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Arduino may make changes to specifications and product descriptions at any time, without notice. The Customer must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Arduino reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The product information on the Web Site or Materials is subject to change without notice. Do not finalize a design with this information.

ARDUINO is a registered trademark.

Use of the ARDUINO name must be compliant with <http://www.arduino.cc/en/Main/Policy>

Figure A-6: Arduino Uno Rev3 schematic

Credit: Arduino, arduino.cc

Earlier in this appendix, you observed that PD0 and PD1 were connected to the USART TX and RX pins. In the Arduino schematic, you can indeed confirm that these pins connect to the corresponding pins on the 16U2 (which is employed as the USB-to-serial converter chip on the Uno). You also know that there is an LED connected (through a resistor) to Pin 13 of the Arduino. In the schematic, you can see that Pin 13 is connected to Pin PB5 on the ATmega. But where is the LED? By using net names, you can indicate an electrical connection between two points on a schematic without actually drawing all the lines. Having every wire shown in a schematic might get confusing very quickly. In the case of PB5, you can see that the wire coming out of the MCU is labeled *SCK*, and that there is a similarly labeled wire at the top of the schematic feeding through a buffer into a resistor and the familiar debug LED.

Most schematics that you'll find are done in a style similar to this one, with a lot of labeled nets that connect without direct wires. More complicated designs will contain many pages, with a hierarchical structure that connects nets on each page to each other. Continue to analyze the Arduino schematic until you understand where all the signals are going. See how many components you can match to the actual board.

If you are using an Adafruit METRO 328 instead of an Arduino Uno, then you can obtain the schematic for it at blum.fyi/metro-328-schematic and compare it to your board.

Index

Numbers and Symbols

32-bit architecture, 7

8-bit architecture, 7

9V batteries, 70

A

- AC (alternating current), 354–355
 - double insulated, 356
 - power transmission, 354
 - relays, 356–357
 - connecting to Arduino, 360–361
 - control, programming, 358–360
 - SPDT (Single Pole Double Throw), 357
 - wiring, 356
- accel.begin() function, 239
- accelerometers
 - analog, triple-axis, 56–57
 - audiovisual instrument, 241
 - hardware, 242
 - software, 242–246
 - data streaming, 241
 - datasheets, 231–233
 - description, 229–230
 - hardware
 - RGB LED, 235
 - setup, 233–235

MEMS (Micro Electro Mechanical Systems)

228–229

micro-machined, 231

sensor libraries, Adafruit, 236–241

single-axis accelerometer, 230

software, writing, 235–241

SPI bus, 228–229

Adafruit

- LED Backpack library, 443
- sensor libraries, 236–241

Adafruit Bluefruit LE Connect, 377

Adafruit Feather 32u4 Bluefruit LE, 14

Adafruit GFX library, 443

Adafruit LIS3DH Library, 236–237

Adafruit METRO 328, 4

drivers, 4

USB/serial interface, 172

Adafruit Unified Sensor Library

236–237

ADCs (analog-to-digital converters), 8, 50

Arduino Due, 12

pins, 9

ALS (ambient light sensor), 63

alternate.ino, 190

amplitude, 126

sound waves, 127

- analog accelerometers, triple-axis, 56–57
- analog input, analog output
 - control, 64–65
- analog output, controlling, 64–65
- analog sensors, 48, 56
 - potentiometers, reading, 51–55
 - Sharp infrared proximity sensor, 56
 - temperature sensor, 57–60
 - TMP36 temperature sensor, 56
- triple-axis analog
 - accelerometers, 56–57
- variable resistors and
 - analog input and analog output, 64–65
- resistive voltage dividers, 60–64
- analog signals, 48
 - ADC (analog-to-digital converter), 50
 - AM and FM modulation, 344
 - compared to digital, 48–49
 - converting to digital, 49–51
- analogRead() command, 51–60, 52, 54
- analogWrite() command,
 - 31–35, 64–65
- anodes, 25–26
- antenna, RF communications, 346–347
- APIs (application programming interface)
 - browser response, 432
 - call, successful, 439
 - keys, invalid, 436
 - structure, 430
 - weather, 428–429
 - data parsing, 431–433
 - JSON-formatted, 430–431
 - live temperature display, 440–449
 - local temperature, 433–439
 - server provider account, 429
 - structure, 430
 - web, interfacing with, 427–428
- applications, Processing, 161–162
- color selection, 168
- IDE, 163
- installation, 162
- sending data to Arduino, 166–169
- sketches
 - controlling, 162–166
 - examples, 165–166
- Arduino
 - boards, 4, 10–15
 - clones, 14
 - functionality, 5–10
 - software, 4
 - Wi-Fi, 404
- Arduino Cloud IDE, 15. *See also* IDE (integrated environment)
- Arduino Diecimila, 142
- Arduino Due, 7, 12, 13, 51, 126, 136, 148
- Arduino Duemilanove, 142
- Arduino Extreme, 142
- Arduino IDE, 4. *See also* IDE (integrated environment)
- Arduino Leonardo, 9, 12, 143, 145, 172, 358
- joystick mouse circuit, 179
- mouse emulation, 176
- USB and, 172
- Arduino Mega, 106, 142
- Arduino Mega 2560, 11
- Arduino Mega ADK, 148
- Arduino Micro, 12, 13
- Arduino Nano, 141, 142
 - FTDI chip, 144
- Arduino NG, 142
- Arduino Uno, 4, 11, 17
 - 16U2 serial converter chip, 144
- accelerometer breakdown, 231

