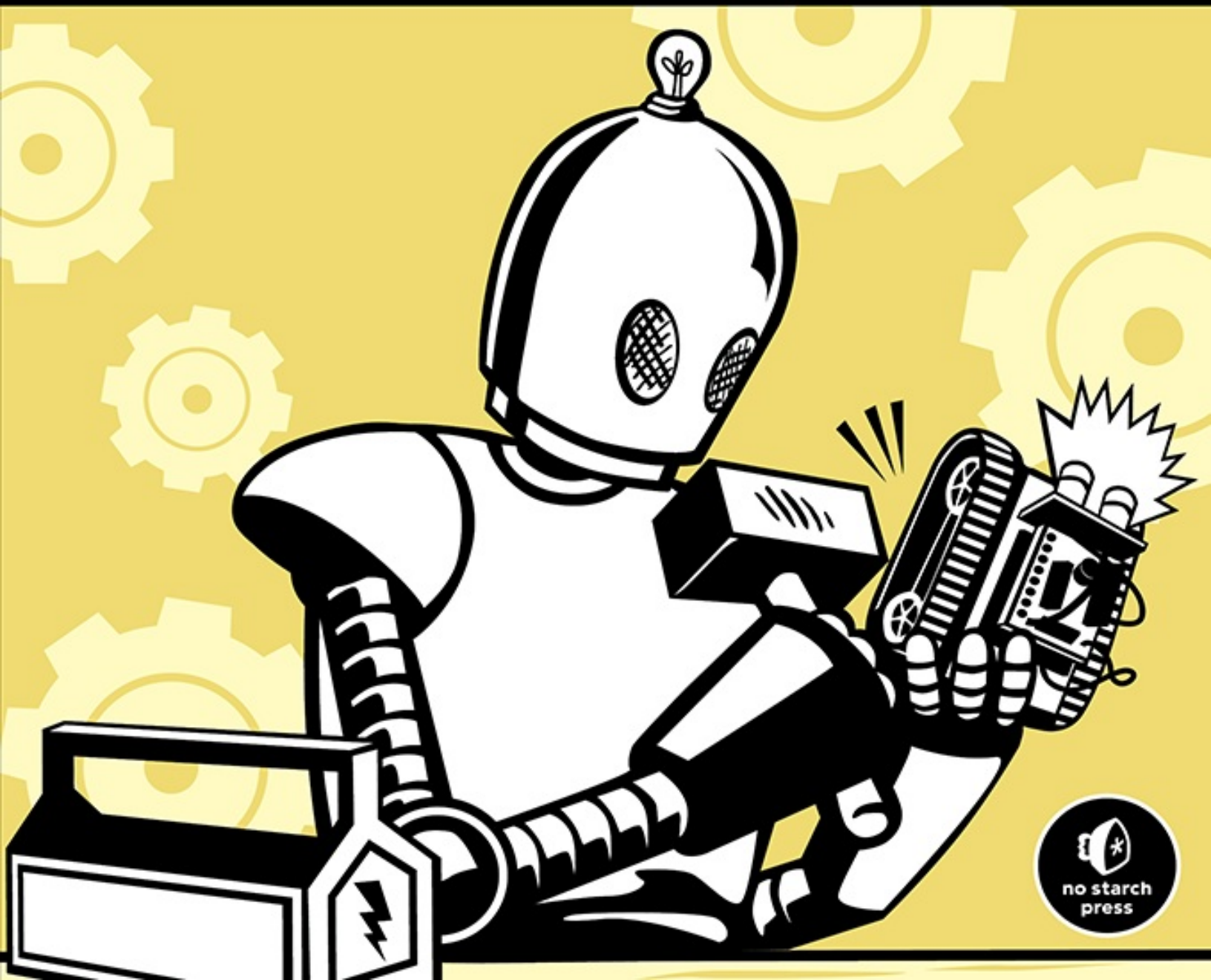


SECOND EDITION

ARDUINO WORKSHOP

A HANDS-ON INTRODUCTION
WITH 65 PROJECTS

JOHN BOXALL



CONTENTS IN DETAIL

REVIEWS FOR THE FIRST EDITION OF ARDUINO WORKSHOP

TITLE PAGE

COPYRIGHT

DEDICATION

ABOUT THE AUTHOR

ACKNOWLEDGMENTS

CHAPTER 1: GETTING STARTED

The Possibilities Are Endless

Strength in Numbers

Parts and Accessories

Required Software

macOS

Windows 10

Ubuntu Linux

Using Arduino Safely

Looking Ahead

CHAPTER 2: EXPLORING THE ARDUINO BOARD AND THE IDE

The Arduino Board

Taking a Look Around the IDE

The Command Area

[The Text Area](#)

[The Output Window](#)

[Creating Your First Sketch in the IDE](#)

[Comments](#)

[The setup\(\) Function](#)

[Controlling the Hardware](#)

[The loop\(\) Function](#)

[Verifying Your Sketch](#)

[Uploading and Running Your Sketch](#)

[Modifying Your Sketch](#)

[Looking Ahead](#)

[CHAPTER 3: FIRST STEPS](#)

[Planning Your Projects](#)

[About Electricity](#)

[Current](#)

[Voltage](#)

[Power](#)

[Electronic Components](#)

[The Resistor](#)

[The Light-Emitting Diode](#)

[The Solderless Breadboard](#)

[Project #1: Creating a Blinking LED Wave](#)

[The Algorithm](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Running the Sketch](#)

[Using Variables](#)

[Project #2: Repeating with for Loops](#)

[Varying LED Brightness with Pulse-Width Modulation](#)

[Project #3: Demonstrating PWM](#)

[More Electric Components](#)

[The Transistor](#)

[The Rectifier Diode](#)

[The Relay](#)

[Higher-Voltage Circuits](#)

[Looking Ahead](#)

[CHAPTER 4: BUILDING BLOCKS](#)

[Using Schematic Diagrams](#)

[Identifying Components](#)

[Wires in Schematics](#)

[Dissecting a Schematic](#)

[The Capacitor](#)

[Measuring the Capacity of a Capacitor](#)

[Reading Capacitor Values](#)

[Types of Capacitors](#)

[Digital Inputs](#)

[Project #4: Demonstrating a Digital Input](#)

[The Algorithm](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Modifying Your Sketch: Making More Decisions with if-else](#)

[Boolean Variables](#)

[Comparison Operators](#)

[Making Two or More Comparisons](#)

[Project #5: Controlling Traffic](#)

[The Goal](#)

[The Algorithm](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Running the Sketch](#)

[Analog vs. Digital Signals](#)

[Project #6: Creating a Single-Cell Battery Tester](#)

[The Goal](#)

[The Algorithm](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Doing Arithmetic with an Arduino](#)

[Float Variables](#)

[Comparison Operators for Calculations](#)

[Improving Analog Measurement Precision with a Reference Voltage](#)

[Using an External Reference Voltage](#)

[Using the Internal Reference Voltage](#)

[The Variable Resistor](#)

[Piezoelectric Buzzers](#)

[Piezo Schematic](#)

[Project #7: Trying Out a Piezo Buzzer](#)

[Project #8: Creating a Quick-Read Thermometer](#)

[The Goal](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Looking Ahead](#)

CHAPTER 5: WORKING WITH FUNCTIONS

[Project #9: Creating a Function to Repeat an Action](#)

[Project #10: Creating a Function to Set the Number of Blinks](#)

[Creating a Function to Return a Value](#)

[Project #11: Creating a Quick-Read Thermometer That Blinks the Temperature](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Displaying Data from the Arduino in the Serial Monitor](#)

[The Serial Monitor](#)

[Project #12: Displaying the Temperature in the Serial Monitor](#)

[Debugging with the Serial Monitor](#)

[Making Decisions with while Statements](#)

[while](#)

[do-while](#)

[Sending Data from the Serial Monitor to the Arduino](#)

[Project #13: Multiplying a Number by Two](#)

[long Variables](#)

[Project #14: Using long Variables](#)

[Looking Ahead](#)

CHAPTER 6: NUMBERS, VARIABLES, AND ARITHMETIC

[Generating Random Numbers](#)

[Using Ambient Current to Generate a Random Number](#)

[Project #15: Creating an Electronic Die](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Modifying the Sketch](#)

[A Quick Course in Binary](#)

[Binary Numbers](#)

[Byte Variables](#)

[Increasing Digital Outputs with Shift Registers](#)

[Project #16: Creating an LED Binary Number Display](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Project #17: Making a Binary Quiz Game](#)

[The Algorithm](#)

[The Sketch](#)

[Arrays](#)

[Defining an Array](#)

[Referring to Values in an Array](#)

[Writing to and Reading from Arrays](#)

[Seven-Segment LED Displays](#)

[Controlling the LED](#)

[Project #18: Creating a Single-Digit Display](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Modifying the Sketch: Displaying Double Digits](#)

[Project #19: Controlling Two Seven-Segment LED Display Modules](#)

[The Hardware](#)

[The Schematic](#)

[Modulo](#)

[Project #20: Creating a Digital Thermometer](#)

[The Hardware](#)

[The Sketch](#)

[Looking Ahead](#)

[CHAPTER 7: EXPANDING YOUR ARDUINO](#)

[Shields](#)

[ProtoShields](#)

[Project #21: Creating a Custom Shield](#)

[The Hardware](#)

[The Schematic](#)

[The Layout of the ProtoShield Board](#)

[The Design](#)

[Soldering the Components](#)

[Testing Your ProtoShield](#)

[Expanding Sketches with Libraries](#)

[Downloading an Arduino Library as a ZIP File](#)

[Importing an Arduino Library with Library Manager](#)

[SD Memory Cards](#)

[Connecting the Card Module](#)

[Testing Your SD Card](#)

[Project #22: Writing Data to the Memory Card](#)

[The Sketch](#)

[Project #23: Creating a Temperature-Logging Device](#)

[The Hardware](#)

[The Sketch](#)

[Timing Applications with millis\(\) and micros\(\)](#)

[Project #24: Creating a Stopwatch](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Interrupts](#)

[Interrupt Modes](#)

[Configuring Interrupts](#)

[Activating or Deactivating Interrupts](#)

[Project #25: Using Interrupts](#)

[The Sketch](#)

[Looking Ahead](#)

[CHAPTER 8: LED NUMERIC DISPLAYS AND MATRICES](#)

[LED Numeric Displays](#)

[Installing the Library](#)

[Project #26: Digital Stopwatch](#)

[Project #27: Using LED Matrix Modules](#)

[Installing the Library](#)

[Editing the Display Font](#)

[Looking Ahead](#)

[CHAPTER 9: LIQUID CRYSTAL DISPLAYS](#)

[Character LCD Modules](#)

[Using a Character LCD in a Sketch](#)

[Displaying Text](#)

[Displaying Variables or Numbers](#)

[Project #28: Defining Custom Characters](#)

[Graphic LCD Modules](#)

[Connecting the Graphic LCD](#)

[Using the LCD](#)

[Controlling the Display](#)

[Project #29: Seeing the Text Functions in Action](#)

[The Sketch](#)

[Running the Sketch](#)

[Creating More Complex Display Effects with Graphic Functions](#)

[Project #30: Seeing the Graphic Functions in Action](#)

[The Sketch](#)

[Project #31: Creating a Temperature History Monitor](#)

[The Algorithm](#)

[The Hardware](#)

[The Sketch](#)

[Running the Sketch](#)

[Modifying the Sketch](#)

[Looking Ahead](#)

[CHAPTER 10: CREATING YOUR OWN ARDUINO LIBRARIES](#)

[Creating Your First Arduino Library](#)

[Anatomy of an Arduino Library](#)

[The Header File](#)

[The Source File](#)

[The KEYWORDS.TXT File](#)

[Installing Your New Arduino Library](#)

[Creating a ZIP File Using Windows 7 and Later](#)

[Creating a ZIP File Using Mac OS X or Later](#)

[Installing Your New Library](#)

[Creating a Library That Accepts Values to Perform a Function](#)

[Creating a Library That Processes and Displays Sensor Values](#)

[Looking Ahead](#)

CHAPTER 11: NUMERIC KEYPADS

[Using a Numeric Keypad](#)

[Wiring a Keypad](#)

[Programming for the Keypad](#)

[Testing the Sketch](#)

[Making Decisions with switch case](#)

[Project #32: Creating a Keypad-Controlled Lock](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Testing the Sketch](#)

[Looking Ahead](#)

CHAPTER 12: ACCEPTING USER INPUT WITH TOUCHSCREENS

[Touchscreens](#)

[Connecting the Touchscreen](#)

[Project #33: Addressing Areas on the Touchscreen](#)

[The Hardware](#)

[The Sketch](#)

[Testing the Sketch](#)

[Mapping the Touchscreen](#)

[Project #34: Creating a Two-Zone On/Off Touch Switch](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Testing the Sketch](#)

[Using the map\(\) Function](#)

[Project #35: Creating a Three-Zone Touch Switch](#)

[The Touchscreen Map](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Looking Ahead](#)

[CHAPTER 13: MEET THE ARDUINO FAMILY](#)

[Project #36: Creating Your Own Breadboard Arduino](#)

[The Hardware](#)

[The Schematic](#)

[Running the Sketch](#)

[The Many Arduino and Alternative Boards](#)

[Arduino Uno](#)

[Freetronics Eleven](#)

[The Adafruit Pro Trinket](#)

[The Arduino Nano](#)

[The LilyPad](#)

[The Arduino Mega 2560](#)

[The Freetronics EtherMega](#)

[The Arduino Due](#)

[Looking Ahead](#)

[CHAPTER 14: MOTORS AND MOVEMENT](#)

[Making Small Motions with Servos](#)

[Selecting a Servo](#)

[Connecting a Servo](#)

[Putting a Servo to Work](#)

[Project #37: Building an Analog Thermometer](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Using Electric Motors](#)

[Selecting a Motor](#)

[The TIP120 Darlington Transistor](#)

[Project #38: Controlling the Motor](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Using Small Stepper Motors](#)

[Project #39: Building and Controlling a Robot Vehicle](#)

[The Hardware](#)

[The Schematic](#)

[Connecting the Motor Shield](#)

[The Sketch](#)

[Connecting Extra Hardware to the Robot](#)

[Sensing Collisions](#)

[Project #40: Detecting Robot Vehicle Collisions with a Microswitch](#)

[The Schematic](#)

[The Sketch](#)

[Infrared Distance Sensors](#)

[Wiring It Up](#)

[Testing the IR Distance Sensor](#)

[Project #41: Detecting Robot Vehicle Collisions with an IR Distance Sensor](#)

[The Sketch](#)

[Modifying the Sketch: Adding More Sensors](#)

[Ultrasonic Distance Sensors](#)

[Connecting the Ultrasonic Sensor](#)

[Testing the Ultrasonic Sensor](#)

[Project #42: Detecting Collisions with an Ultrasonic Distance Sensor](#)

[The Sketch](#)

[Looking Ahead](#)

[CHAPTER 15: USING GPS WITH YOUR ARDUINO](#)

[What Is GPS?](#)

[Testing the GPS Shield](#)

[Project #43: Creating a Simple GPS Receiver](#)

[The Hardware](#)

[The Sketch](#)

[Running the Sketch](#)

[Project #44: Creating an Accurate GPS-Based Clock](#)

[The Hardware](#)

[The Sketch](#)

[Project #45: Recording the Position of a Moving Object over Time](#)

[The Hardware](#)

[The Sketch](#)

[Running the Sketch](#)

[Looking Ahead](#)

[CHAPTER 16: WIRELESS DATA](#)

[Using Low-Cost Wireless Modules](#)

[Project #46: Creating a Wireless Remote Control](#)

[The Transmitter Circuit Hardware](#)

[The Transmitter Schematic](#)

[The Receiver Circuit Hardware](#)

[The Receiver Schematic](#)

[The Transmitter Sketch](#)

[The Receiver Sketch](#)

[Using LoRa Wireless Data Modules for Greater Range and Faster Speed](#)

[Project #47: Remote Control over LoRa Wireless](#)

[The Transmitter Circuit Hardware](#)

[The Transmitter Schematic](#)

[The Receiver Circuit Hardware](#)

[The Receiver Schematic](#)

[The Transmitter Sketch](#)

[The Receiver Sketch](#)

[Project #48: Remote Control over LoRa Wireless with Confirmation](#)

[The Transmitter Circuit Hardware](#)

[The Transmitter Schematic](#)

[The Transmitter Sketch](#)

[The Receiver Sketch](#)

[Project #49: Sending Remote Sensor Data Using LoRa Wireless](#)

[The Transmitter Circuit Hardware](#)

[The Receiver Circuit Hardware](#)

[The Receiver Schematic](#)

[The Transmitter Sketch](#)

[The Receiver Sketch](#)

[Looking Ahead](#)

CHAPTER 17: INFRARED REMOTE CONTROL

[What Is Infrared?](#)

[Setting Up for Infrared](#)

[The IR Receiver](#)

[The Remote Control](#)

[A Test Sketch](#)

[Testing the Setup](#)

[Project #50: Creating an IR Remote Control Arduino](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Modifying the Sketch](#)

[Project #51: Creating an IR Remote Control Robot Vehicle](#)

[The Hardware](#)

[The Sketch](#)

[Looking Ahead](#)

CHAPTER 18: READING RFID TAGS

[Inside RFID Devices](#)

[Testing the Hardware](#)

[The Schematic](#)

[Testing the Schematic](#)

[The Test Sketch](#)

[Displaying the RFID Tag ID Number](#)

[Project #52: Creating a Simple RFID Control System](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Storing Data in the Arduino's Built-in EEPROM](#)

[Reading and Writing to the EEPROM](#)

[Project #53: Creating an RFID Control with “Last Action” Memory](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Looking Ahead](#)

CHAPTER 19: DATA BUSES

[The I2C Bus](#)

[Project #54: Using an External EEPROM](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Running the Sketch](#)

[Project #55: Using a Port Expander IC](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[The SPI Bus](#)

[Pin Connections](#)

[Implementing the SPI](#)

[Sending Data to an SPI Device](#)

[Project #56: Using a Digital Rheostat](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Looking Ahead](#)

CHAPTER 20: REAL-TIME CLOCKS

[Connecting the RTC Module](#)

[Project #57: Adding and Displaying Time and Date with an RTC](#)

[The Hardware](#)

[The Sketch](#)

[Understanding and Running the Sketch](#)

[Project #58: Creating a Simple Digital Clock](#)

[The Hardware](#)

[The Sketch](#)

[Understanding and Running the Sketch](#)

[Project #59: Creating an RFID Time-Clock System](#)

[The Hardware](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Looking Ahead](#)

[CHAPTER 21: THE INTERNET](#)

[What You'll Need](#)

[Project #60: Building a Remote Monitoring Station](#)

[The Hardware](#)

[The Sketch](#)

[Troubleshooting](#)

[Understanding the Sketch](#)

[Project #61: Creating an Arduino Tweeter](#)

[The Hardware](#)

[The Sketch](#)

[Controlling Your Arduino from the Web](#)

[Project #62: Setting Up a Remote Control for Your Arduino](#)

[The Hardware](#)

[The Sketch](#)

[Controlling Your Arduino Remotely](#)

[Looking Ahead](#)

[CHAPTER 22: CELLULAR COMMUNICATIONS](#)

[The Hardware](#)

[Hardware Configuration and Testing](#)

[Project #63: Building an Arduino Dialer](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Project #64: Building an Arduino Texter](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Project #65: Setting Up an SMS Remote Control](#)

[The Hardware](#)

[The Schematic](#)

[The Sketch](#)

[Understanding the Sketch](#)

[Looking Ahead](#)

[INDEX](#)

REVIEWS FOR THE FIRST EDITION OF *ARDUINO WORKSHOP*

“When it comes to technology, there’s really something to be said for learning by example, and with each key point focused around a specific project, the information in this book is easy to learn and retain.”