ADC, 50
analog sensor, 57
ATmega 328P, 9
components, 6
enclosures, 3D-printable, 230
integrated circuit, 146
interface, 9
LEDs, wiring, 26
multiplexing, 205, 234
PWM pins, 32, 106
real-time clock, 317
receiver antenna, 347
resolution, 51
RGB LED wiring, 231
serial converter emulation, 146–147
serial port, 143
schematics, 443–459
SPI pins, 230
square wave frequency, 33
stepper motor wiring, 112
USB/serial interface, 172, 376
Arduino.cc, 5
 Arduino IDE download, 16
Arduino.org, 5
arguments, 20
Arm, 8
ARPA (Advanced Research Projects Agency), 401
ARPANET, 401
ASCII table, 155
asynchronous sketches, 276
AT commands, 371–372
ATmega 32U4, USB, 376
Atmel. *See* Microchip
Atmel ATmega, 5
attachInterrupt() function, 278–279
AVR architecture, 5, 7
AVRISP mkII, 141

B

bar graph
 decimal representations, 196
 distance-responsive, 196–197
bare_minimum_server.ino, 412–415
bargraph.ino, 195–196
batteries, 9V, 70
baud rate, 54
begin() function, 370–371
binary to decimal conversion, 192
bipolar stepper motors, 111–112
 NEMA-17, 109–110
BJT (bipolar junction transistor), 72
 NPN BJTs, 72, 73
BLE (Bluetooth Low Energy), 364
Blink, 18
 code, 18–21
blink.ino, 30–31
Bluetooth
 BLE (Bluetooth Low Energy), 364
 Bluetooth Classic, 365
 BTLE (Bluetooth Low Energy), 364
 GATT (Generic Attribute), 366
 IEEE standards, 364
 profiles, 365–366
 security, 395–396
 smart home lamp, 389
 pairing phone, 394–396
 pairing phone to BTLE
 device, 389–390
 proximity control software, 390–394
boards, 4
Adafruit
 LED Backpack library, 443
 sensor libraries, 236–241
 Adafruit Bluefruit LE Connect, 377
 Adafruit Feather 32u4 Bluefruit LE, 14

- Adafruit GFX library, 443
Adafruit LIS3DH Library, 236–237
Adafruit METRO 328, 3, 4
 drivers, 4
 USB/serial interface, 172
Adafruit Unified Sensor
 Library, 236–237
architectures, 5–7
Arduino Diecimila, 142
Arduino Due, 7, 12, 13, 51,
 126, 136, 148
Arduino Duemilanove, 142
Arduino Extreme, 142
Arduino Leonardo, 9, 12, 143,
 145, 172, 358
 joystick mouse circuit, 179
 mouse emulation, 176
 USB and, 172
Arduino Mega, 106, 142
Arduino Mega 2560, 11
Arduino Mega ADK, 148
Arduino Micro, 12, 13
Arduino Nano, 141, 142, 144
Arduino NG, 142
Arduino Uno, 3, 4, 10, 11, 17
 16U2 serial converter chip, 144
accelerometer breakdown, 231
ADC, 50
analog sensor, 57
ATmega 328P, 9
components, 6
enclosures, 3D-printable, 230
integrated circuit, 146
interface, 9
LEDs, wiring, 26
multiplexing, 205, 234
PWM pins, 32, 106
real-time clock, 317
receiver antenna, 347
resolution, 51
RGB LED wiring, 231
serial converter emulation, 146–147
serial port, 143
schematics, 443–459
SPI pins, 230
square wave frequency, 33
stepper motor wiring, 112
USB/serial interface, 172, 376
breadboards, 24–25
 buses, 25
 connections, 25
 LCD wired to Arduino, 251
 robot, 91
breakout boards, 228–229
clock pins, 8
connecting to, 17–18
counterfeit, 4
Feather, programming, 369–373
first-party, 4
LED, 17–18
oscillating crystal, 8
Particle Photon, 15
pins, 20
resonator, 8
third-party, 4, 12, 363–365
USB-host capabilities, 147–148
USB-to-serial converter, 143–146
Boolean variables, 41
bootloader, 9
 M0, 407
 setup, 10
breadboards, 24–25
 buses, 25
 connections, 25
 LCD wired to Arduino, 251
 robot, 91

breakout boards, 228–229
 brushed DC motors, 71
 brushless DC motors, 71
 stepper motors, 109
 BTLE (Bluetooth Low Energy), 364
 Feather board, potentiometer,
 366–367
 GATT (Generic Attribute), 366
 HID (Human Interface Device)
 profile, 390
 module library, 369
 natural language commands,
 384–389
 Nordic chip, 384
 smartphone connection, 376–379
 sending commands, 379–389
 BTLE_led.ino, 384–387, 392–394
 BTLE_sensor.ino, 373–376
 buses, power/ground, 25
 buttons
 bouncy, 38–42
 switch bouncing, 39

C

capacitors, decoupling, 88
 car.ino, 93–96
 cascaded shift registers, 191
 cathodes, 25–26
 CdS (cadmium sulfide)
 photoresistors, 62–63
 CERN (European Organization for
 Nuclear Research), 401
 changeKey() function, 293
 chronograph, 118
 programming, 119–124
 start button, 117
 stepper motor, 117
 stop button, 117

transform to clock, 324
 wiring diagram, 117
 chronograph.ino, 121–123
 circuits, 4, 27
 H-bridges, 79
 building, 80–81
 photoresistors, 63
 temperature and light sensor, 174
 voltage divider circuit, 61
 client-to-client communication, 412
 clients, 403–404
 CLK (clock), 227
 clock phase, 224
 clock polarity, 224
 clock signal (SCL), I²C bus, 203, 209
 clone boards, 14
 Cloud, 401
 code, comments
 multiline, 18
 single-line, 19
 command strings, serial
 monitor, 152–153
 commands
 arguments, 20
 natural language, 381–383
 comments
 multiline, 18
 single-line, 19
 communication buses, 8
 I²C bus, 204
 communications, pin connections, 250
 connect_to_wifi.ino, 410–411
 const int statement, 38
 constrain() functions, 64
 continuous rotation servo
 motor, 100–101
 control gloves, 136
 Cortex-M Series, 8

- counterfeit boards, 4
- CS (chip select), 227 *See also*
SS (Slave Select)
- pin, SD cards, SPI interface, 307
- CSV files, 297
- csv_logger.ino, 174–176
- D**
- DACs (digital-to-analog converters), 12, 31–32
- data logging
- CSV files, 297
 - entrance logger, 330–334
 - SD cards, 29
 - formatting, 298–303
 - shields, shield stacking headers, 306
 - spreadsheet, 312
- data streaming, accelerometers, 241
- dataFile.println() function, 309
- datasheets
- accelerometers, 231–233
 - ATmega328P, 452
 - chips, 453
 - I/O pins diagram, 454
 - pin-outs, 454–456
 - reading, 451–456
- DC (direct current), 354–355
- motors, 70–71
 - brushed DC motors, 71
 - direction, H-bridges, 78–86
 - inductive load, high current, 71–76
 - speed, PWM and, 76–78
 - stall current, 87
 - wiring, 74–76
 - debounce.ino, 40–41
 - debounce() function, 264
 - debugging, 424
 - decimal formats
- bar graph, 195
- converting from binary, 192
- decoupling capacitors, 88
- delay() function, 21
- device miniaturization, 228–229
- DHCP (Dynamic Host Configuration Protocol), 401
- address reservation, 426
- IP address retrieval, 409–412
- digital input
- bouncy buttons, 38–42
 - reading, pull-down resistors, 35–38
- digital output, 24
- programming, 29–30
 - for loops, 30–31
- digital pins, 20
- digital signals, 48
- ADC (analog-to-digital converter), 50
 - ASK modulation, 344
 - compared to analog, 48–49
 - converting from analog, 49–51
- digitalPinToInterrupt()
- function, 279
- digitalRead statement, 38
- digitalRead() function, 278, 348
- digitalWrite() function, 21, 29–30, 348
- diodes
- Light Emitting Diode (LEDs), 17–18
 - flybacks, 72
 - freewheeling, 72
 - protection, 73
 - snubbers, 72
- DIP (Dual-Inline Package), 25, 229
- display_temp.pde, 219
- distance sensor, 105–109
- distance-responsive bar graph, 196
- DNS (Domain Name System), 401
- updating service, 426–427

doorbell.ino, 350–351
 downloads, IDE (integrated environment), 16
 duty cycle, 33
 PWM signals and, 34