—DAVE RANKIN, ABOUT.COM OPEN SOURCE

“*Arduino Workshop* was the first book I’ve read that helped me really make sense of the practical applications the Arduino is capable of.”

—AMATEURRADIO.COM

“A very thorough primer for those wishing to jump on the [Arduino] bandwagon.”

—KEVIN WIERZBICKI, CAMPUS CIRCLE

“I’ve checked out several Arduino ‘primers,’ and found the best one for my purposes to be *Arduino Workshop: A Hands-on Introduction with 65 Projects* by John Boxall.”

—JEFF ROWE, MCADCAFE.COM BLOG

“A good book for getting started . . . I highly recommend it if you’re thinking about getting into Arduino projects and you’re brand new to this stuff.”

—NATHAN YAU, FLOWINGDATA

ARDUINO WORKSHOP

2nd Edition

A Hands-on Introduction with 65 Projects

John Boxall



**no starch
press**

San Francisco

ARDUINO WORKSHOP, 2ND EDITION. Copyright © 2021 by John Boxall.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-7185-0058-7 (print)

ISBN-13: 978-1-7185-0059-4 (ebook)

Publisher: William Pollock

Executive Editor: Barbara Yien

Production Editor: Rachel Monaghan

Developmental Editors: Patrick DiJusto and Nathan Heidelberger

Cover Illustration: Charlie Wylie

Interior Design: Octopod Studios

Technical Reviewer: Xander Soldaat

Copyeditor: Paula L. Fleming

Compositor: Happenstance Type-O-Rama

Proofreader: Rachel Head

Indexer: JoAnne Burek

Circuit diagrams made using Fritzing (<http://fritzing.org/>).

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 1-415-863-9900; info@nostarch.com

www.nostarch.com

The Library of Congress issued the following Cataloging-in-Publication Data for the first edition:

Boxall, John, 1975-

Arduino workshop : a hands-on introduction with 65 projects / by John Boxall.

pages cm

Includes index.

ISBN-13: 978-1-59327-448-1

ISBN-10: 1-59327-448-3

1. Arduino (Microcontroller) I. Title.

TJ223.P76B68 2013

629.8'95--dc23

2013008261

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

For the two
people who
have always
believed in me:
my mother and
my dearest
Kathleen

About the Author

John Boxall has been in the electronics design, distribution, and e-commerce field for over 26 years. He has lately been writing Arduino tutorials, projects, and kit reviews during his spare time.

About the Technical Reviewer

Xander Soldaat is a former Mindstorms Community Partner for LEGO MINDSTORMS. He was an IT infrastructure architect and engineer for 18 years before becoming a full-time software developer, first for Robomatter and VEX Robotics and now as an R&D engineer for an embedded Wi-Fi solutions provider. In his spare time, he likes to tinker with robots, 3D printing, and home-built retro-computers.

ACKNOWLEDGMENTS

First of all, a huge thank you to the Arduino team: Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis. Without your vision, thought, and hard work, none of this would have been possible.

Thank you to all the readers of the first edition for your feedback and constructive criticism.

Many thanks to my technical reviewer, Xander Soldaat, for his contributions and for having the tenacity to follow through with such a large project.

I also want to thank the following organizations for their images and encouragement: Adafruit, Keysight Technologies, Freetronics, PMD Way, Seeed Studio, Sharp Corporation, SparkFun, and Tronixlabs.

Furthermore, a big thanks to Freetronics and PMD Way for the use of their excellent hardware products. And thank you to all those who have contributed their time making Arduino libraries, which makes life much easier for everyone. Kudos and thanks to the Fritzing team for their open source circuit schematic design tool, which I've used throughout this book.

Thanks also to the following people (in no particular order) from whom I've received encouragement, inspiration, and support: Elizabeth Pryce, Jonathan Oxer, Philip Lindsay, Ken Shirriff, Nathan Kennedy, and David L. Jones.

Finally, thank you to everyone at No Starch Press for their efforts in this updated edition, including Patrick DiJusto for his editorial input, Dapinder Dosanjh and Nathan Heidelberger for their endless patience, Rachel Monaghan for guiding the book through the production process, Paula Fleming for copyediting, Rachel Head for proofreading, JoAnne Burek for indexing, and of course Bill Pollock for his support and guidance and for convincing me that sometimes there is a better way to explain something.

1

GETTING STARTED

Have you ever looked at some gadget and wondered how it *really* worked? Maybe it was a remote control boat, an elevator, a vending machine, or an electronic toy. Or have you wanted to create your own robot or make electronic signals for a model railroad? Or perhaps you'd like to capture and analyze weather data over time? Where and how do you start?

The Arduino microcontroller board (shown in [Figure 1-1](#)) can help you find the answers to some of the mysteries of electronics in a hands-on way. The original creation of Massimo Banzi and David Cuartielles, the Arduino system offers an inexpensive way to build interactive projects, such as remote-controlled robots, GPS tracking systems, and electronic games.

The Arduino project has grown exponentially since its introduction in 2005. It's now a thriving industry, supported by a community of people united with the common bond of creating something new. You'll find individuals and groups ranging from small clubs to local hackerspaces to educational institutions, all interested in trying to make things with the Arduino.

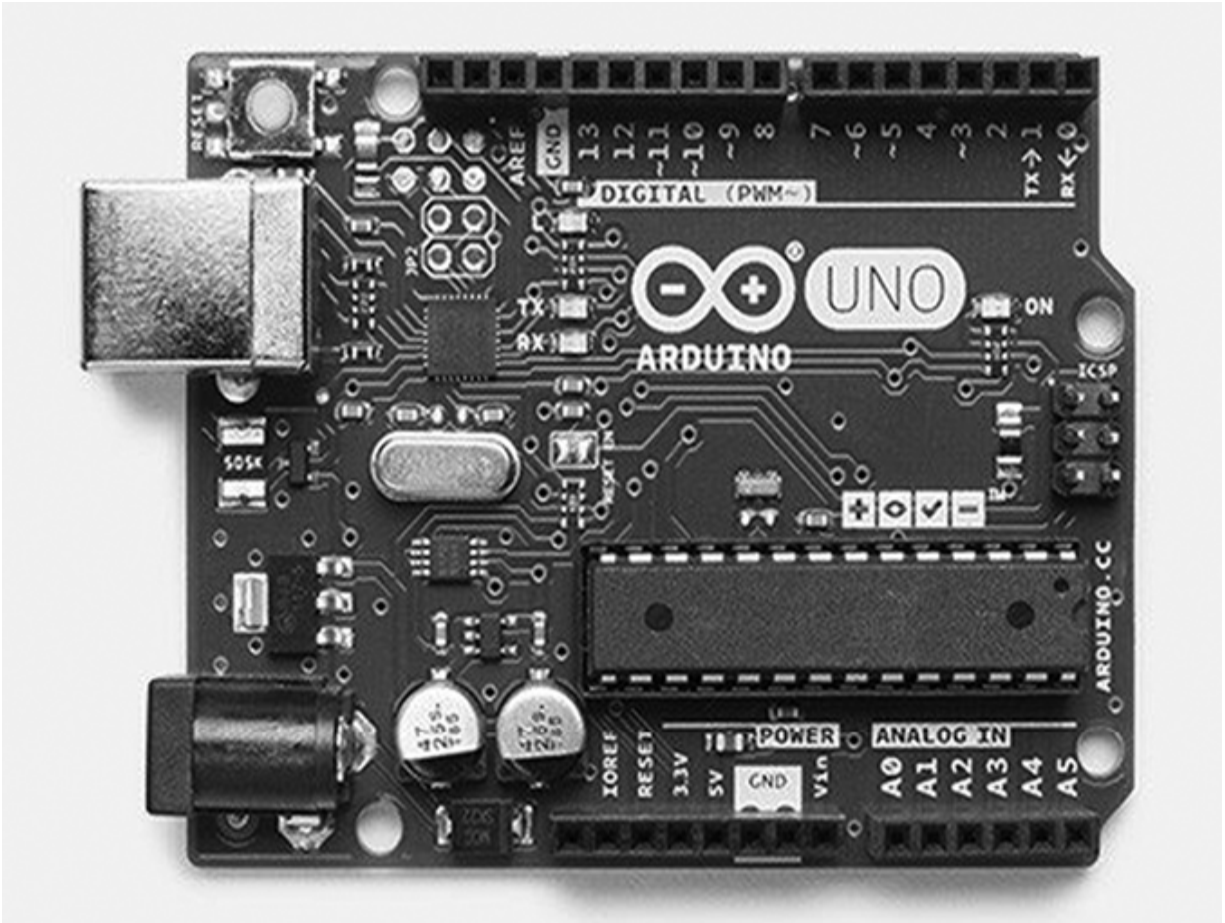


Figure 1-1: The Arduino board

To get a sense of the variety of Arduino projects in the wild, simply search the internet. There, you'll find an incredible number of projects, blogs, experiences, and ideas that show what is possible with the Arduino.

The Possibilities Are Endless

A quick scan through this book will show you that you can use the Arduino to do something as simple as blinking a small light or something as complicated as interacting with a cellular phone—and many different things in between.

For example, look at Becky Stern's Wi-Fi Weather Display, different examples of which are shown in [Figure 1-2](#). It uses an Arduino-compatible board with a Wi-Fi interface to receive the local weather forecast. It then

displays the daily maximum temperature and illuminates a colored triangle to represent the weather forecast for the day.



Figure 1-2: Various examples of a weather forecast display device

Thanks to the ease of interrogating various internet-based information services, you can use this to display data other than the weather. For more information, visit <https://www.instructables.com/id/WiFi-Weather-Display-With-ESP8266/>.

How about reproducing a classic computer from the past? Thanks to the power of the Arduino's internal processor, you can emulate computers from days gone by. One example is Oscar Vermeulen's KIM Uno, shown in [Figure 1-3](https://en.wikipedia.org/wiki/KIM-1), which emulates the 1976 KIM-1 computer. Visit <https://en.wikipedia.org/wiki/KIM-1> to learn more.

an Arduino, mechanisms from old CD drives, and other inexpensive items to create a computer numerical control (CNC) device that can draw with precision on a flat surface (see [Figure 1-4](#)). For more information, visit <http://www.ardumotive.com/new-cnc-plotter.html>.

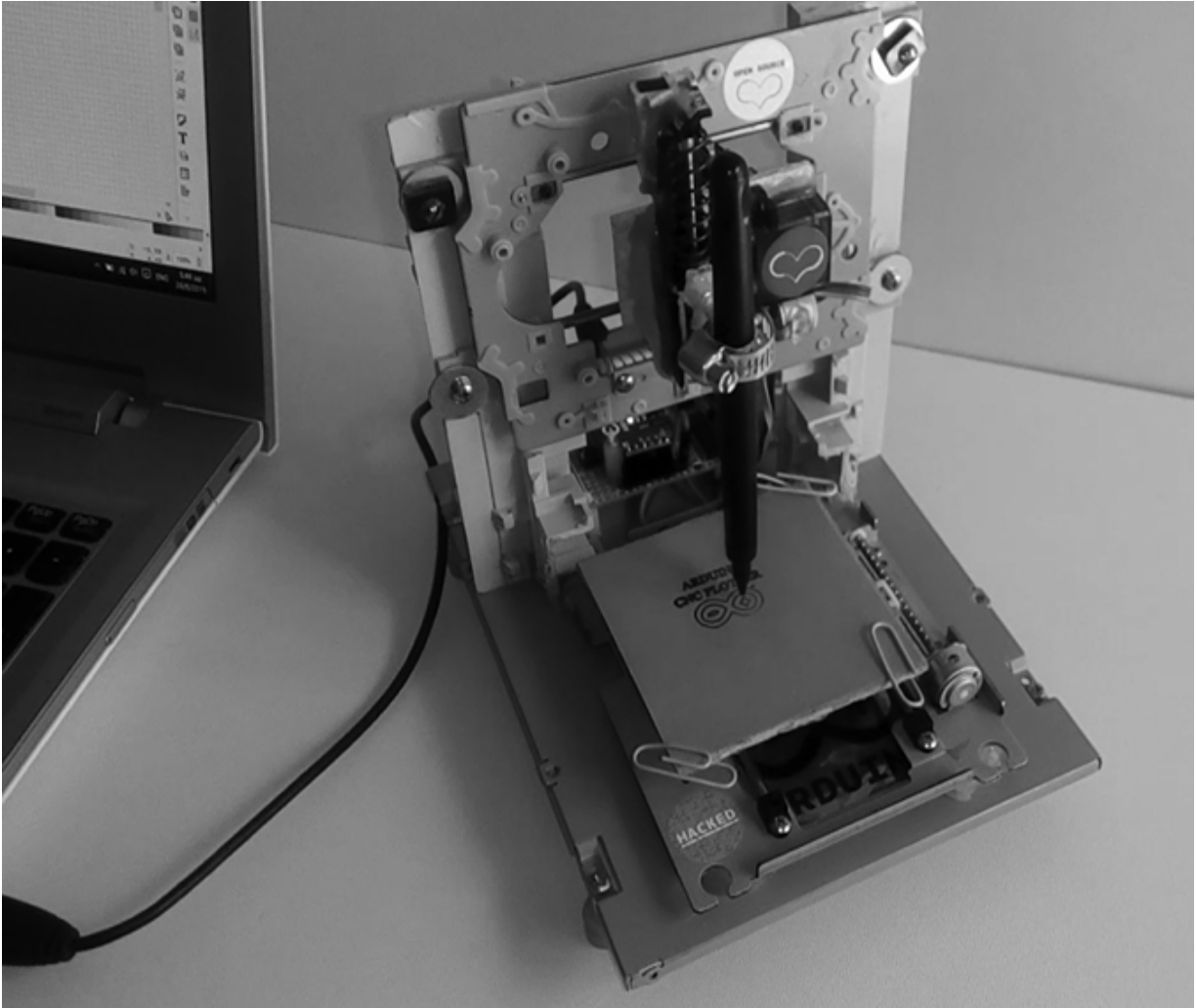


Figure 1-4: The Arduino Mini CNC Plotter

These are only a few random examples of what is possible using an Arduino. You can create your own simple projects without much difficulty—and after you’ve worked through this book, even more complex projects will certainly be within your reach.

Strength in Numbers

If you're more of a social learner and enjoy class-oriented situations, search the web for a local hackerspace or enthusiasts' group to see what people are making and to find Arduino-related groups. Members of Arduino groups can do such things as introduce the world of Arduino from an artist's perspective or work together to create a small Arduino-compatible board. These groups can be a lot of fun, introduce you to interesting people, and let you share your Arduino knowledge with others.

NOTE

You can also download the sketch files and find any updates at the book's website: <https://nostarch.com/arduino-workshop-2nd-edition/>.

Parts and Accessories

As with any other electronic device, the Arduino is available from many retailers offering a range of products and accessories. When you're shopping, be sure to purchase the original Arduino, or a quality derivative. Otherwise, you run the risk of receiving faulty or poorly performing goods. Why risk your project with an inferior board that could end up costing you more in the long run? For a list of authorized Arduino distributors, visit <http://arduino.cc/en/Main/Buy/>.

Here's a list of current suppliers (in alphabetical order) that I recommend for your purchases of Arduino-related parts and accessories:

Adafruit Industries (<http://www.adafruit.com/>)

Arduino Store USA (<https://store.arduino.cc/usa/>)

PMD Way (<https://pmdway.com/>)

SparkFun Electronics (<https://sparkfun.com/>)

You can download a list of the parts used in this book and find any updates at the book's website: <https://nostarch.com/arduino-workshop-2nd-edition/>. All the required parts are easily available from the various resellers listed above, as well as other retailers you may already be familiar with.

But don't go shopping yet. Take the time to read the first few chapters to get an idea of what you'll need so that you won't waste money buying unnecessary things.