E

echo.ino, 154
 emulation
 mouse, 178–182
 USB keyboard, 173–176
 key combinations, 177–178
 entrance logger
 data analysis, 334
 hardware, 327–328
 IR distance sensor, 327
 shield stacking headers, 327
 software, 328–334
 change threshold, 329
 entrance_logger.ino, 329–333
 external hardware, 4
 programmers, 141–142
 external LED, pin 9, 24–25

F

fade.ino, 32
 FAT16, SD cards, 298
 FAT32, SD cards, 298
 Feather boards, 369–371
 BTLE board
 LED and, 380
 potentiometer, 366–367
 IDE setup, 406–407
 LED temperature display, 442
 programming, 369–371
 relay controller, 396
 smartphone connection, 376–379
 Wi-Fi

MAC address on module, 426
 server sketch, 408–423
 FireWire (IEEE 1394), 364
 firmware, setup, 10
 flybacks, 72
 for loops, 30–31
 freewheeling diodes, 72
 frequency, 126
 sound waves, 127
 frequency of signals *versus* period, 33
 FTDI USB-to-serial converter, 143–146
 fun_with_sound.ino, 292–293
 functions, 40
 accel.begin(), 239
 analogRead(), 52
 attachInterrupt(), 278–279
 begin(), 370–371
 changeKey(), 293
 constrain(), 64
 dataFile.println(), 309
 debounce(), 264
 delay(), 21
 digitalPinToInterrupt(), 279
 digitalRead(), 278, 348
 digitalWrite(), 348
 Keyboard.begin(), 176
 loop(), 29
 map(), 64
 millis(), 120, 348
 Mouse.press(), 181
 Mouse.release(), 181
 readJoystick(), 181
 RTC.adjust(), 319
 Serial.available(), 154
 Serial.begin(), 148
 Serial.println(), 52, 148
 setup(), 19–20, 54
 shiftOut(), 187–188

`tone()`, 129–136, 137, 265
`updateDateTime()`, 323
`void loop()`, 20
`void setup()`, 19
`waitForOK()`, 373
`Wire.beginTransmission()`, 213–214
`Wire.requestFrom()`, 213–214

G

GATT (Generic Attribute), 366
GET, 403
global variables, 41
GND (ground) connection, 56
digital ground, 356
Google Docs, 401
GPIO (general-purpose input/output), 8, 9
Great Arduino Schism and
 Reformation, 5
gyroscope, 229–230

H

H-bridges, 78–79
circuits
 building, 80–81
 operating, 81–82
operation states, 79
rate variables, 83
short circuits, 79
wiring diagram, 82
hardware
 accelerometers, 233–235
 circuits, 4
 external, 4
I/O, 404–405
programmers, 141–142
shields, 4
SPI bus, 225–227
hardware interrupts

accuracy, 278
capabilities, 278–279
hardware implementation, 277–278
hardware-debounced circuit, 280–284
 assembling, 285
software, 285–288
multitasking, 278
polling, 277–278
Schmitt triggers, 282
software implementation, 277
hbridge.ino, 85–86
HID (Human Interface Device)
 profile, 390
high-current inductive load, 71–72
motor wiring, 74–76
protection diodes, 73
secondary power source, 74
switches, transistors as, 72–73
HTML (Hypertext Markup
 Language), 402
HTTP (Hypertext Transfer
 Protocol), 402–403
requests, 413
response codes, 413
HTTPS (HTTP over Secure Sockets
 Layer), 403
hw_multitask.ino, 286–287
hysteresis, 282
I
I/O (Input/Output), 5, 8, 9
control hardware, 404–405
parallel data, 185
serial data, 185
I²C bus, 202
AD7414 addressing, 205
clock signal, 203
embedded program, 215–218

- hardware design, 203
 temperature monitoring system, 214–215
 hardware requirements, 206–208
 history, 202
 I/O pins, 206–207
 I²C protocol, 202
 ID numbers, 204
 part selection, 205–206
 Processing program, 218–221
 product design, 205–206
 pull-up resistors, 206–208
 SCL (clock signal), 203, 209
 slave devices, 202, 203–204
 communications, 204
 temperature sensor, 208, 213–214
 SPI comparison, 227–228
 TC74 address options, 204
 temperature probe, 208
 datasheet, 210–212
 hardware, 208–209
 serial output, 214
 shift register bar graph, 215
 software, writing, 212–214
 tuning over wires, 441
 ICs (integrated circuits), 25, 70, 201–202
 pins, 80–81
 Schmitt triggers, 282
 ICSP (In-Circuit Serial Programming), 8
 IDE (integrated environment), 4. *See also* Arduino IDE
 Arduino Cloud IDE, 15
 Blink, 18
 downloading, 16
 Feather board setup, 406–407
 header files, 130
 installing, 16
 port, 18
 running, 17–18
 servo control, 104–105
 third-party boards, 367–369
 IEEE (Institute of Electrical and Electronics Engineers), 364
 FireWire (IEEE 1394), 364
 POSIX (IEEE 1003), 364
 Power over Ethernet (IEEE 802.3), 364
 if() statements, 372
 if/else statements, 38
 IMU (inertial measurement unit), 245
 inductive load, high-current, 71–72
 motor wiring, 74–76
 protection diodes, 73
 secondary power source, 74
 switches, transistors as, 72–73
 inductors, 73
 installation, IDE (integrated environment), 16
 instrument.ino, 244–245
 Internet, 401
 interrupts
 polling, 277–278
 sound machine
 hardware, 291
 software, 291–293
 IP (Internet Protocol) address, 401–402
 retrieving, DHCP and, 409–412
 ipconfig, 425
 IR (infrared), distance sensor, 105–109
 IR LED, 105–109
 ISP (internet service provider), 365
- J**
 JSON (JavaScript Object Notation), 430
 library, 434

K

`Keyboard.begin()` function, 176

L

`lamp_remote.ino`, 358–360
 LCD (liquid crystal display), 248
 breadboard, 251
 headers, 249
 LiquidCrystal library, 251
 animations, 254–258
 special characters, 254–258
 text, adding, 252–254
 pins, parallel, 249–250
 setup, 245–251
 thermostat
 audible warning, 265–266
 custom characters, 263
 data display, 261–264
 fan, 265–266
 hardware, 258–261
 LCD_thermostat.ino, 266–270
 program, 266–270
 schematic, 259
 set point button, 264–265
 system diagram, 261
`LCD_progress_bar.ino`, 255–258
`LCD_text.ino`, 253
`lcd.print()` function, 253–254
 LED Backpack library (Adafruit), 443
`led_button.ino`, 38
`led.ino`, 29
 LEDs (light-emitting diodes), 17–18
 bar graph, 196
 external, pin 9, 24–25
 interrupts, hardware-debounced
 pushbutton, 279–288
 live temperature display, 440–442
 nightlight, building, 42–46

single characters controlling, 156–158
 wiring, 25–26

libraries

Adafruit, 236–241
 Adafruit GFX, 443
 BTLE module library, 369
 LiquidCrystal, 248
 RTCLib, 317–318
 timer interrupts, 288
 WiFi101, Wi-Fi module and, 407–408
`lightrider.ino`, 192–193
 linear regulators, 88
 power supply and, 89
Linux
 SD card formatting, 298–303
 serial ports, 18
 LiquidCrystal library, 248, 251
 animations, 254–258
 progress bar, 255
 special characters, 254–258
 text, adding, 252–254
`list_control.ino`, 159–161
 local network
 Arduino control, 423–424
 web and, 400
`lock_computer.ino`, 177–178
`loop()` function, 29

M

M0 bootloader, 407
 MAC addresses, 402
 Feather Wi-Fi module, 426
MacOS
 SD card formatting, 300–302
 serial ports, 18
`map()` function, 64
 MCU (microcontroller unit), 5–7
 Arm Cortex-M3 SAM3X, 12
 ATmega 328P, 9

Atmel ATmega, 5
 I²C bus, 202
 Microchip/Atmel ATmega2560, 11
 single-use USB, 147
 MEMS (Micro Electro Mechanical Systems), 228–229
 micro piano, 136–139
 tone() function, 137
 wiring diagram, 137
 Microchip, 7
 ATmega chip, 7
 millis() function, 120, 344
 MISO (master input, slave output), SD cards, SPI interface, 307
 MOSI (master output, slave input), SD cards, SPI interface, 307
 motion-based instrument wiring, 243
 motor_pot.ino, 78
 motors
 servo motors, 101
 stepper motors, 109
 brushed DC motors, 71
 wiring, 74–76
 mouse emulation, 178–182
 mouse.ino, 179–180
 Mouse.press() function, 181
 Mouse.release() function, 181
 multiline comments, 18
 multimeters, 53
 music.ino, 134–135

N

NAT (network address translation), 402
 natural languages commands, 381–383
 NEMA-17, bipolar motor, 109–110
 networks
 client-to-client communication, 412
 clients, 403–404
 the Cloud, 401