Required Software

You should be able to program your Arduino with just about any computer. You'll begin by installing a piece of software called an *integrated development environment (IDE)*. To run this software, your computer should have an internet connection and one of the following operating systems installed:

macOS 10.14 64-bit, or higher

Windows 10 Home 32- or 64-bit, or higher

Linux 32- or 64-bit (Ubuntu or similar)

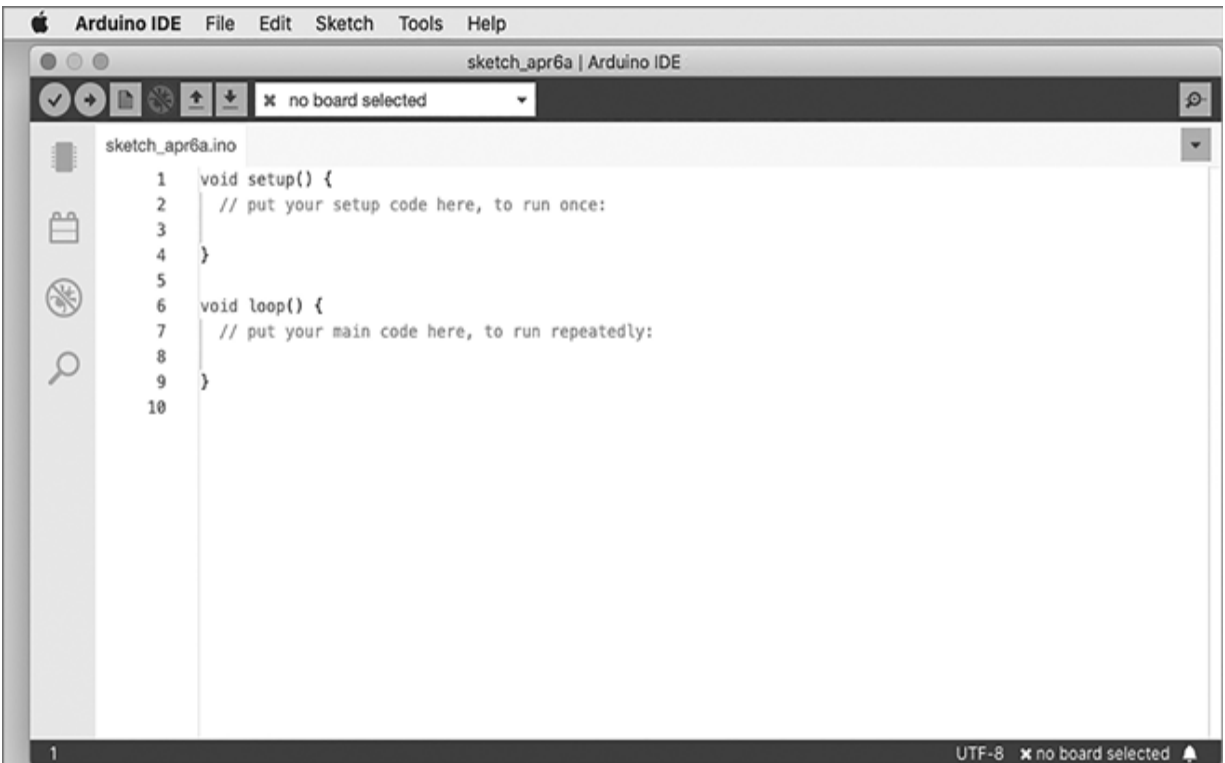
The IDE has been a work in progress since 2005 and is currently up to version 2.x (the exact number may change, but the instructions in this book should still work). Compared to version 1.x, version 2.x has some features that make writing and editing sketches easier, including interactive autocomplete, improved navigation through sketches, and more user-friendly board and library managers. Furthermore, a live debugger allows you to start and stop your Arduino sketch interactively when used with certain Arduino boards. However, if you're new to the world of Arduino, you don't need to worry about these features. Just remember that the Arduino team and community are always working on improvements.

Now is a good time to download and install the IDE, so jump to the heading that matches your operating system and follow the instructions. Make sure you have or buy the matching USB cable for your Arduino from the supplier as well. Even if you don't have your Arduino board yet, you can still download and explore the IDE.

macOS

In this section, you'll find instructions for downloading and configuring the Arduino IDE in macOS.

- Visit the software download page (<https://www.arduino.cc/en/software/>) and download the latest available version of the IDE for your operating system.
- Double-click the Arduino .dmg file in your *Downloads* folder. When the installation window pops up, drag the Arduino icon to the *Applications* folder.
- Open the IDE, as shown in [Figure 1-5](#).



[Figure 1-5](#): The IDE in macOS

- Now to configure the IDE for the Arduino Uno board. Click the top icon in the left sidebar of the IDE to open the Boards Manager. Find the option that includes the Arduino Uno and click **Install**.
- Expand the drop-down menu at the top of the IDE that reads No Board Selected and choose **Select Other Board & Port**. Then select the Arduino Uno from the list of boards.

You may be prompted to install Apple’s Command Line Developer tools.

Now your hardware and software are ready to work for you. Next, move on to “Using Arduino Safely” on page 8.

Windows 10

In this section, you'll find instructions for downloading and configuring the IDE in Windows.

- . Visit the software download page (<http://arduino.cc/en/software/>) and download the latest available version of the IDE for your operating system.
- . Your browser software may ask you to save or run the downloaded file. Click **Run** so the installation starts automatically once downloading has finished. Otherwise, launch the Arduino .exe file in your *Downloads* folder to install the IDE. When you're finished, run the IDE.
- . Now to configure the IDE for the Arduino Uno board. Click the top icon in the left sidebar of the IDE to open the Boards Manager. Find the option that includes the Arduino Uno and click **Install**.
- . Expand the drop-down menu at the top of the IDE that reads No Board Selected and choose **Select Other Board & Port**. Then select the Arduino Uno from the list of boards.

Now that your Arduino IDE is set up, you can move on to “Using Arduino Safely” on page 8.

Ubuntu Linux

If you are running Ubuntu Linux, here are instructions for downloading and setting up the Arduino IDE.

- . Visit the software download page (<http://arduino.cc/en/software/>) and download the latest available version of the IDE for your operating system.
- . If prompted, choose **Save File** and click **OK**.
- . Find the Arduino .zip file in the Archive Manager and extract it, saving it to the desktop.
- . Navigate to the extracted folder in a terminal and enter `./arduino-ide` to launch the IDE.
- . Now to configure the IDE. Connect your Arduino to your PC with the USB cable.

Select **Tools► Port** in the IDE and select the **/dev/ttyACMx** port, where *x* is a single digit (there should be only one port with a name like this).

Now your hardware and software are ready to work for you.

Using Arduino Safely

As with any hobby or craft, it's up to you to take care of yourself and those around you. As you'll see in this book, I discuss working with basic hand tools, battery-powered electrical devices, sharp knives, and cutters—and sometimes soldering irons. At no point in your projects should you work with the main household current. Leave that to a licensed electrician who is trained for such work. Remember that coming into contact with the wall power will kill you.

Looking Ahead

You're about to embark on a fun and interesting journey, and you'll be creating things you may never have thought possible. You'll find 65 Arduino projects in this book, ranging from the very simple to the relatively complex. All are designed to help you learn and make something useful. So let's go!

2

EXPLORING THE ARDUINO BOARD AND THE IDE

In this chapter, you'll explore the Arduino board as well as the IDE software that you'll use to create and upload Arduino *sketches* (Arduino's name for its programs) to the board itself. You'll learn the basic framework of a sketch and some basic functions that you can implement in a sketch, and you'll create and upload your first sketch.

The Arduino Board

What exactly is Arduino? According to the Arduino website (<http://www.arduino.cc/>), it is:

An open-source electronics platform based on easy-to-use hardware and software. It's intended for anyone making interactive projects.

In simple terms, the Arduino is a tiny computer system that can be programmed with your instructions to interact with various forms of input and output. The current Arduino board model, the Uno, is quite small compared to the adult human hand, as you can see in [Figure 2-1](#).



Figure 2-1: An Arduino Uno is quite small.

Although it might not look like much to the uninitiated, the Arduino system allows you to create devices that can interact with the world around you. With an almost unlimited range of input and output devices, such as sensors, indicators, displays, motors, and more, you can program the exact interactions you need to create a functional device. For example, artists have created installations with patterns of blinking lights that respond to the movements of passers-by, high school students have built autonomous robots that can detect an open flame and extinguish it, and geographers have designed systems that monitor temperature and humidity and transmit this data back to their offices via text message. In fact, a quick internet search will turn up an almost infinite number of examples of Arduino-based devices.

Let's explore our Arduino Uno *hardware* (in other words, the “physical part”) in more detail and see what we have. Don't worry too much about

understanding what you see here, because all these things will be discussed in greater detail in later chapters.

Starting at the left side of the board, you'll see two connectors, as shown in [Figure 2-2](#).

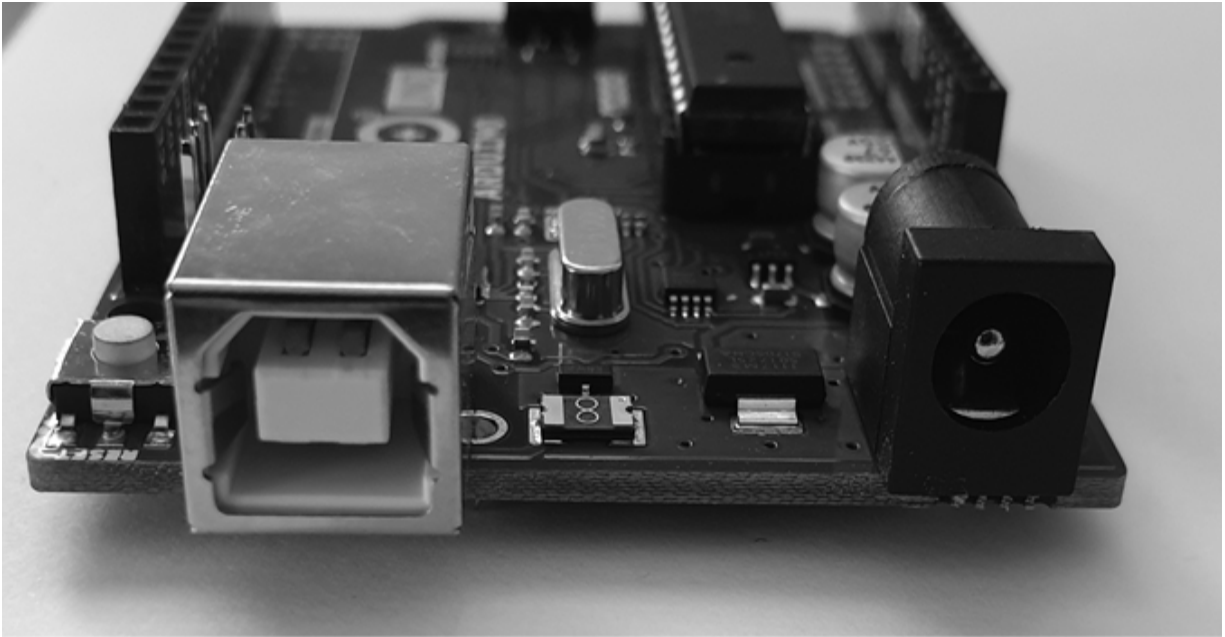


Figure 2-2: The USB and power connectors

On the left is the Universal Serial Bus (USB) connector. This connects the board to your computer, for three reasons: to supply power to the board, to upload your instructions to the Arduino, and to send data to and receive it from a computer. On the right is the power connector. Through this connector, you can power the Arduino with a standard wall power adapter (stepped down to 5 volts, of course).

At the lower middle is the heart of the board: the microcontroller, as shown in [Figure 2-3](#).

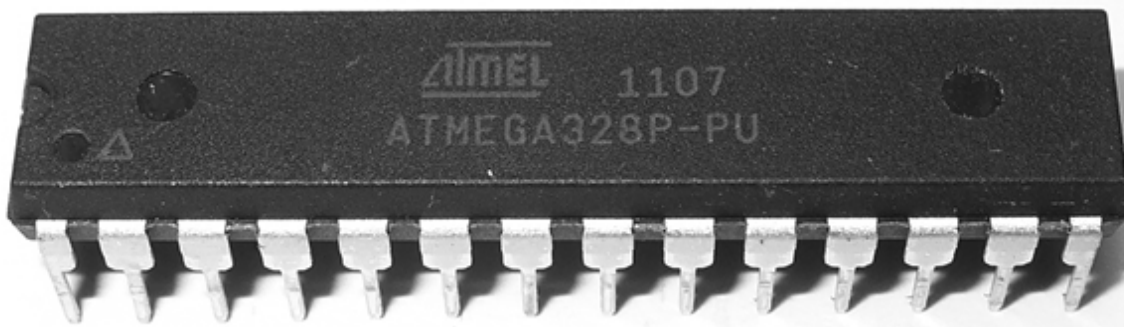


Figure 2-3: The microcontroller

The *microcontroller* is the “brains” of the Arduino. It is a tiny computer that contains a processor to execute instructions, includes various types of memory to hold data and instructions from our sketches, and provides various avenues for sending and receiving data. Just below the microcontroller are two groups of small sockets, as shown in [Figure 2-4](#).



Figure 2-4: The power and analog sockets

The group on the left offers power connections and the ability to use an external RESET button. The group on the right offers six analog inputs that are used to measure electrical signals that vary in voltage. Furthermore, pins A4 and A5 can also be used for sending data to and receiving it from other devices.

Along the top of the board are two more groups of sockets, as shown in [Figure 2-5](#).



Figure 2-5: The digital input/output pins

The sockets (or *pins*) numbered 0 to 13 are digital input/output (I/O) pins. They can either detect whether or not an electrical signal is present or generate a signal on command. Pins 0 and 1 are also known as the *serial port*, which is used to exchange data with other devices, such as a computer via the USB connector circuitry. The pins labeled with a tilde (~) can also generate a varying electrical signal (which looks like an ocean wave on an oscilloscope—thus the wavy tilde). This can be useful for such things as creating lighting effects or controlling electric motors.

The Arduino has some very useful devices called *light-emitting diodes* (*LEDs*); these very tiny devices light up when a current passes through them. The Arduino board has four LEDs: one on the far right labeled ON, which indicates when the board has power, and three in another group, as shown in [Figure 2-6](#).

The LEDs labeled *TX* and *RX* light up when data is being transmitted or received, respectively, between the Arduino and attached devices via the serial port and USB. The *L* LED is for your own use (it is connected to the digital I/O pin number 13). The little black square to the left of the LEDs is a tiny microcontroller that controls the USB interface that allows your Arduino to send data to and receive it from a computer, but you don't generally have to concern yourself with it.

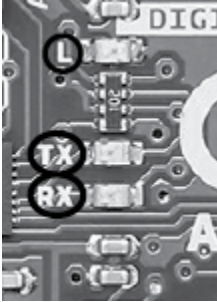


Figure 2-6: The onboard LEDs

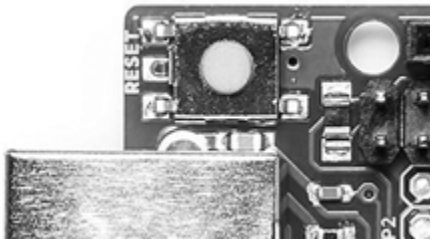


Figure 2-7: The RESET button

Finally, the RESET button is shown in [Figure 2-7](#).

As with a normal computer, sometimes things can go wrong with the Arduino. When all else fails, you might need to reset the system and restart your Arduino. The simple RESET button on the board is used to restart the system to resolve these problems.

One of the great advantages of the Arduino system is its ease of expandability—that is, it's easy to add more hardware functions. The two rows of sockets along each side of the Arduino allow the connection of a *shield*, another circuit board with pins that allow it to plug into the Arduino. For example, the shield shown in [Figure 2-8](#) contains an Ethernet interface that allows the Arduino to communicate over networks and the internet.

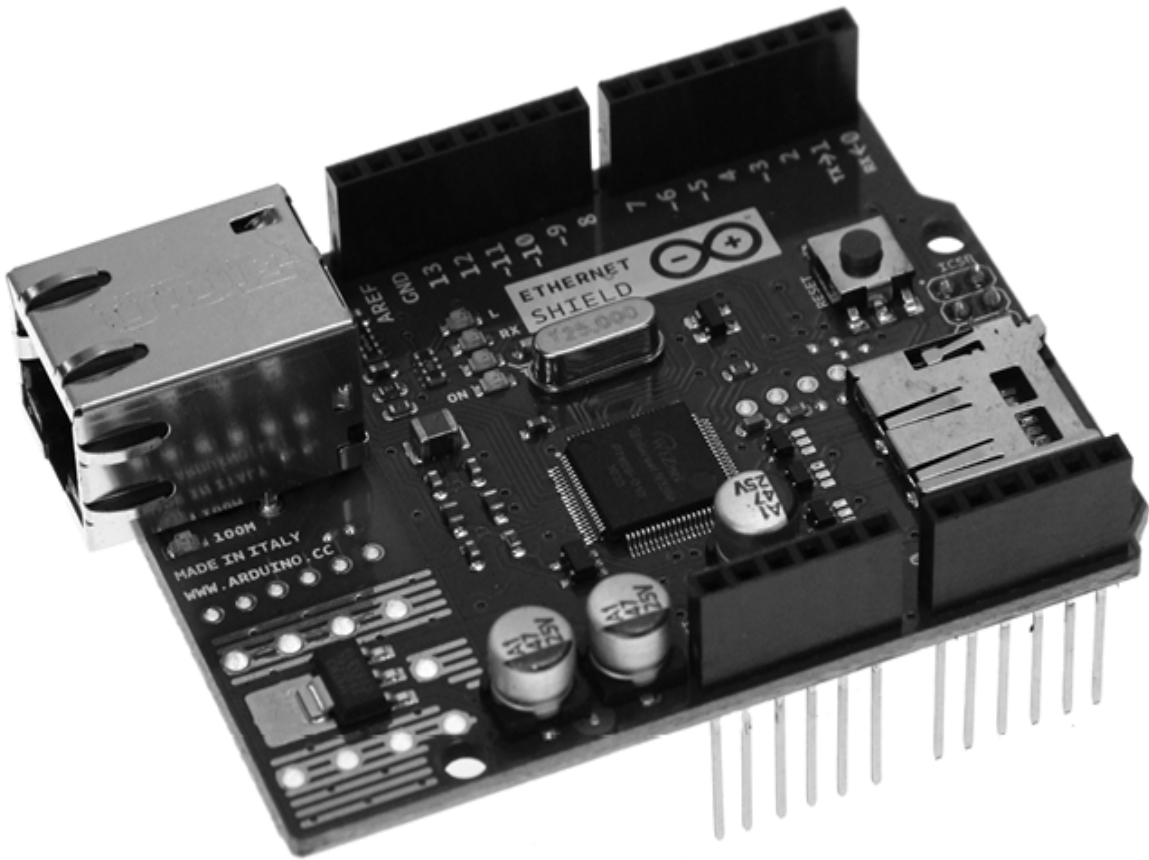


Figure 2-8: Arduino Ethernet interface shield

Notice that the Ethernet shield also has rows of sockets. These enable you to insert one or more shields on top. For example, [Figure 2-9](#) shows that another shield with a large numeric display, a temperature sensor, extra data storage space, and a large LED has been inserted.