DHCP (Dynamic Host Configuration Protocol), 403
 address reservation, 426
 IP address retrieval, 409–412
 DNS (Domain Name System), 403
 GET, 403
 HTML (Hypertext Markup Language), 402
 HTTP (Hypertext Transfer Protocol), 402–403
 the Internet, 401
 IP address, 401–402
 local
 Arduino control, 423–424
 web and, 400
 MAC addresses, 402
 NAT (network address translation), 402
 port forwarding, 425–427, 426
 POST, 403
 router, login, 425
 servers, 403–404
 the web, 401
 nightlight LED project, 42–46
 nightlight.ino, 65
 NPN BJTs, 72, 73

O

Ohm's law, 27–28
 orientation.ino, 238–239

P

parallel data, shift register and, 185
 Particle Photon, 15
 period of signals, *versus* frequency, 33
 phone communication,
 potentiometer, 366–367
 photoresistors, 61–62
 CdS (cadmium sulfide), 62–63
 circuit, 63

- piano.ino, 138
- pin-outs, 454–456
- ping requests, 411–412
- pinMode() command, 20, 29
- pins, 20
 - active low, 185–186
 - ADC, 9
 - breadboards, 24–25
 - clock pins, 8
 - communications connections, 250
 - digital, 20
 - ICs (integrated circuits), 80–81
 - LCD, parallel, 249–250
 - serial clear pin, 187
 - servo motors, 101
 - dedicated control pin, 102
 - shift register, 185–186
- port forwarding, 426
 - Arduino control, 425–427
- portability, 20
- ports
 - IDE, 18
 - serial, 142–143
 - USB, 142–143
- POSIX (IEEE 1003), 364
- POST, 403
- pot_tabular.ino, 150–151
- pot_to_processing/arduino_read_pot, 163–164
- pot_to_processing/processing_display_color, 164–165
- pot.ino, 53, 149–150
- potentiometers
 - DC motor speed, 77–78
 - phone connection, 366–367
 - reading, 51–55
 - speakers, 127
 - wiring diagram, 149
- power, 5, 9
 - linear regulators, 89
- Power over Ethernet (IEEE 802.3), 364
- power sources, secondary, 74
- print statements, 148–150
- Processing, 161–162
 - color selection, 168
 - font creator, 218
 - IDE, 163
 - installation, 162
 - program writing, 218–219
 - sending data to Arduino, 166–169
 - sketches
 - controlling, 162–166
 - examples, 165–166
 - temperature sensor, 215–221
- processing_control_RGB/processing_control_RGB, 167–168
- programmers, 141–142
- programming, 29–30
 - functions, 40
 - loop() function, 29
 - for loops, 30–31
 - pinMode() function, 29
 - setup() function, 29
- sound
 - definition file, 129–130
 - tone() function, 129–136
- programming interfaces, 5
- ICSP (In-Circuit Serial Programming), 8
- portability, 20
- protection diodes, 73
- pull-down resistors, 35–36
 - strong pull-downs, 37
 - weak pull-downs, 37
- pushbutton input, pull-down resistors, 35–36

PWM (pulse-width modulation), 12, 24, 31–35, 71
`analogWrite()`, 31
 LED brightness control, 246
 DC motor speed control, 76–78
 thermostat fan, 270

R

rate variables, H-bridges, 83
 RC (radio control), cars, 71
`read_temp.ino`, 212–213
 reading digital inputs, pull-down resistors, 35–38
`readJoystick()` function, 181
 real-time clocks, 317–318
 communications, 317
 RTC module, 319
 RTCLib, 317–318
 SD card module, 319
 SD card test, 323
 software, 319
 refresh speed, 312–313
 register clock pin, 185
 resistance, 27
 resistors
 pull-up, 206–208
 current limiting, 21
 leakage, 37
 pull-down, 35
 RF communications
 AM (amplitude modulation), 344
 analog modulation, 344
 antenna, 346–347
 ASK modulation, 344
 carrier waves, 344
 digital modulation, 344
 electromagnetic spectrum, 336–337
 frequency, 338

ISM band, 339
 radio spectrum allocation, 338
 wavelengths, 338
 FM (frequency modulation), 344
 key presses, receiver connection, 346–347
 lamp, 354–361
 modulation, 344
 receiver module
 installed, 347
 programming, 347–351
 state variables, 350
 wireless doorbell, 351–353
`rf_test.ino`, 348–349
 sending/receiving data, 339–341
 serial output, RF test, 350
 smart home lamp, 354
 AC (alternating current), 355–357
 connecting to Arduino, 360–361
 relay connection, 360–361
 wireless doorbell
 receiver programming, 351–353
 receiver wiring, 351
`rf_test.ino`, 348–349
 RGB (Red, Green, Blue), 24
 RGB LED
 nightlight project, 42–46
 values controlling, 158–161
 robot
 breadboard, 91
 construction, 89–92
 electronics, 92
 parts, 86
 gearbox, 87
 motor, 87
 power, 87–89
 software, writing, 92–96
 rotors, 71
 routers, login, 425

`RTC.adjust()` function, 319
RTCLib, 317–318
 functions, 324
running mode, 176

S

schematics, 456
 ATmega pin mapping, 457
 Rev3, 456
Schmitt triggers, 282
SCK (clock), 227
SCLK (serial clock line), 224
 SD cards, SPI interface, 307
SD cards, 29
 formatting
 FAT16, 298
 FAT32, 298
 Linux, 303–303
 Mac OS, 300–302
 Windows, 298
 micro SD-to-SD adapter, 297
 reading from, 312–316
 real-time clock, 319
 refresh speed, 312–313
 reporting, 308–309
 debugging, output, 311
 shields, 304–307
 SPI interface, 307
 writing to, 307
sd_read_write_rtc.ino, 320–324
sd_read_write.ino, 314–316
SD.open() command, 310
SDI (serial data in), 227
SDO (serial data out), 227
secondary power sources, 74
sensor libraries, Adafruit, 236–241
sensors, analog, 48
serial clear pin, 187

serial clock, 227
serial communications, 142
 data type representations, 151–152
 desktop apps, 161–162
 Processing, 162–169
 incoming data, 153
 chars *versus* ints, 155–156
 echoing, 154–155
 single characters, LED and, 156–158
 values, RGB LED and, 158–161
 print statements, 148–150
 serial converter emulation, 146–147
 software, 215–216
 special characters, 150–151
 USB-to-serial converter, 143–146
serial data
 SDI (serial data in), 227
 SDO (serial data out), 227
 shift register and, 186–191
serial debugging, 424
serial interfaces, 8
 incoming serial data, 55
 serial monitor button, 55
 setup() function, 54
serial monitor
 Adafruit Bluefruit LE Connect, 377
 command strings, sending, 152–153
serial ports, 18, 142–143
 Virtual Serial Port, 143–146
Serial.available() function, 154
Serial.begin() function, 148
Serial.println() function, 52, 148
server_form.html, 417–418
servers, 403–404
servo motors
 color coding, 101
 continuous rotation, 100–101
 control, 101–104

- pins, 101–102
standard, compared to continuous rotation, 100–101
sweeping distance resistor, 105–109
timing, diagram, 102
`servo.ino`, 104–105
`setup()` function, 19–20, 54
Shaper Tools, stepper motors, 109
Sharp infrared proximity sensor, 56
shields, 4
 SD card shields, 304
 data logging, 306
 shield stacking headers, 308
shift registers, 183–184
 74HC595, 186–188
 animations, 192–197
 cascaded, 191
 eight-LED circuit diagram, 190
 input/output diagram, 185
 LED bar graph, 194–197
 light rider effect, 192–194
 overview, 187
 pin functions, 186–187
 register clock pin, 185
 serial clear pin, 187
 serial data, 189–191
 shifting value to, 187–191
 SIPO (serial in, parallel out), 185
`shiftOut()` function, 187–188
signals
 analog, 48
 digital, 48
 duty cycles, 34
Silicon Labs USB-to-serial converter, 143–146
`single_char_control.ino`, 156–157
single-axis accelerometer, 230
single-line comments, 19
sinusoidal voltage signal, 128
SIPO (serial in, parallel out) shift registers, 185
Slave Select (SS), 224
slave devices
 CS (chip select), 227 *see also* Slave Select (SS)
 I²C bus, 202, 203–204
 temperature sensor, 208, 213–214
 SPI bus, 224
SM Bus (System Management Bus), 202
smart home lamp
 AC (alternating current)
 digital ground, 356
 double insulated wire, 356
 power transmission, 354
 relay control programming, 360–361
 relays, 356–357
 wiring, 356
Bluetooth control, 389
 pairing phone, 394–396
 pairing phone to BTLE device, 389–390
 proximity control software, 390–394
 relays, connections to Arduino, 360–361
smartphone connection
 Bluetooth pairing
 Android, 394–395
 iPhone, 395–396
 BTLE transmitter connection, 376–379
 sending commands, 379–389
SMT (Surface Mount Technology), 229
snubbers, 72
software
 robot, 92–96
 serial communication, 215–216
 volatile variables, 286