If you use Arduino shields in your devices, you will need to remember which shield uses which individual inputs and outputs to ensure that “clashes” do not occur. You can also purchase completely blank shields that allow you to add your own circuitry. This will be explained further in Chapter 7.

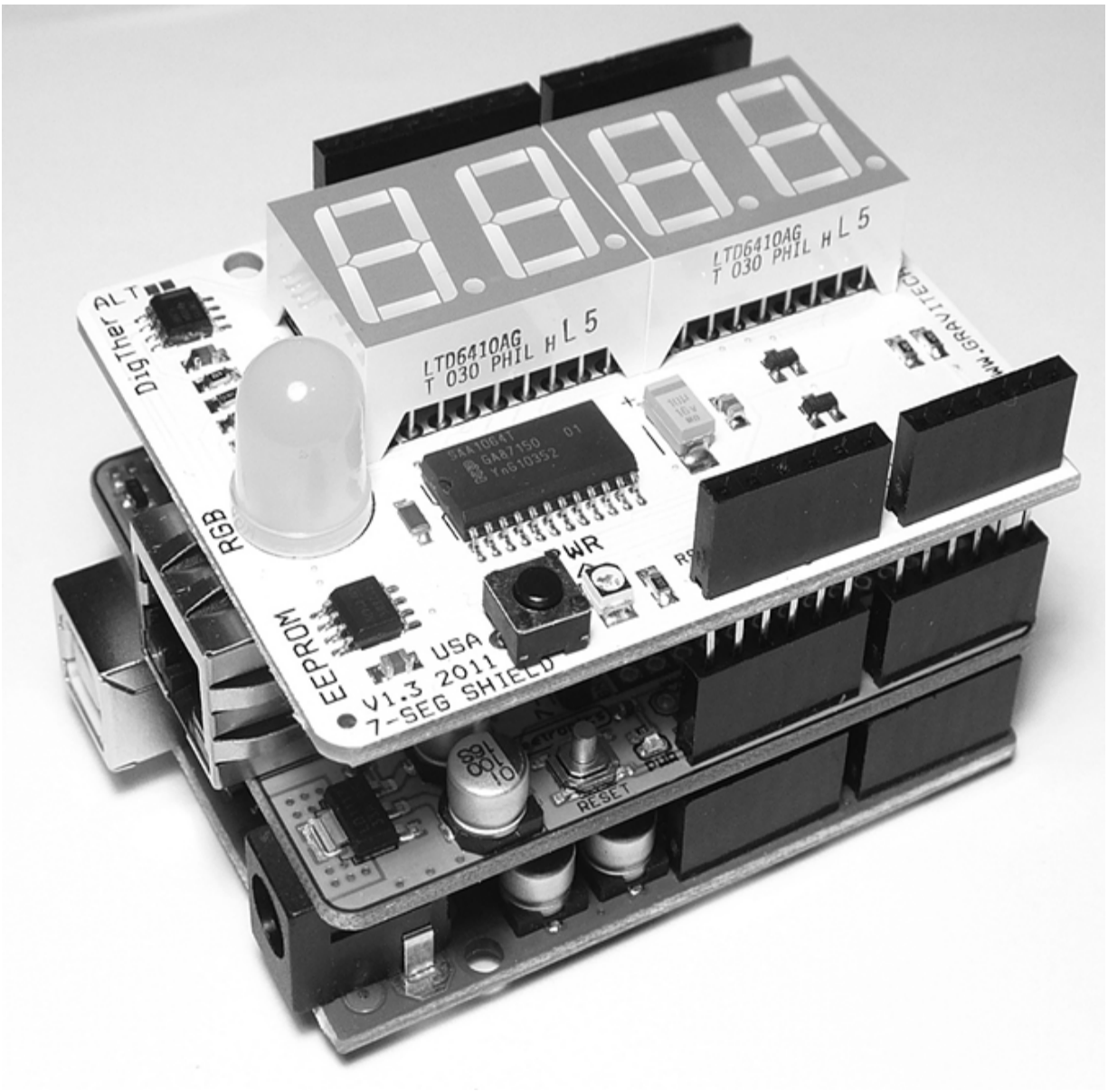


Figure 2-9: Numeric display and temperature shield

The companion to the Arduino hardware is the *software*, a collection of instructions that tell the hardware what to do and how to do it.

Back in Chapter 1, you installed the IDE software on your personal computer and configured it for your Arduino. Now you're going to look more closely at the IDE and then write a simple program—known as a *sketch*—for the Arduino.

Taking a Look Around the IDE

As shown in [Figure 2-10](#), the Arduino IDE resembles a simple word processor. The IDE is divided into three main areas: the command area, the text area, and the message window area.

The Command Area

The command area, shown at the top of [Figure 2-10](#), includes the title bar, menu items, and icons. The title bar displays the sketch's filename (such as *Blink*), as well as the version of the IDE (such as *Arduino 2.0.0-beta.4*). Below this is a series of menu items (File, Edit, Sketch, Tools, and Help) and icons, as described next.

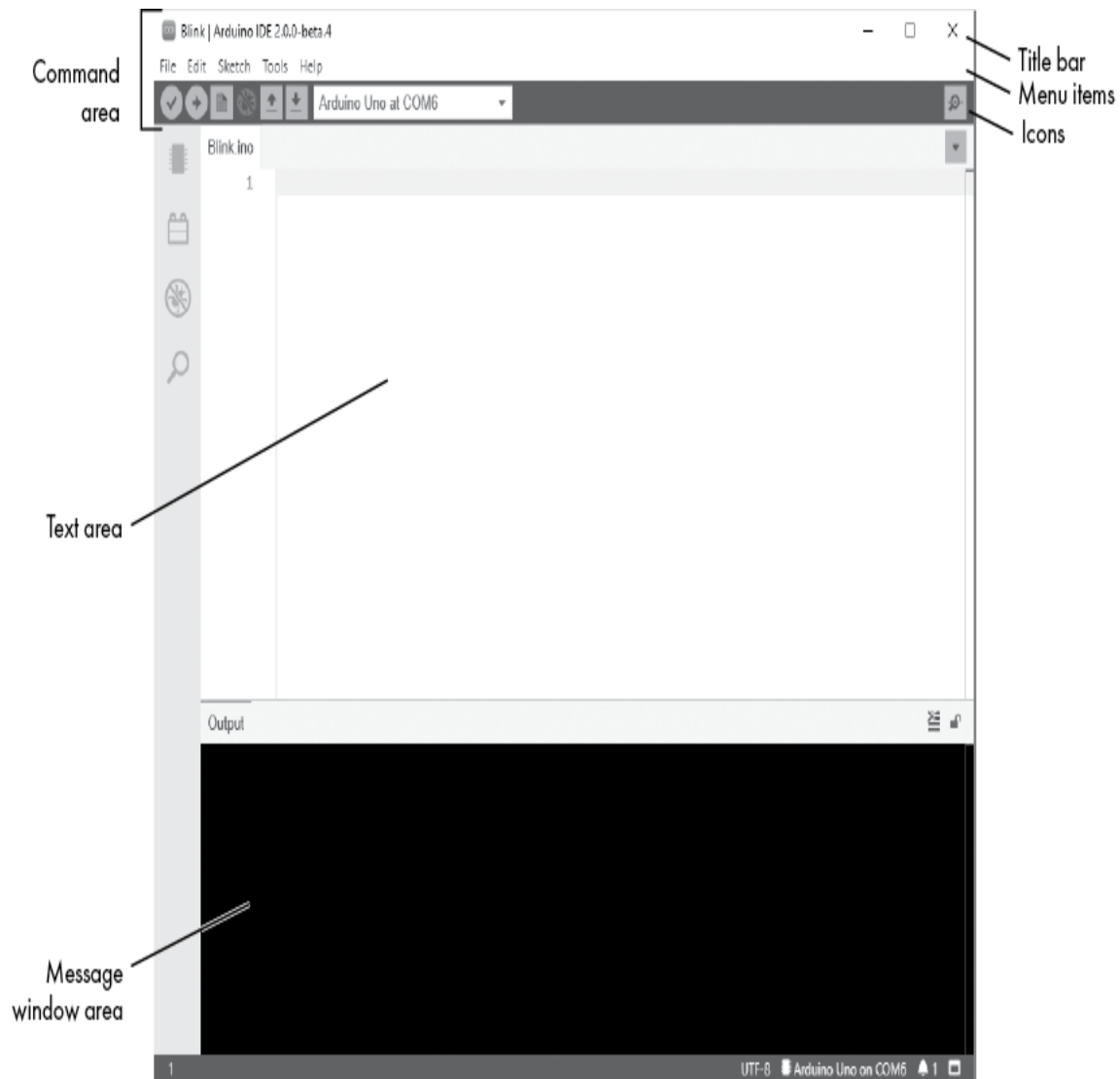


Figure 2-10: The Arduino IDE

Menu Items

As with any word processor or text editor, you can click one of the menu items to display its various options:

File Contains options to save, load, and print sketches; a thorough set of example sketches to open; and the Preferences submenu

Edit Contains the usual copy, paste, and search functions common to any word processor

Sketch Contains a function to verify your sketch before uploading it to a board, as well as some sketch folder and import options

Tools Contains a variety of functions as well as the commands to select the Arduino board type and USB port

Help Contains links to various topics of interest and the version of the IDE

The Icons

Below the menu toolbar are six icons. Mouse over each icon to display its name. The icons, from left to right, are as follows:

Verify Click this to check that the Arduino sketch is valid and doesn't contain any programming mistakes.

Upload Click this to verify and then upload your sketch to the Arduino board.

New Click this to open a new blank sketch in a new window.

Debug Used with more complex Arduino boards for real-time debugging.

Open Click this to open a saved sketch.

Save Click this to save the open sketch. If the sketch doesn't have a name, you will be prompted to create one.

Serial Monitor Click this to open a new window for use in sending and receiving data between your Arduino and the IDE.

The Text Area

The text area is shown in the middle of [Figure 2-10](#). This is where you'll create your sketches. The name of the current sketch is displayed in the tab at the upper left of the text area. (The default name is the current date.) You'll enter the contents of your sketch here as you would in any text editor.

The Output Window

The output window is shown at the bottom of [Figure 2-10](#). Messages from the IDE appear in the black area. The messages you see will vary and will include messages about verifying sketches, status updates, and so on.

At the bottom right of the output window, you should see the name of your Arduino board type as well as its connected USB port—*Arduino/Genuino Uno on COM4* in this case.

Creating Your First Sketch in the IDE

An Arduino sketch is a set of instructions that you create to accomplish a particular task; in other words, a sketch is a *program*. In this section, you'll create and upload a simple sketch that will cause the Arduino's LED (shown in [Figure 2-11](#)) to blink repeatedly, by turning it on and then off at one second intervals.



Figure 2-11: The LED on the Arduino board, next to the capital L

NOTE

Don't worry too much about the specific commands in the sketch we're creating here. The goal is to show you how easy it is to get the Arduino to do something so that you'll keep reading when you get to the harder stuff.

To begin, connect your Arduino to your computer with the USB cable. Then open the IDE and select your board (Arduino Uno) and USB port type from the drop-down menu, as shown in [Figure 2-12](#). This ensures that the Arduino board is properly connected.

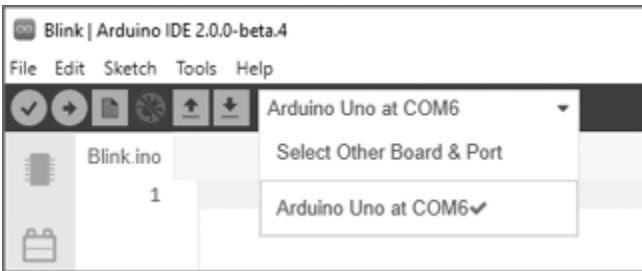


Figure 2-12: Selecting the Arduino Uno board

Comments

First, enter a comment as a reminder of what your sketch will be used for. A *comment* in a sketch is a note written for the user's benefit. Comments can be notes to yourself or others, and they can include instructions or any other details. When creating sketches for your Arduino, it's a good idea to add comments about your intentions for the code; these comments can prove useful later when you're revisiting a sketch.

To add a comment on a single line, enter two forward slashes and then the comment, like this:

```
// Blink LED sketch by Mary Smith, created 07/01/2021
```

The two forward slashes tell the IDE to ignore that line of text when *verifying* a sketch, or checking that everything is written properly with no errors.

To enter a comment that spans two or more lines, enter the characters `/*` on a line before the comment and end the comment with the characters `*/` on the following line, like this:

```
/*  
Arduino Blink LED Sketch  
by Mary Smith, created 07/01/2021  
*/
```

The `/*` and `*/` tell the IDE to ignore the text that they bracket.

Enter a comment describing your Arduino sketch using one of these methods. Then save your sketch by choosing **File►Save As**. Enter a short name for your sketch (such as *blinky*) and click **OK**.

The default filename extension for Arduino sketches is *.ino*, and the IDE should add this automatically. The name for your sketch should be, in this case, *blinky.ino*, and you should be able to see it in your Sketchbook.

The setup() Function

The next stage in creating any sketch is to fill in the `void setup()` function. This function contains a set of instructions for the Arduino to execute once only, each time it is reset or turned on. To create the `setup()` function, add the following lines to your sketch, after the comments:

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

Controlling the Hardware

Our program will blink the user LED on the Arduino. The user LED is connected to the Arduino's digital pin 13. A digital pin can either detect an electrical signal or generate one on command. In this project, we'll generate an electrical signal that will light the LED.

Enter the following into your sketch between the braces (`{` and `}`):

```
pinMode(13, OUTPUT); // set digital pin 13 to output
```

The number 13 in the listing represents the digital pin you're addressing. You're setting this pin to `OUTPUT`, which means it will generate an electrical signal. If you wanted it to detect an incoming electrical signal, then you would set the pin's mode to `INPUT` instead. Notice that the `pinMode()` line ends with a semicolon (`;`). Every instruction line in your Arduino sketches will end with a semicolon.

Save your sketch at this point to make sure that you don't lose any of your work.

The loop() Function

Remember that our goal is to make the LED blink repeatedly. To do this, we'll create a `loop()` function to tell the Arduino to execute an instruction

over and over until the power is shut off or someone presses the RESET button.

Enter the code shown in boldface after the `void setup()` section in the following listing to create an empty `loop()` function. Be sure to end this new section with another brace `}`, and then save your sketch again:

```
/*
Arduino Blink LED Sketch
by Mary Smith, created 07/01/21
*/

void setup()
{
  pinMode(13, OUTPUT); // set digital pin 13 to output
}
void loop()
{ // place your main loop code here:
}
```

WARNING

The Arduino IDE does not automatically save sketches, so save your work frequently!

Next, enter the actual functions into `void loop()` for the Arduino to execute.

Enter the following between the `loop()` function's braces. Then click **Verify** to make sure that you've entered everything correctly:

```
digitalWrite(13, HIGH); // turn on digital pin 13
delay(1000);           // pause for one second
digitalWrite(13, LOW);  // turn off digital pin 13
delay(1000);           // pause for one second
```

Let's take this apart. The `digitalWrite()` function controls the voltage that is output from a digital pin: in this case, pin 13, connected to the LED. By setting the second parameter of this function to `HIGH`, we tell the pin to

output a “high” digital voltage; current will flow from the pin, and the LED will turn on.

The `delay()` function causes the sketch to do nothing for a period of time—in this case, with the LED turned on, `delay(1000)` causes it to remain lit for 1,000 milliseconds, or 1 second.

Next, we turn off the voltage to the LED with `digitalWrite(13, LOW);`. The current flowing through the LED stops, and the light turns off. Finally, we pause again for 1 second while the LED is off, with `delay(1000);`.

The completed sketch should look like this:

```
/*
Arduino Blink LED Sketch
by Mary Smith, created 07/01/21
*/

void setup()
{
  pinMode(13, OUTPUT); // set digital pin 13 to output
}

void loop()
{
  digitalWrite(13, HIGH); // turn on digital pin 13
  delay(1000);             // pause for one second
  digitalWrite(13, LOW);  // turn off digital pin 13
  delay(1000);             // pause for one second
}
```

Before you do anything further, save your sketch!

Verifying Your Sketch

When you verify your sketch, you ensure that it has been written correctly in a way that the Arduino can understand. To verify your complete sketch, click **Verify** in the IDE and wait a moment. Once the sketch has been verified, a note should appear in the output window, as shown in [Figure 2-13](#).

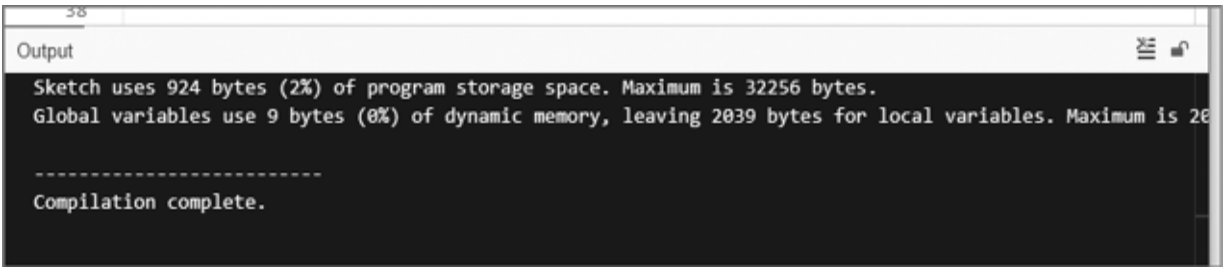


Figure 2-13: The sketch has been verified.

This “Done compiling” message tells you that the sketch is okay to upload to your Arduino. It also shows how much memory it will use (924 bytes in this case) of the total available on the Arduino (32,256 bytes).

But what if your sketch isn’t okay? Say, for example, you forgot to add a semicolon at the end of the second `delay(1000)` function. If something is broken in your sketch, then when you click **Verify**, the message window should display a verification error message similar to the one shown in [Figure 2-14](#).

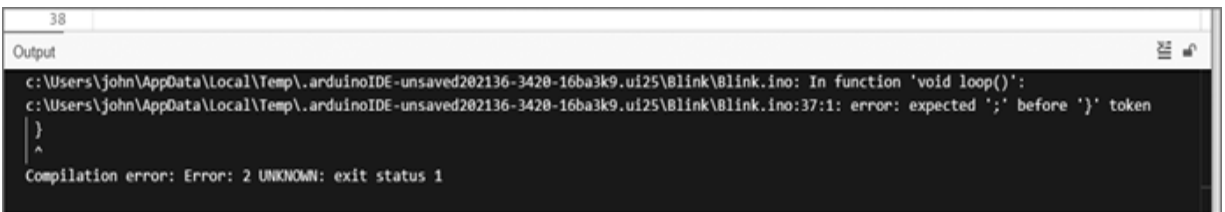


Figure 2-14: The message window with a verification error

The IDE displays the error itself (the missing semicolon, described by error: expected ';' before '}' token). It should also highlight the location of the error, or a spot just after it. This helps you easily locate and rectify the mistake.

Uploading and Running Your Sketch

Once you’re satisfied that your sketch has been entered correctly, save it. Then make sure that your Arduino board is connected to your computer and click **Upload** in the IDE. The IDE verifies your sketch again and then uploads it to your Arduino board. During this process, the TX/RX LEDs on your board (shown in [Figure 2-6](#)) should blink, indicating that information is traveling between the Arduino and your computer.

Now for the moment of truth: your Arduino should start running the sketch. If you've done everything correctly, the LED should blink on and off once every second!

Congratulations. You now know the basics of how to enter, verify, and upload an Arduino sketch.

Modifying Your Sketch

After running your sketch, you may want to change how it operates by, for example, adjusting the on or off delay time for the LED. Because the IDE is a lot like a word processor, you can open your saved sketch, adjust the values, and then save your sketch again and upload it to the Arduino. For example, to increase the rate of blinking, change both `delay` functions to make the LEDs blink for one-quarter of a second by adjusting the delay to 250, like this:

```
delay(250); // pause for one-quarter of one second
```

Then upload the sketch again. The LED should now blink faster, for one-quarter of a second each time.

Looking Ahead

Armed with your newfound knowledge of how to enter, edit, save, and upload Arduino sketches, you're ready for the next chapter, where you'll learn how to use more functions, implement good project design, construct basic electronic circuits, and do much more.

3

FIRST STEPS

In this chapter you will

Learn the concepts of good project design

Learn the basic properties of electricity

Be introduced to the resistor, light-emitting diode (LED), transistor, rectifier diode, and relay

Use a solderless breadboard to construct circuits

Learn how integer variables, for loops, and digital outputs can be used to create various LED effects

Now you'll begin to bring your Arduino to life. As you will see, there is more to working with Arduino than just the board itself. You'll learn how to plan projects in order to make your ideas a reality, then move on to a quick primer on electricity. Electricity is the driving force behind everything we do in this book, and it's important to have a solid understanding of the basics in order to create your own projects. You'll also take a look at the components that bring real projects to life. Finally, you'll examine some new functions that are the building blocks for your Arduino sketches.

Planning Your Projects

When starting your first few projects, you might be tempted to write your sketch immediately after you've come up with a new idea. But before you start writing, a few basic preparatory steps are in order. After all, your Arduino board isn't a mind reader; it needs precise instructions, and even if

these instructions can be executed by the Arduino, if you overlook so much as a minor detail, the results may not be what you expected.

Whether you are creating a project that simply blinks a light or one that controls an automated model railway signal, you'll be more successful if you have a detailed plan. When designing your Arduino projects, follow these basic steps:

Define your objective. Determine what you want to achieve.

Write your algorithm. An *algorithm* is a set of instructions that describes how to accomplish your goal. Your algorithm will list the steps necessary for you to achieve your project's objective.

Select your hardware. Determine how your hardware will connect to the Arduino.

Write your sketch. Create your initial program that tells the Arduino what to do.

Wire it up. Connect your hardware to the Arduino board.

Test and debug. Does it work? During this stage, you identify errors and find their causes, whether in the sketch, hardware, or algorithm.

The more time you spend planning your project, the easier a time you'll have during the testing and debugging stage.

NOTE

Even well-planned projects sometimes fall prey to feature creep. Feature creep occurs when people think up new functionality that they want to add to a project and then try to force new elements into an existing design. When you need to change a design, don't try to "slot in" changes or modify it with 11th-hour additions. Instead, start fresh by redefining your objective.

About Electricity

Let's spend a bit of time discussing electricity, since you'll soon be building electronic circuits with your Arduino projects. In simple terms, *electricity* is a form of energy that we can harness and convert into heat, light, movement, and power. Electricity has three main properties that will be important to us as we build projects: current, voltage, and power.

Current

The flow of electrical energy is called the *current*. Electrical current flows through a *circuit* (a path for the current) from the positive side of a power source, such as a battery, to the negative side of the power source. This is known as *direct current (DC)*. (For the purposes of this book, we will not deal with *alternating current*, or AC.) In some circuits, the negative side is called *ground (GND)*. Current is measured in *amperes* or "amps" (A); 1 amp is 6.2415×10^{18} electrons flowing past a single point in 1 second. Smaller amounts of current are measured in *milliamps (mA)*, where 1,000 milliamps equal 1 amp.

Voltage

Voltage is a measure of the difference in potential energy between a circuit's positive and negative ends. This is measured in *volts (V)*. If you think of electrons flowing the way water flows, then voltage would be equivalent to pressure: the greater the voltage, the faster the current moves through a circuit.

Power

Power is a measurement of the rate at which an electrical device converts energy from one form to another. Power is measured in *watts (W)*. For example, a 100 W light bulb is much brighter than a 60 W bulb because the higher-wattage bulb converts more electrical energy into light.

A simple mathematical relationship exists among voltage, current, and power:

$$\text{Power (W)} = \text{Voltage (V)} \times \text{Current (A)}$$

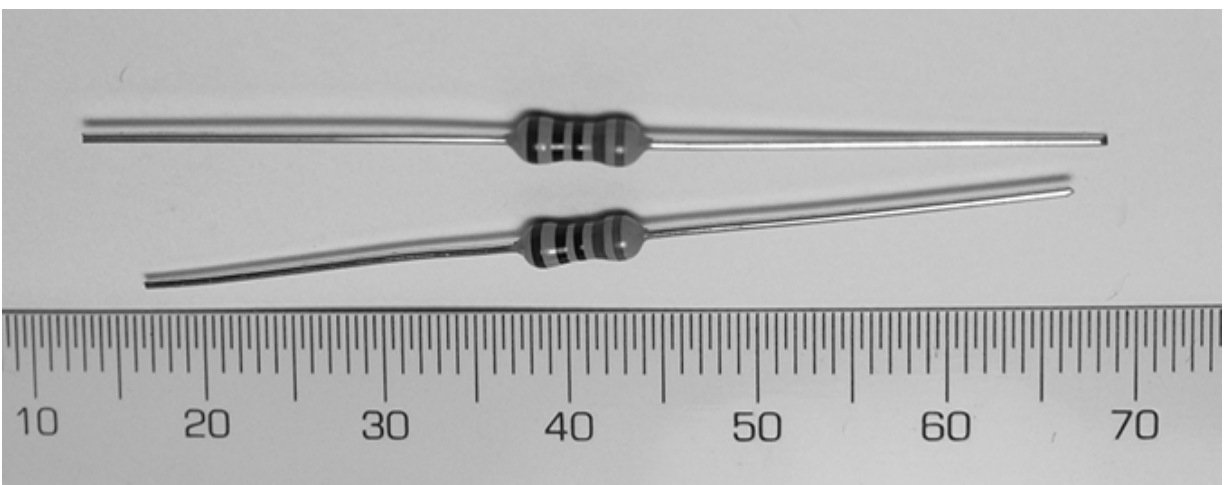
Electronic Components

Now that you know a little bit about the basics of electricity, let's look at how it interacts with various electronic components and devices. Electronic *components* are the various parts that control electric current in a circuit. Just as the various parts of a car's engine work together to store fuel, filter fuel, pump fuel, and inject fuel to allow us to drive, electronic components work together to control and harness the flow of electricity to help us create useful devices.

Throughout this book, I'll explain specialized components as we use them. The following sections describe some of the fundamental components.

The Resistor

Various components, such as the Arduino's LED, require only a small amount of current to function—usually around 10 mA. When the LED receives more current than it needs, it converts the excess to heat—and too much heat can kill an LED. To reduce the flow of current to components such as LEDs, we can add a *resistor* between the voltage source and the component. Current flows freely along normal copper wire, but when it encounters a resistor, its movement is slowed. Some current is converted into a small amount of heat, which is proportional to the value of the resistor. [Figure 3-1](#) shows some commonly used resistors.



[Figure 3-1](#): Typical resistors

Resistance

The level of resistance can be either fixed or variable. Resistance is measured in *ohms* (Ω) and can range from zero to thousands of ohms (*kilohms*, or $k\Omega$) to millions of ohms (*megohms*, or $M\Omega$).

Reading Resistance Values

Although you can test resistance with a multimeter, you can also read resistance directly from a physical resistor. The resistors we will use will be physically very small, so their resistance value usually cannot be printed on them. One common way to show a component's resistance is with a series of color-coded bands, read from left to right, as follows:

First band Represents the first digit of the resistance

Second band Represents the second digit of the resistance

Third band Represents the multiplier (for four-band resistors) or the third digit (for five-band resistors)

Fourth band Represents the multiplier (for five-band resistors) or the *tolerance*, or accuracy of the component's resistance (for four-band resistors)

Fifth band Shows the tolerance for five-band resistors

[Table 3-1](#) lists the colors of resistors and their corresponding values.

Because it is difficult to manufacture resistors with exact values, you select a margin of error as a percentage when buying a resistor. For five-band resistors, a brown band in the fifth position indicates tolerance of 1 percent, gold indicates 5 percent, and silver indicates 10 percent.

[Figure 3-2](#) shows a resistor diagram. The yellow, violet, and orange resistance bands are read as 4, 7, and 3, respectively, as listed in [Table 3-1](#). The third band represents the multiplier; in this example, the 47 is multiplied by 10 to the power of 3 to arrive at the value of 47,000 Ω , more commonly read as 47 $k\Omega$. The brown band indicates a very precise resistor, which should be accurate to within 1 percent.

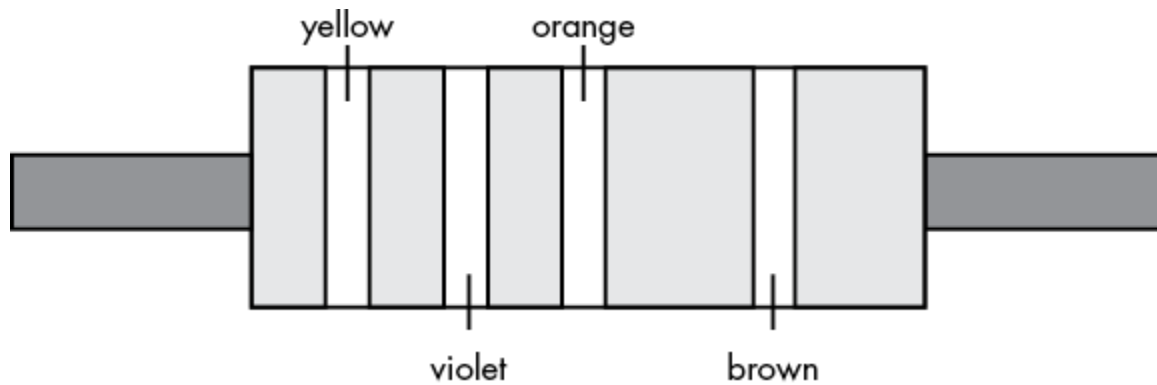


Figure 3-2: Example resistor diagram

Table 3-1: *Values of Bands Printed on a Resistor, in Ohms*

Color	Ohms
Black	0
Brown	1
Red	2
Orange	3
Yellow	4
Green	5
Blue	6
Violet	7
Gray	8
White	9

Chip Resistors

Surface-mount chip resistors display a printed number and letter code, as shown in [Figure 3-3](#), instead of colored stripes. The first two digits represent a single number, and the third digit represents the number of zeros to follow that number. For example, the resistor in [Figure 3-3](#) has a value of 10,000 Ω , or 10 k Ω .

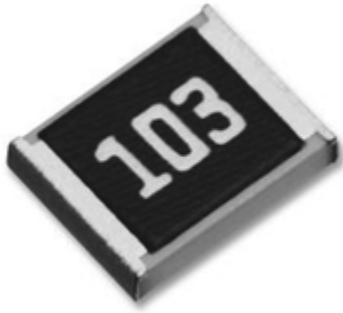


Figure 3-3: A surface-mount resistor

NOTE

If you see a number and letter code on a small chip resistor (such as 01C), google EIA-96 code calculator for lookup tables for that more involved code system.

MULTIMETERS

A *multimeter* is an incredibly useful and relatively inexpensive piece of test equipment that can measure voltage, resistance, current, and more. [Figure 3-4](#) shows a multimeter measuring a resistor.



[Figure 3-4](#): A multimeter measuring a 560 Ω , 1 percent tolerance resistor

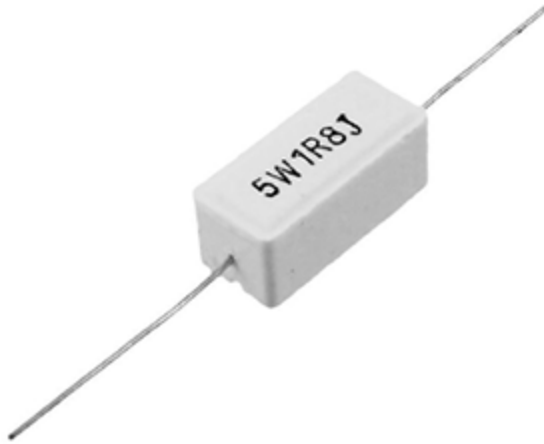
If you have difficulty distinguishing the different codes on a color-coded resistor, a multimeter is essential. As with other good tools, purchase your multimeter from a reputable retailer instead of fishing about on the internet for the cheapest one you can find.

Power Rating

The resistor's *power rating* is a measurement of the power, in watts, that it will tolerate before overheating or failing. The resistors shown in [Figure 3-1](#) are 1/4W resistors, which are the most commonly used resistors with the Arduino system. For the purposes of the projects in this book, you only need 1/4W resistors.

When you're selecting a resistor, consider the relationship between power, current, and voltage. The greater the current and/or voltage in your design, the greater the resistor's power rating should be.

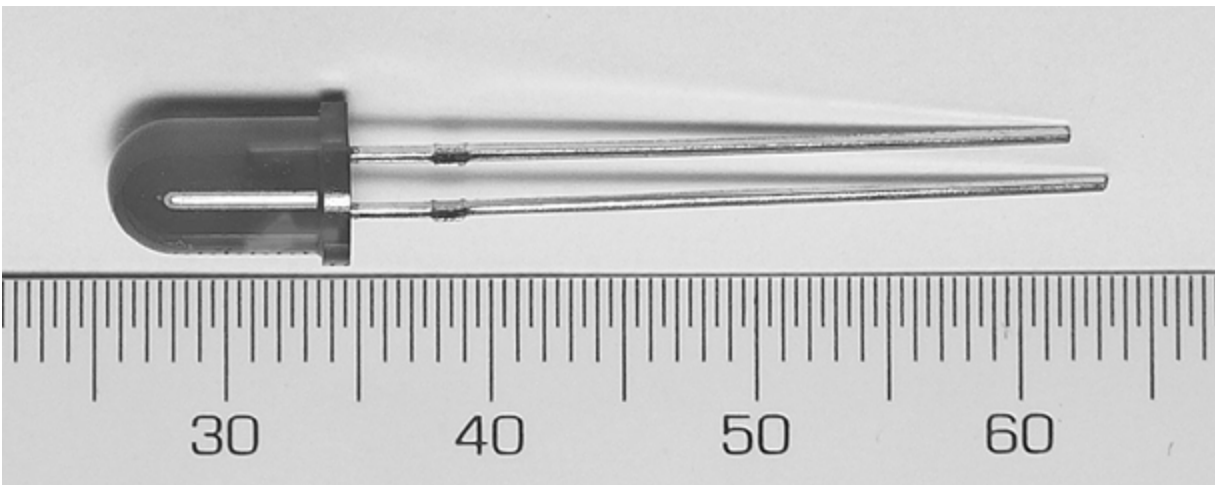
Usually, the greater a resistor's power rating, the greater its physical size. For example, the resistor shown in [Figure 3-5](#) is a 5W resistor, whose body measures 22 mm long by 10 mm wide.



[Figure 3-5](#): A 5W resistor

The Light-Emitting Diode

The LED is a very common, infinitely useful component that converts electrical current into light. LEDs come in various shapes, sizes, and colors. [Figure 3-6](#) shows a common LED.