- sound
 amplitude, 126
 frequency, 126
 micro piano, 136–139
 producing with speaker, 128
 programming
 definition file, 129–130
 tone() function, 129–136
 sequences, 132, 134–135
 arrays, 133–134
 square wave, 128
 tone() function, 129–136
 sound machine
 hardware, 291
 software, 291–293
 wiring diagram, 291
 sound waves
 amplitude, 127
 frequencies, 127
 SPDT (Single Pole Double Throw), 357
 speakers, 126
 cross section, 128
 magnet, 128
 potentiometer, 127
 sinusoidal voltage signal, 128
 sound production, 128
 wiring, 130–132
 diagram, 132
 speed, refresh, 312–313
 SPI (Serial Peripheral Interface)
 bus, 223–224
 accelerometer, 228–229
 BTLE SPI library, 370
 communication lines, 226
 communication modes, 225
 communication scheme, 227
 hardware
 configuration, 225–227
 pins, 234–235
 I²C comparison, 227–228
 IP address retrieval, 409–412
 naming conventions, 227
 SD cards, 306
 as slave devices, 224
 UART comparison, 227–228
 spinning coils, rotors, 71
 square wave, 128
 SSL (Secure Sockets Layer), 403
 stall current, DC motors, 87
 statements
 const int, 38
 digitalRead, 38
 if/else, 38
 stationary magnets, stator, 71
 stators, 71
 stepper motors, 109–110
 bipolar, 111–112
 NEMA-17, 109–110
 chronograph, 117
 movement flow chart, 111–112
 moving, 113–116
 wiring diagram, 115
 wiring schematic, 114
 stepper.ino, 115–116
 strong pull-downs, 37
 SudoGlove, 136
 sweep.ino, 107–109
 sweeping distance resistor, 105–109
 switch bouncing, 39
 switches, transistors as, 72–73
- T**
- TC74 sensor communication
 scheme, 210
 temp_unit.ino, 216
 tempalert.ino, 59–60
 temperature and light sensor circuit, 174
 temperature probe (I²C bus), 208

datasheet, 210–212
 hardware, 208–209
 building, 214–215
 serial output, 214
 shift register bar graph, 215
 software, writing, 212–214
 TC74 register information, 211
 TC74 sensor communication
 scheme, 210
 temperature sensor, 57–60
 third-party boards, Feather, 367–369
 timer interrupts, 288
 library, 289
 multitasking, 288–290
 timer1.ino, 289
 TMP36 temperature sensor, 56
 tone() function, 129–136, 265
 micro piano, 137
 transistors
 BJT (bipolar junction transistor), 72
 as switches, 72–73
 transmission, baud rate, 54
 triple-axis analog accelerometers,
 56–57

U

UART
 Serial UART, 143,223
 Nordic BTLE chip, 384
 SPI comparison, 227–228
 updateDateTime() function, 323
 USART (Universal Synchronous/
 Asynchronous Receiver/
 Transmitter), 9, 141–142
 USB devices, 172
 ATmega 32U4, 376
 keyboard emulation, typing,
 173–176
 Leonardo and, 172

USB interfaces, 8
 USB ports, 142–143
 boards with host capabilities,
 147–148
 single USB-capable MCU, 147
 USB-to-serial converter, 143–146
 emulation, 146–147

V

variable resistors
 photoresistors, 61–62
 CdS (cadmium sulfide), 62–63
 resistive voltage dividers, 60–64
 variables
 Boolean, 41
 global, 41
 volatile, 286
 VCC, 56
 Virtual Serial Port, 143–146
 void loop() function, 20
 void setup() function, 19
 volatile variables, 286
 voltage, 27
 divider circuits, 61

W

waitForOK() function, 373
 weak pull-downs, 37
 weather API, 428–429
 data parsing, 431–433
 JSON-formatted, 430–431
 live temperature display, 440
 LED readout wiring, 440–442
 temperature data, 443–449
 local temperature, 433–439
 server provider account, 429
 structure, 430
 web, 401
 web page design, 416–418

- web server
 - bare-minimum, 412–415
 - sketch, 414–419
 - web_control_server.ino, 419–423
 - web_weather_display.ino, 445–448
 - web_weather.ino, 436–439
 - Wi-Fi
 - Arduino, 404
 - Feather board, WINC1500
 - library, 407–408
 - IEEE 802.11, 364
 - ping requests, 411–412
 - server sketch, 408–423
 - WiFi101 library, Wi-Fi module
 - and, 407–408
 - WINC1500 library, 407–408
 - Windows, SD card formatting, 298–300
 - Wire library, 209–210
 - Wire.beginTransmission()
 - function, 213–214
 - Wire.requestFrom() function,
 - 213–214
 - wireless connectivity. *See also* Bluetooth
 - wiring
 - anodes, 25–26
 - cathodes, 25–26
 - H-bridges, 82
 - motion-based, 243
 - motors, 74–76
 - speakers, 130–132
 - write_to_sd.ino, 307–309
- X-Y-Z**
- XBee radios, 336