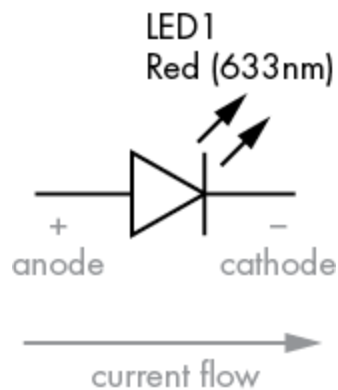


[Figure 3-6](#): A red LED, 5 mm in diameter

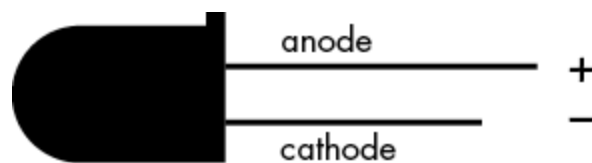
Connecting LEDs in a circuit takes some care, because they are *polarized*; this means that current can enter and leave the LED in one direction only.

The current enters via the *anode* (positive) side and leaves via the *cathode* (negative) side, as shown in [Figure 3-7](#). Any attempt to make too much current flow through an LED in the opposite direction will break the component.

Thankfully, LEDs are designed so that you can tell which end is which. The leg on the anode side is longer (you can think of the “plus” side as having length “added” to it), and the rim at the base of the LED is flat on the cathode side, as shown in [Figure 3-8](#).



[Figure 3-7](#): Current flow through an LED



[Figure 3-8](#): LED design indicates the anode (longer leg) and cathode (flat rim) sides.

When adding LEDs to a project, you need to consider the operating voltage and current. For example, common red LEDs require around 1.7 V and 5 to 20 mA of current. This presents a slight problem for us, because the Arduino outputs a set 5 V and a much higher current. Luckily, we can use a current-limiting resistor to reduce the current flow into an LED. But which value resistor do we use? That’s where Ohm’s law comes in.

To calculate the required current-limiting resistor for an LED, use this formula:

$$R = (V_s - V_f) \div I$$

where V_s is the supply voltage (Arduino outputs 5 V), V_f is the LED forward voltage drop (say, 1.7 V), and I is the current required for the LED (10 mA). (The value of I must be in amps, so 10 mA converts to 0.01 A.)

Now let's apply this formula to our LEDs, using values of 5 V for V_s , 1.7 V for V_f , and 0.01 A for I . Substituting these values into the formula gives a value for R of 330 Ω . However, the LEDs will happily light up when fed current less than 10 mA. It's good practice to use lower currents when possible to protect sensitive electronics, so we'll use 560 Ω , 1/4W resistors with our LEDs, which allow around 6 mA of current to flow.

NOTE

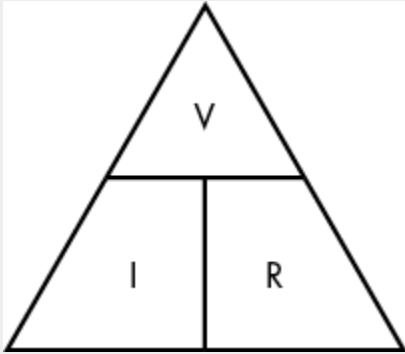
When in doubt, always choose a slightly higher value resistor, because it's better to have a dim LED than a dead one!

THE OHM'S LAW TRIANGLE

Ohm's law states that the relationship between current, resistance, and voltage is as follows:

$$\text{Voltage (V)} = \text{Current (I)} \times \text{Resistance (R)}$$

If you know two of the quantities, you can calculate the third. A popular way to remember Ohm's law is with a triangle, as shown in [Figure 3-9](#).



[Figure 3-9](#): The Ohm's law triangle

The Ohm's law triangle diagram is a convenient tool for calculating voltage, current, or resistance when two of the three values are known. For example, if you need to calculate resistance, put your finger over *R*, leaving voltage divided by current. To calculate voltage, cover *V*, leaving current times resistance.

The Solderless Breadboard

Our ever-changing circuits will need a base—something to hold them together and build upon. A great tool for this purpose is a *solderless breadboard*. The breadboard is a plastic base with rows of electrically connected sockets (just don't cut bread on them). They come in many sizes, shapes, and colors, as shown in [Figure 3-10](#).

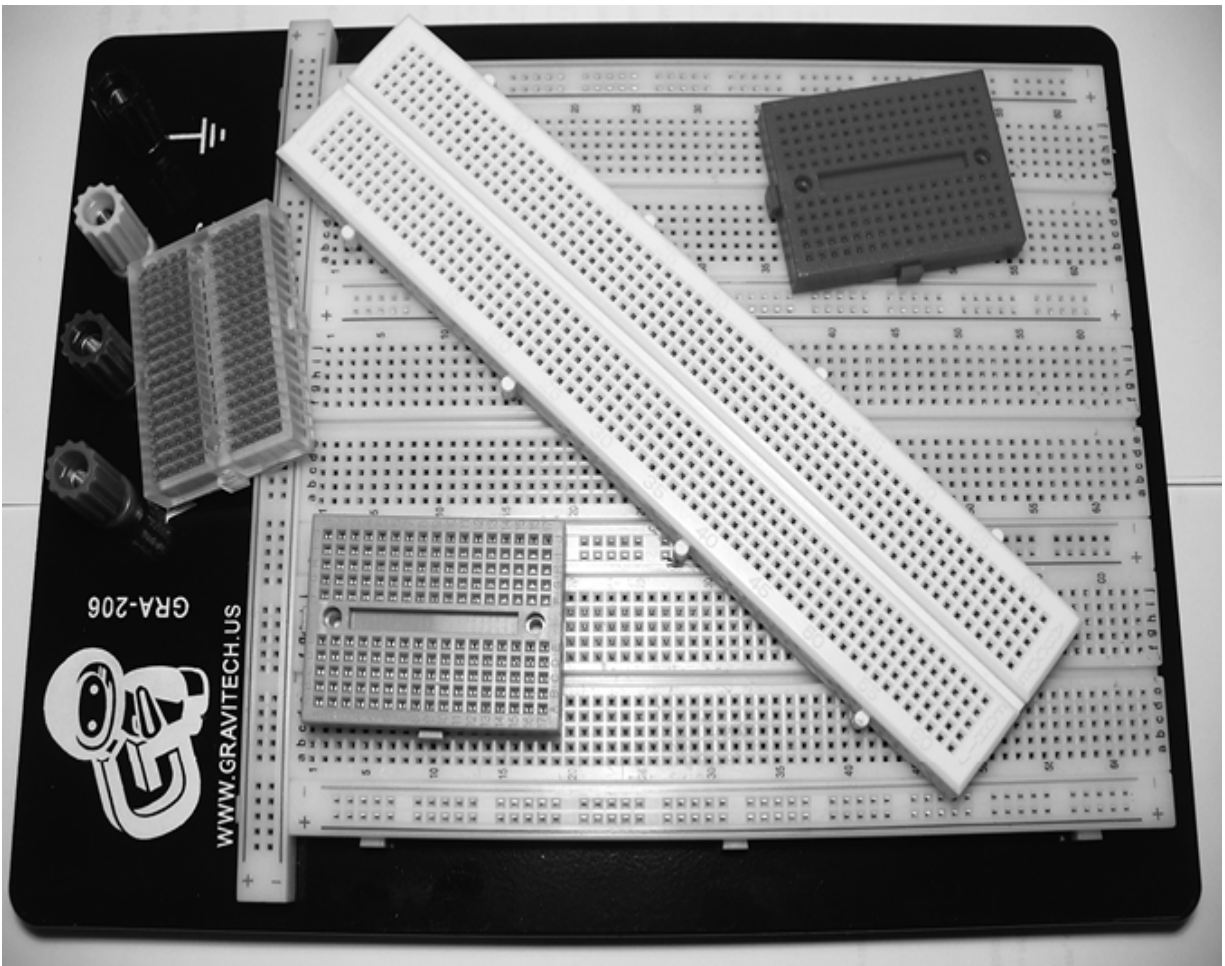
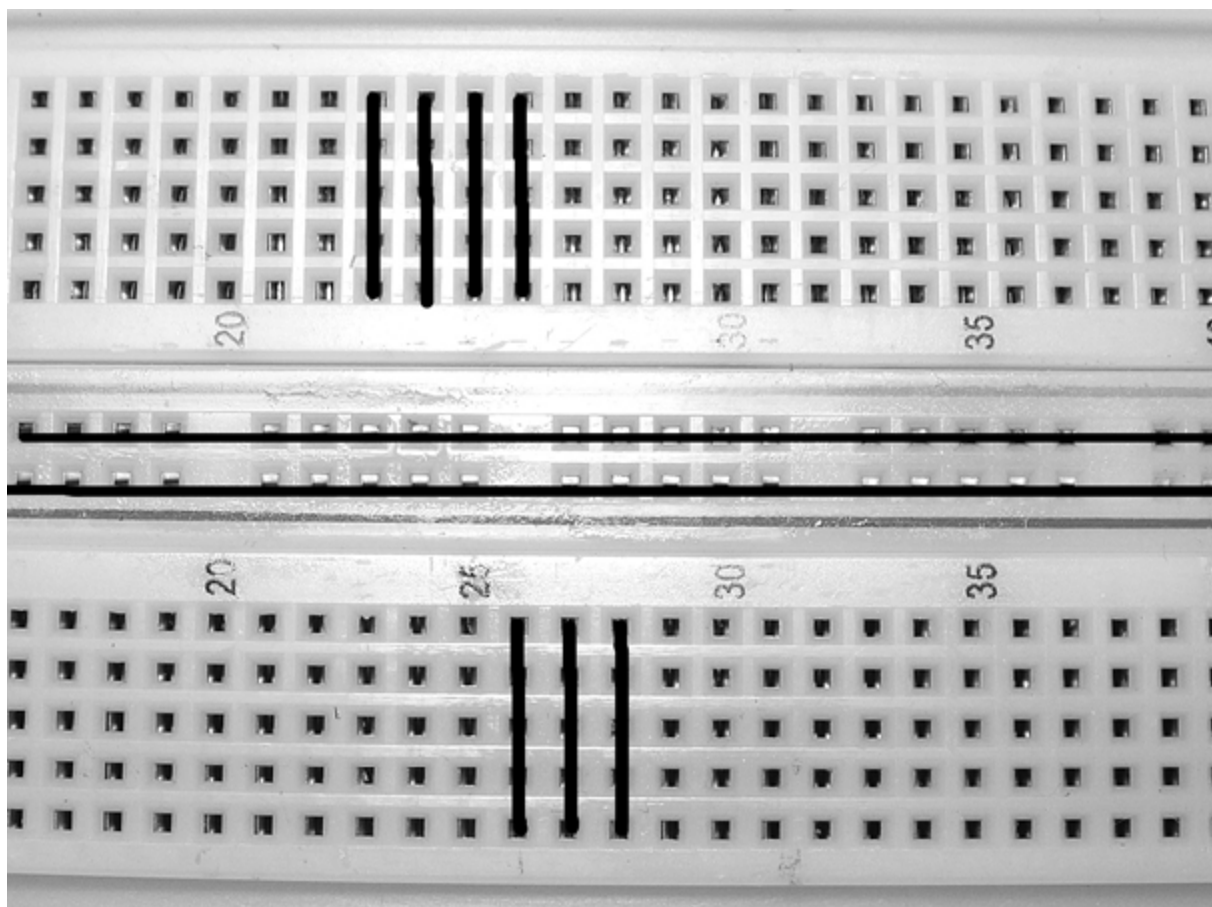


Figure 3-10: Breadboards in various shapes and sizes

The key to using a breadboard is knowing how the sockets are connected—whether in short columns or in long rows along the edge or in the center. The connections vary by board. For example, in the breadboard shown at the top of [Figure 3-11](#), columns of five holes are connected vertically but isolated horizontally. If you place two wires in one vertical row, then they will be electrically connected. By the same token, the long rows in the center between the horizontal lines are connected horizontally. You'll often need to connect a circuit to the supply voltage and ground, and these long horizontal lines of holes are ideal for that purpose.

When you're building more complex circuits, a breadboard will get crowded, and you won't always be able to place components exactly where you want. It's easy to solve this problem using short connecting wires,

however. Retailers that sell breadboards usually also sell small boxes of wires of various lengths, such as the assortment shown in [Figure 3-12](#).



[Figure 3-11](#): Breadboard internal connections



Figure 3-12: Assorted breadboard wires

Project #1: Creating a Blinking LED Wave

Let's put some LEDs and resistors to work. In this project, we'll use five LEDs to emulate the front of the famous vehicle KITT from the television show *Knight Rider*, creating a kind of wavelike light pattern.

The Algorithm

Here's our algorithm for this project:

- . Turn on LED 1.
- . Wait half a second.
- . Turn off LED 1.
- . Turn on LED 2.
- . Wait half a second.
- . Turn off LED 2.
- . Continue until LED 5 is turned on, at which point the process reverses from LEDs 5 to 1.
- . Repeat indefinitely.

The Hardware

Here's what you'll need to create this project:

Five LEDs

Five 560 Ω resistors

One breadboard

Various connecting wires

Arduino and USB cable

We will connect the LEDs to digital pins 2 through 6 via the 560 Ω current-limiting resistors.

The Schematic

Now let's build the circuit. Circuit layout can be described in several ways. For the first few projects in this book, we'll use physical layout diagrams similar to the one shown in [Figure 3-13](#).

By comparing the wiring diagram to the functions in the sketch, you can begin to make sense of the circuit. For example, when we use `digitalWrite(2, HIGH)`, a high voltage of 5 V flows from digital pin 2, through the current-limiting resistor, through the LED via the anode and

then the cathode, and finally back to the Arduino's GND socket to complete the circuit. Then, when we use `digitalWrite(2, LOW)`, the current stops and the LED turns off.

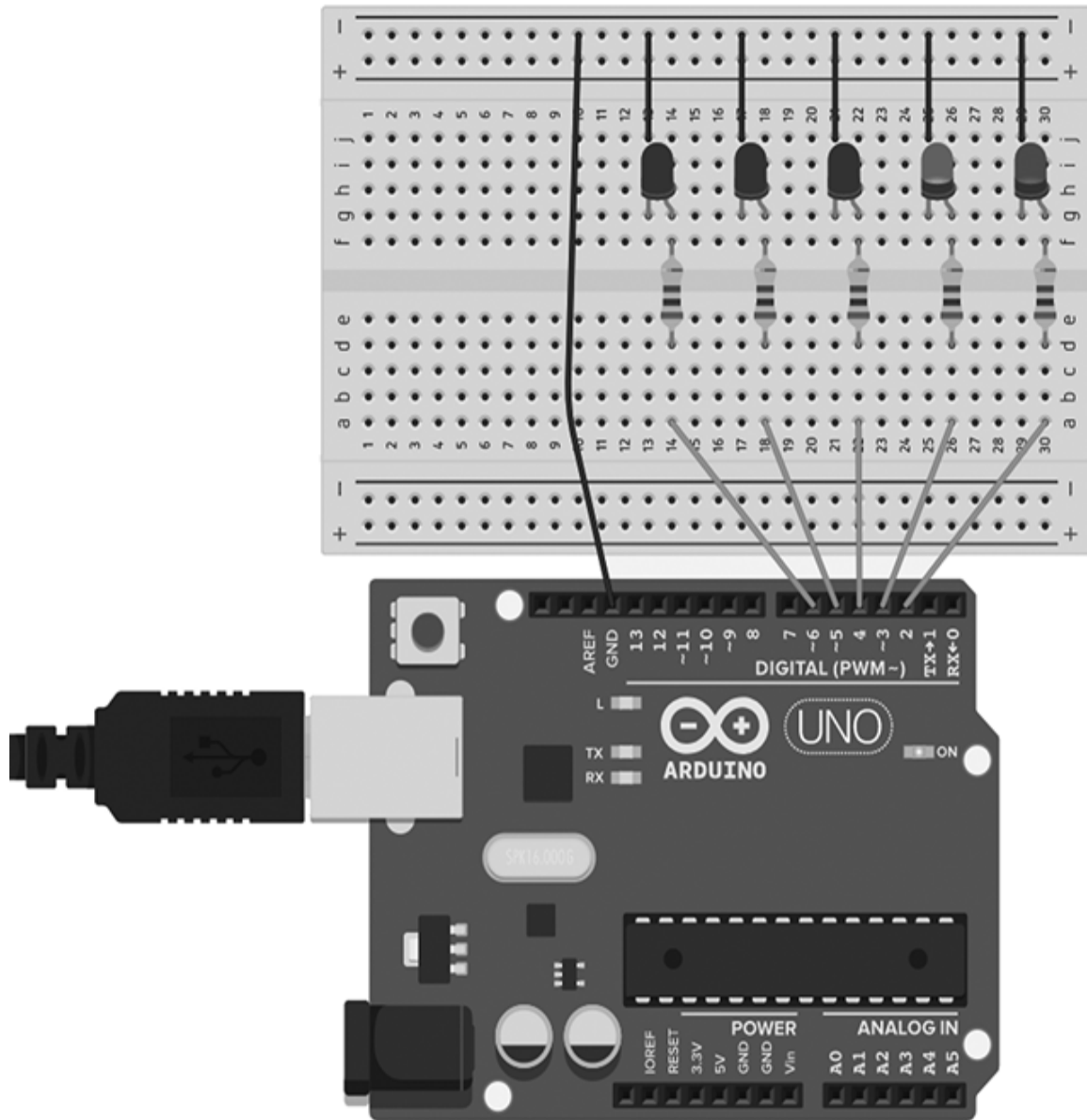


Figure 3-13: Circuit layout for Project 1

The Sketch

Now for our sketch. Enter this code into the IDE:

```
// Project 1 - Creating a Blinking LED Wave
1 void setup()
{
    pinMode(2, OUTPUT); // LED 1 control pin is set up as an
    output
    pinMode(3, OUTPUT); // same for LED 2 to LED 5
    pinMode(4, OUTPUT);
    pinMode(5, OUTPUT);
    pinMode(6, OUTPUT);
}

2 void loop()
{
    digitalWrite(2, HIGH); // Turn LED 1 on
    delay(500); // wait half a second
    digitalWrite(2, LOW); // Turn LED 1 off
    digitalWrite(3, HIGH); // and repeat for LED 2 to 5
    delay(500);
    digitalWrite(3, LOW);
    digitalWrite(4, HIGH);
    delay(500);
    digitalWrite(4, LOW);
    digitalWrite(5, HIGH);
    delay(500);
    digitalWrite(5, LOW);
    digitalWrite(6, HIGH);
    delay(500);
    digitalWrite(6, LOW);
    digitalWrite(5, HIGH);
    delay(500);
    digitalWrite(5, LOW);
    digitalWrite(4, HIGH);
    delay(500);
    digitalWrite(4, LOW);
    digitalWrite(3, HIGH);
    delay(500);
    digitalWrite(3, LOW);
    // The loop() will now loop around and start from the top
    again
}
```

In void setup() at 1, the digital I/O pins are set to outputs, because we want them to send current to the LEDs on demand. We specify when to turn on each LED using the digitalWrite() function in the void loop() section of the sketch at 2.

Running the Sketch

Now connect your Arduino and upload the sketch. After a second or two, the LEDs should blink from left to right and then back again. Success is a wonderful thing—embrace it!

If nothing happens, however, then immediately remove the USB cable from the Arduino and check that you typed the sketch correctly. If you find an error, fix it and upload your sketch again. If your sketch matches exactly and the LEDs still don't blink, check your wiring on the breadboard.

You now know how to make an LED blink with your Arduino, but this sketch is somewhat inefficient. For example, if you wanted to modify it to make the LEDs cycle more quickly, you would need to alter each `delay(500)`. There is a better way.

Using Variables

In computer programs, we can use *variables* to store data. The problem with the sketch for Project 1 as written is that because it doesn't use variables, it's not very flexible. For example, we use the function `delay(500)` to keep the LEDs turned on. If we want to make a change to the delay time, then we have to change each entry manually. To address this problem, we'll create a variable to represent the value for the `delay()` function.

Enter the following line in the Project 1 sketch, above the `void setup()` function and just after the initial comment:

```
int d = 250;
```

This assigns the number 250 to a variable called `d`. The `int` indicates that the variable contains an integer—a whole number between $-32,768$ and $32,767$. Simply put, any integer value has no fraction or decimal places.

Next, change every 500 in the sketch to a `d`. Now when the sketch runs, the Arduino will use the value in `d` for the `delay()` functions. When you upload your sketch after making these changes, the LEDs will turn on and off at a much faster rate, as the delay value is much smaller.

Now, if you want to change the delay, simply change the variable declaration at the start of the sketch. For example, entering 100 for the delay would speed things up even more:

```
int d = 100;
```

Experiment with the sketch, perhaps altering the delays and the sequence of HIGH and LOW. Have some fun with it. Don't disassemble the circuit yet, though; we'll continue to use it with more projects in this chapter.

Project #2: Repeating with for Loops

When designing a sketch, you'll often repeat the same function. You could simply copy and paste the function to duplicate it in a sketch, but that's inefficient and a waste of your Arduino's program memory. Instead, you can use for loops. The benefit of using a for loop is that you can determine how many times the code inside the loop will repeat.

To see how a for loop works, enter the following code as a new sketch:

```
// Project 2 - Repeating with for Loops
int d = 100;

void setup()
{
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);
  pinMode(6, OUTPUT);
}

void loop()
{
  for ( int a = 2; a < 7 ; a++ )
  {
    digitalWrite(a, HIGH);
    delay(d);
    digitalWrite(a, LOW);
    delay(d);
  }
}
```

The for loop will repeat the code within the curly brackets as long as some condition is true. Here we have used a new integer variable, `a`, which starts with the value 2. Every time the code is executed, the `a++` will add 1 to the value of `a`. The loop will continue in this fashion while the value of `a` is less than 7 (the *condition*). Once it is equal to or greater than 7, the Arduino moves on and continues with whatever code comes after the for loop.

The number of loops that a for loop executes can also be set by counting down from a higher number to a lower number. To demonstrate this, add the following loop to the Project 2 sketch after the first for loop:

```
1 for ( int a = 5 ; a > 1 ; a-- )
  {
    digitalWrite(a, HIGH);
    delay(d);
    digitalWrite(a, LOW);
    delay(d);
  }
```

Here, the for loop at 1 sets the value of `a` equal to 5 and then subtracts 1 after every loop due to the `a--`. The loop continues in this manner while the value of `a` is greater than 1 (`a > 1`) and finishes once the value of `a` falls to 1 or less than 1.

We have now re-created Project 1 using less code. Upload the sketch and see for yourself!

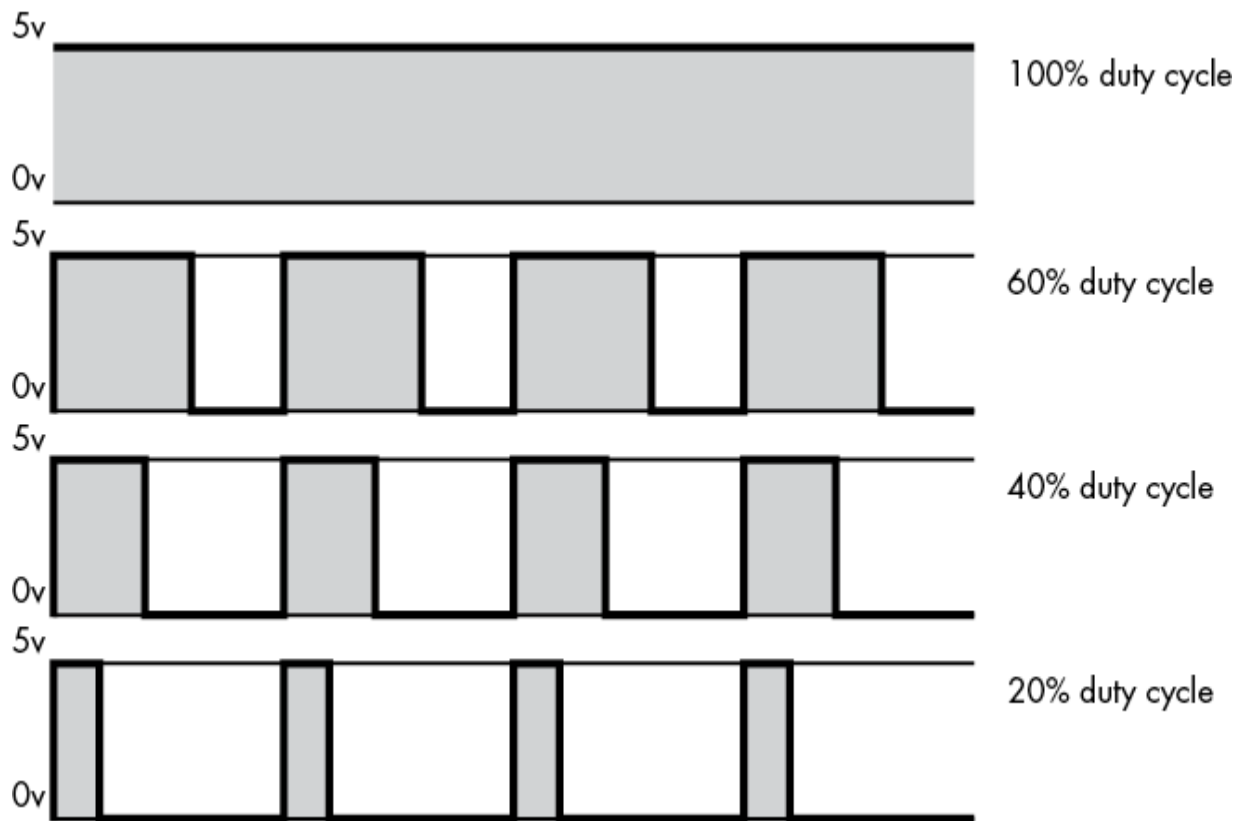
Varying LED Brightness with Pulse-Width Modulation

Rather than just turning LEDs on and off rapidly using `digitalWrite()`, we can define the level of brightness of an LED by adjusting the amount of time between each LED's on and off states using *pulse-width modulation* (*PWM*). PWM can be used to create the illusion that an LED is shining at different levels of brightness by turning the LED on and off rapidly, at around 500 cycles per second. The brightness we perceive is determined by the amount of time the digital output pin is on versus the amount of time it is off—that is, how long the LED is lit or unlit. Because our eyes can't see

flickers faster than 50 cycles per second, the LED appears to have a constant brightness.

The greater the *duty cycle* (the longer the pin is on compared to off in each cycle), the greater the perceived brightness of the LED connected to the digital output pin.

[Figure 3-14](#) shows various PWM duty cycles. The filled-in gray areas represent the amount of time that the light is on. As you can see, the amount of time per cycle that the light is on increases with the duty cycle.



[Figure 3-14](#): Various PWM duty cycles

Only digital pins 3, 5, 6, 9, 10, and 11 on a regular Arduino board can be used for PWM. They are marked on the Arduino board with a tilde (~), as shown in [Figure 3-15](#).



Figure 3-15: The PWM pins are marked with a tilde (~).

To create a PWM signal, we use the function `analogWrite(x, y)`, where `x` is the digital pin and `y` is a value for the duty cycle. `y` can be any value between 0 and 255, where 0 indicates a 0 percent duty cycle and 255 indicates a 100 percent duty cycle.

Project #3: Demonstrating PWM

Now let's try this with our circuit from Project 2. Enter the following sketch into the IDE and upload it to the Arduino:

```
// Project 3 - Demonstrating PWM
int d = 5;
void setup()
{
  pinMode(3, OUTPUT); // LED control pin is 3, a PWM-capable pin
}

void loop()
{
  for ( int a = 0 ; a < 256 ; a++ )
  {
    analogWrite(3, a);
    delay(d);
  }
  for ( int a = 255 ; a >= 0 ; a-- )
  {
    analogWrite(3, a);
    delay(d);
  }
}
```

```
    delay(200);  
}
```

The LED on digital pin 3 will exhibit a “breathing effect” as the duty cycle increases and decreases. In other words, the LED will turn on, increasing in brightness until fully lit, and then reverse until it is dark. Experiment with the sketch and circuit. For example, make all five LEDs breathe at once, or have them do so sequentially.

More Electric Components

You’ll usually find it easy to plan on having a digital output do something without taking into account how much current the control really needs to get the job done. As you create your project, remember that each digital output pin on the Arduino Uno can offer a maximum of 40 mA of current per pin and 200 mA total for all pins. However, the three electronic hardware components discussed next can help you increase the current-handling ability of the Arduino.

WARNING

If you attempt to exceed 40 mA on a single pin, or 200 mA total, then you risk permanently damaging the microcontroller integrated circuit (IC).

The Transistor

Almost everyone has heard of a *transistor*, but most people don’t really understand how it works. In the spirit of brevity, I will keep the explanation as simple as possible. A transistor can turn on or off the flow of a much larger current than the Arduino Uno can handle. We can, however, safely control a transistor using an Arduino digital output pin. A popular transistor is the BC548, shown in [Figure 3-16](#).

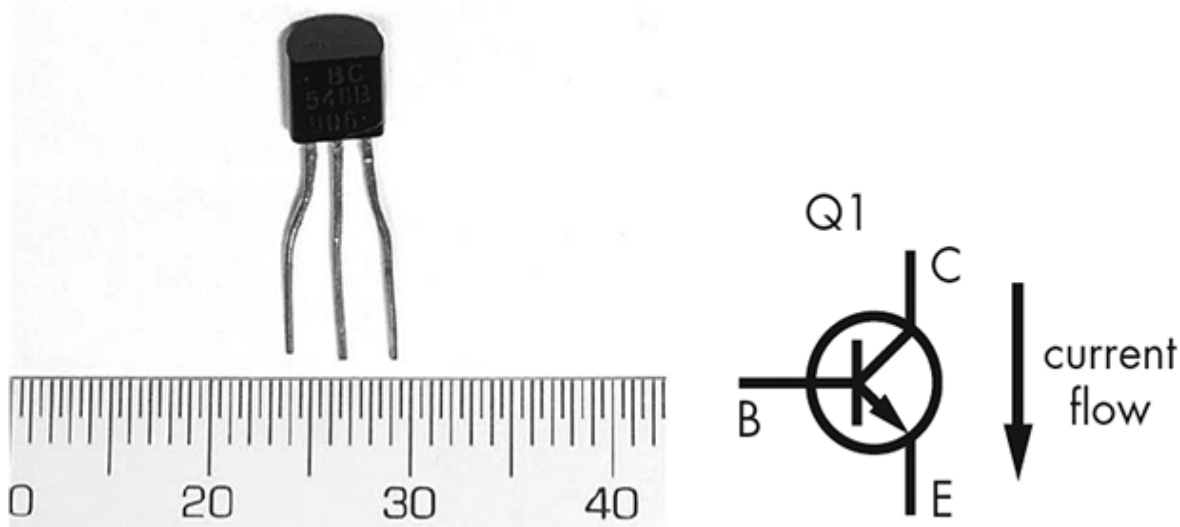


Figure 3-16: A typical transistor: the BC548

Similar to the LED, the transistor's pins have a unique function and need to be connected in the proper orientation. With the flat front of the transistor facing you (as shown on the left of [Figure 3-16](#)), the pins on the BC548 are called, from left to right, the *collector* (C), *base* (B), and *emitter* (E). (Note that this pin order, or *pinout*, is for the BC548 transistor; other transistors may be oriented differently.) When a small current is applied to the base, such as from an Arduino digital I/O pin, the larger current we want to switch enters through the collector. It's combined with the small current from the base before flowing out via the emitter. When the small control current at the base is turned off, no current can flow through the transistor.

The BC548 can switch up to 100 mA of current at a maximum of 30 V—much more than the Arduino's digital output. In projects later in the book, you'll read about transistors in more detail.

NOTE

Always pay attention to the pin order for your particular transistor, because each transistor can have its own orientation.

The Rectifier Diode

The *diode* is a very simple yet useful component that allows current to flow in one direction only. It looks a lot like a resistor, as you can see in [Figure 3-17](#).



[Figure 3-17](#): A 1N4004-type rectifier diode

The projects in this book will use the 1N4004-type rectifier diode. Current flows into the diode via the anode and out through the cathode, which is marked with the ring around the diode's body. These diodes can protect parts of the circuit against reverse current flow, but there is a price to pay: diodes also cause a drop in the voltage of around 0.7 V. The 1N4004 diode is rated to handle 1 A and 400 V, much higher than we will be using. It's a tough, common, and low-cost diode.

The Relay

Relays are used for the same reason as transistors—to control a large current and voltage. A relay has the advantage of being *electrically isolated* from the control circuit, allowing the Arduino to switch very large currents and voltages without actually coming into contact with those voltages, which could damage it. Inside the relay is an interesting pair of items: mechanical switch contacts and a low-voltage coil of wire, as shown in [Figure 3-18](#).

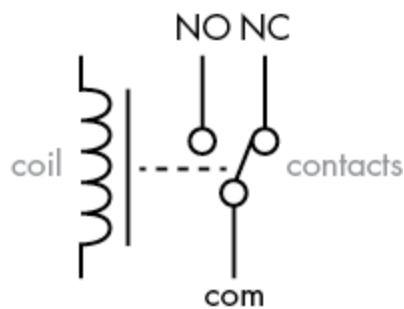


Figure 3-18: Inside a typical relay

When a current is applied to the relay, the coil becomes an electromagnet and attracts a bar of metal that acts just like the toggle of a switch. The magnet pulls the bar in one direction when on and lets it fall back when off, thereby turning it on or off as current is applied to and removed from the coil. This movement has a distinctive “click” that you might recognize from the turn signal in older cars.

Higher-Voltage Circuits

Now that you understand a bit about the transistor, rectifier diode, and relay, let’s use them together to control higher currents and voltages. For example,

you may wish to turn a large motor on or off. Connecting the components is simple, as shown in [Figure 3-19](#).

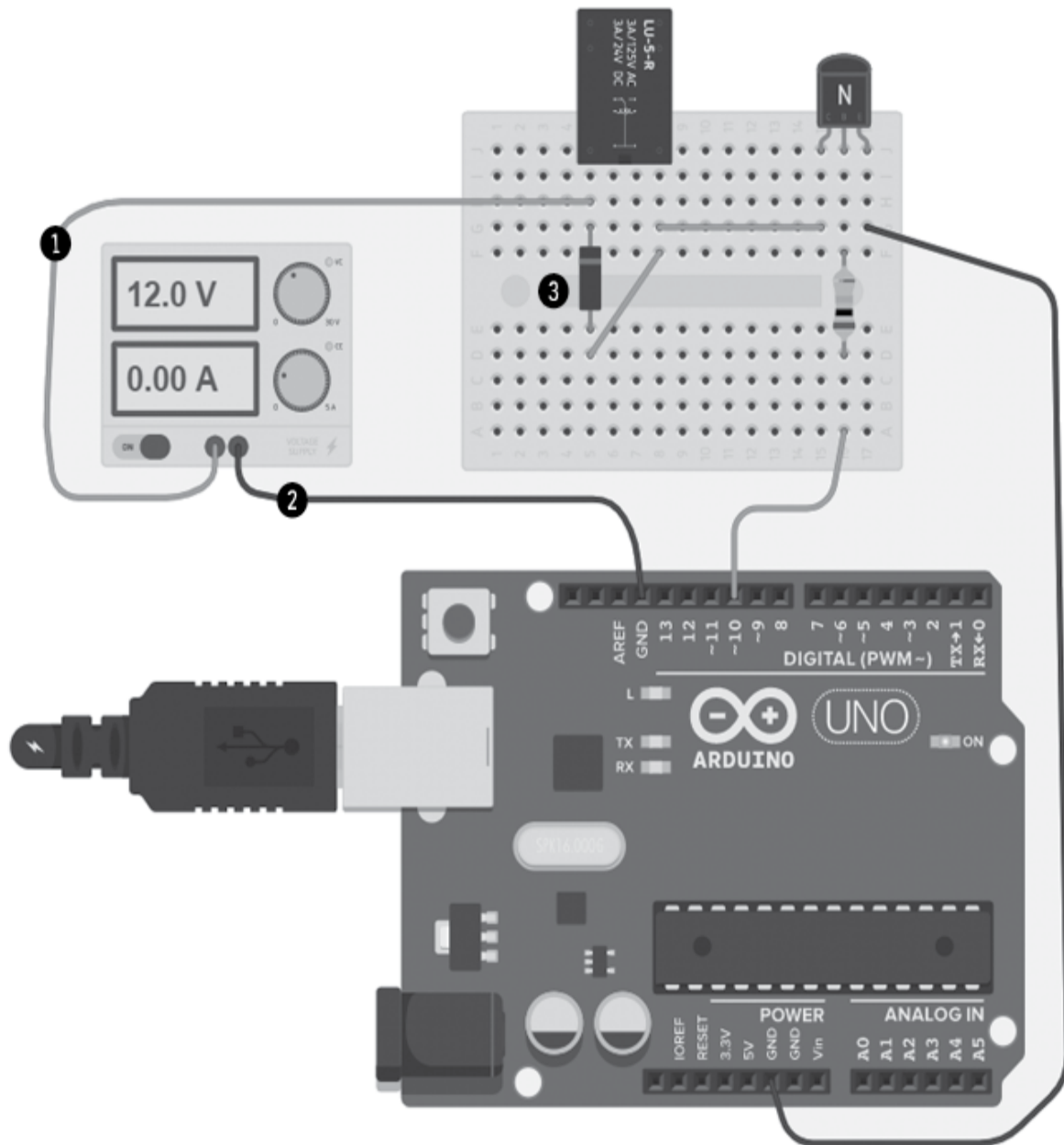


Figure 3-19: A relay control circuit

This simple example circuit controls a relay that has a 12 V coil. One use for this circuit might be to control a lamp or cooling fan connected to the relay switching contacts. The Arduino's digital pin 10 is connected to the transistor's base via a 1 kΩ resistor. The transistor controls the current

through the relay's coil by switching it on and off. Remember that the pins are *C*, *B*, and then *E* when looking at the flat surface of the transistor. The object on the left of the breadboard at 1 represents a 12 V power supply for the relay coil. The negative or ground at 2 from the 12 V supply, the transistor's emitter pin, and Arduino GND are all connected together. Finally, a 1N4004 rectifier diode is connected across the relay's coil at 3, with the cathode on the positive supply side. You can check the relay's data sheet to determine the pins for the contacts and to connect the controlled item appropriately.

The diode is in place to protect the circuit. When the relay coil changes from on to off, stray current remains briefly in the coil and becomes a high-voltage spike that has to go somewhere. The diode allows the stray current to loop around through the coil until it is dissipated as a tiny amount of heat. It prevents the turn-off spike from damaging the transistor or Arduino pin.

WARNING

If you want to control wall power electricity (110–250 V) at a high current with a relay, contact a licensed electrician to complete this work for you. Even the slightest mistake can be fatal.

Looking Ahead

And now Chapter 3 draws to a close. I hope you had fun trying out the examples and experimenting with LED effects. In this chapter, you got to create blinking LEDs on the Arduino in various ways, did a bit of hacking, and learned how functions and loops can be used to efficiently control components connected to the Arduino. Studying this chapter has set you up for more success in the forthcoming chapters.

Chapter 4 will be a lot of fun. You will create some more advanced projects, including traffic lights, a thermometer, a battery tester, and more—so when you're ready to take it to the next level, turn the page!

4

BUILDING BLOCKS

In this chapter you will

Learn how to read schematic diagrams, the language of electronic circuits

Be introduced to the capacitor

Work with input pins

Use arithmetic and test values

Make decisions with `if` statements

Learn the difference between analog and digital

Measure analog voltage sources at different levels of precision

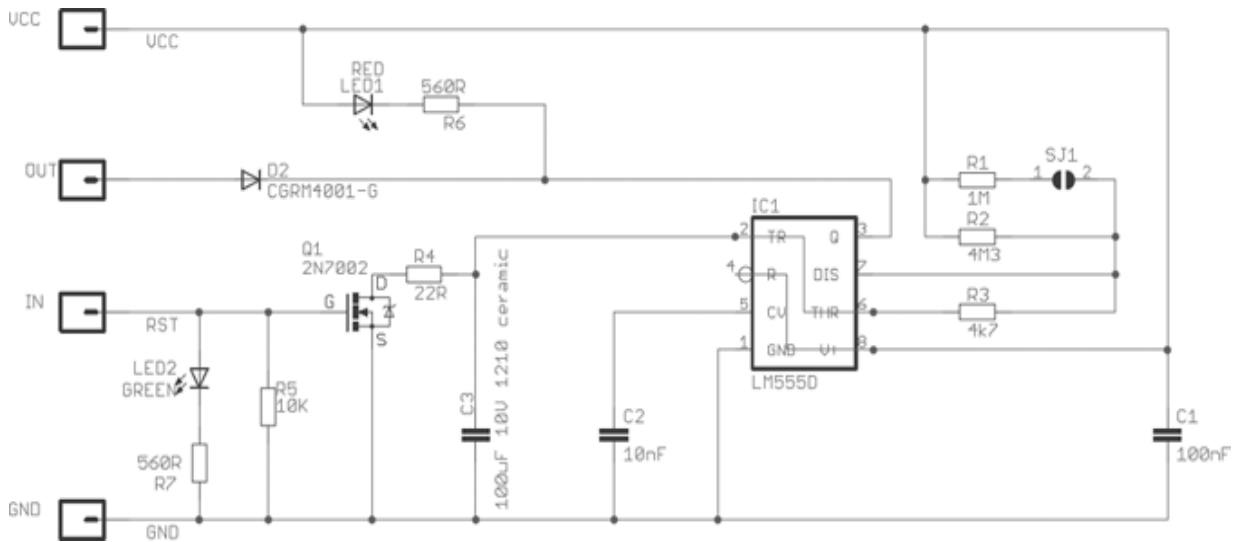
Be introduced to variable resistors, piezoelectric buzzers, and temperature sensors

Consolidate your knowledge by creating traffic lights, a battery tester, and a thermometer

The information in this chapter will help you understand an Arduino's potential. We'll continue to learn about electronics, including how to read schematic diagrams (the "road maps" of electronic circuits). We'll also explore some new components and the types of signals that we can measure. Then we'll discuss additional Arduino functions, such as storing values, performing mathematical operations, and making decisions. Finally, we'll examine a few more components and put them to use in some useful projects.

Using Schematic Diagrams

Chapter 3 described how to build a circuit using physical layout diagrams to represent the breadboard and components mounted on it. Although such physical layout diagrams may seem like the easiest way to diagram a circuit, you'll find that as more components are added, diagrams that are direct representations can become a real mess. Because our circuits are about to get more complicated, we'll start using *schematic diagrams* (also known as *circuit diagrams*) to illustrate them, such as the one shown in [Figure 4-1](#).



[Figure 4-1](#): Example of a schematic diagram

Schematics are simply circuit road maps that show the path of the electrical current flowing through various components. Instead of showing components and wires, a schematic uses symbols and lines.

Identifying Components

Once you know what the symbols mean, reading a schematic is easy. To begin, let's examine the symbols for the components we've already used.

The Arduino

[Figure 4-2](#) shows a symbol for the Arduino itself. As you can see, all of the Arduino's connections are displayed and neatly labeled.

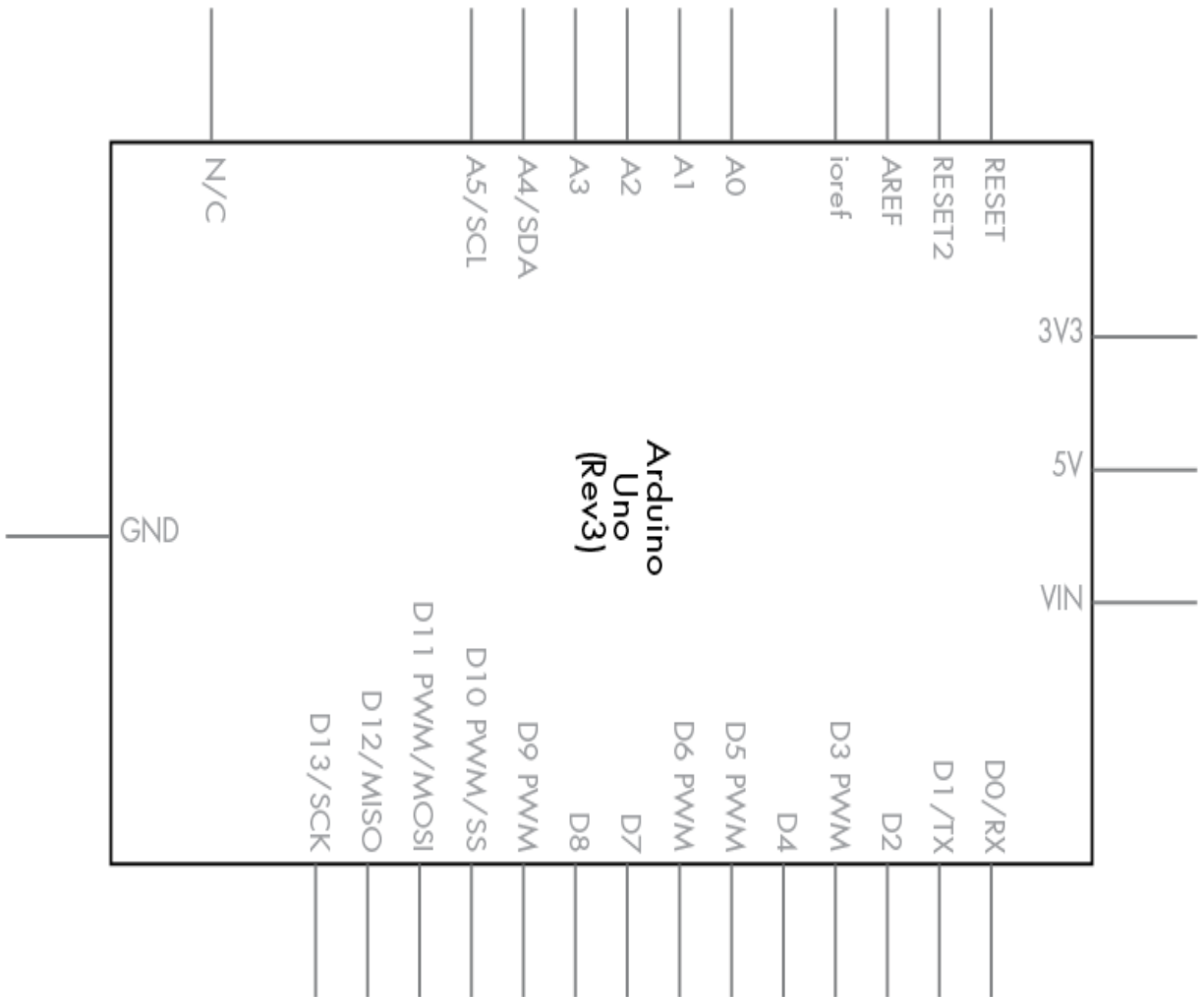


Figure 4-2: Arduino Uno symbol

The Resistor

The resistor symbol is shown in [Figure 4-3](#).

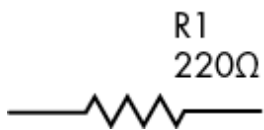


Figure 4-3: Resistor symbol

It's good practice to display the resistor value and part designator along with the resistor symbol (220 Ω and R1 in this case). This makes life a lot easier for everyone trying to make sense of the schematic (including you). Often you may see ohms written as *R* instead—for example, 220 R.

The Rectifier Diode

The rectifier diode symbol is shown in [Figure 4-4](#).

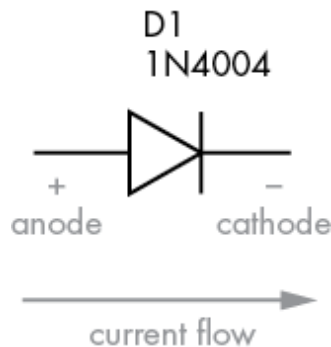


Figure 4-4: Rectifier diode symbol

Recall from Chapter 3 that rectifier diodes are polarized and current flows from the anode to the cathode. In the symbol shown in [Figure 4-4](#), the anode is on the left and the cathode is on the right. An easy way to remember this is to think of current flowing toward the point of the triangle only. Current cannot flow the other way because the vertical bar “stops” it.

The LED

The LED symbol is shown in [Figure 4-5](#).

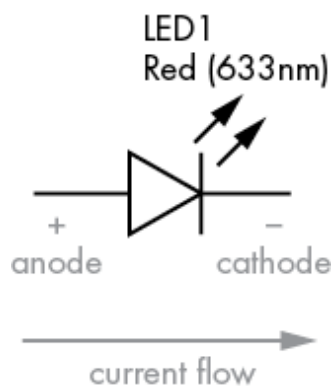


Figure 4-5: LED symbol

All members of the diode family share a common symbol: the triangle and vertical line. However, LED symbols show two parallel arrows pointing away from the triangle to indicate that light is being emitted.

The Transistor

The transistor symbol is shown in [Figure 4-6](#). We'll use this to represent our BC548.

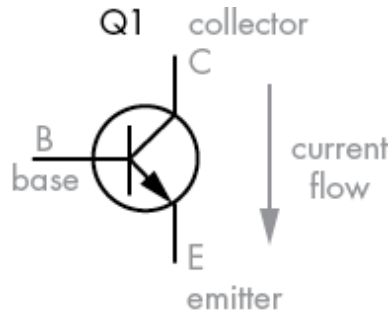


Figure 4-6: Transistor symbol

The vertical line at the top of the symbol (labeled *C*) represents the collector, the horizontal line at the left represents the base (labeled *B*), and the bottom line represents the emitter (labeled *E*). The arrow inside the symbol, pointing down and to the right, tells us that this is an *NPN*-type transistor, because *NPN* transistors allow current to flow from the collector to the emitter. (*PNP*-type transistors allow current to flow from the emitter to the collector.)

When numbering transistors we use the letter *Q*, just as we use *R* to number resistors.

The Relay

The relay symbol is shown in [Figure 4-7](#).

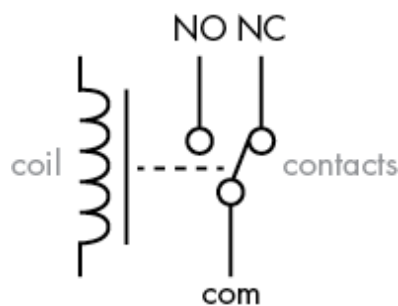


Figure 4-7: Relay symbol

Relay symbols can vary in many ways and may have more than one set of contacts, but all relay symbols share certain elements in common. The first is

the *coil*, which is the curvy vertical line at the left. The second element is the relay *contacts*. The *COM* (for common) contact is often used as an input, and the contacts marked *NO* (normally open) and *NC* (normally closed) are often used as outputs.

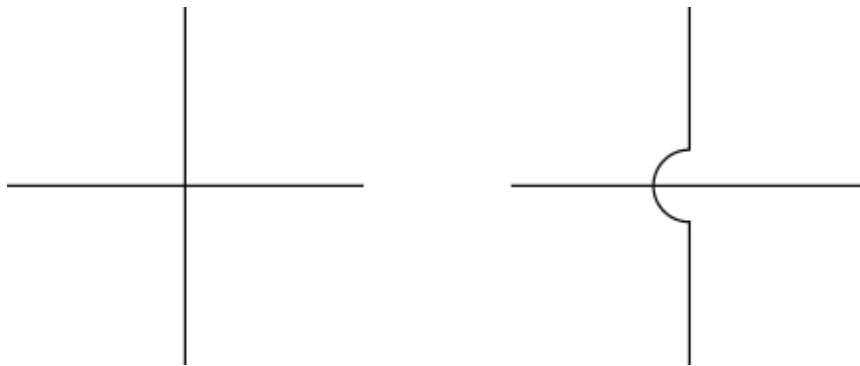
The relay symbol is always shown with the relay in the off state and the coil not *energized*—that is, with the COM and NC pins connected. When the relay coil is energized, the COM and NO pins will be connected in the symbol.

Wires in Schematics

When wires cross or connect in schematics, they are drawn in particular ways, as shown in the following examples.

Crossing but Not Connected Wires

When two wires cross but are not connected, the crossing can be represented in one of two ways, as shown in [Figure 4-8](#). There is no one right way; it's a matter of preference.



[Figure 4-8](#): Non-connecting crossed wires

Connected Wires

When wires are meant to be physically connected, a *junction dot* is drawn at the point of connection, as shown in [Figure 4-9](#).

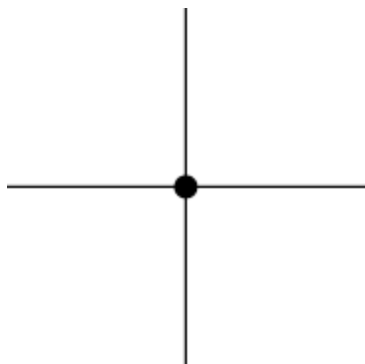


Figure 4-9: Two wires that are connected

Wire Connected to Ground

When a wire is connected back to ground (GND), the standard method is to use the symbol shown in [Figure 4-10](#).

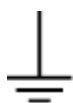


Figure 4-10: The GND symbol

The GND symbol at the end of a line in a schematic in this book tells you that the wire is physically connected to the Arduino GND pin.

Dissecting a Schematic

Now that you know the symbols for various components and their connections, let's dissect the schematic we would draw for Project 1, on page 33 in Chapter 3. Recall that you made five LEDs blink backward and forward.

Compare the schematic shown in [Figure 4-11](#) with Figure 3-13 on page 34, and you'll see that using a schematic is a much easier way to describe a circuit.

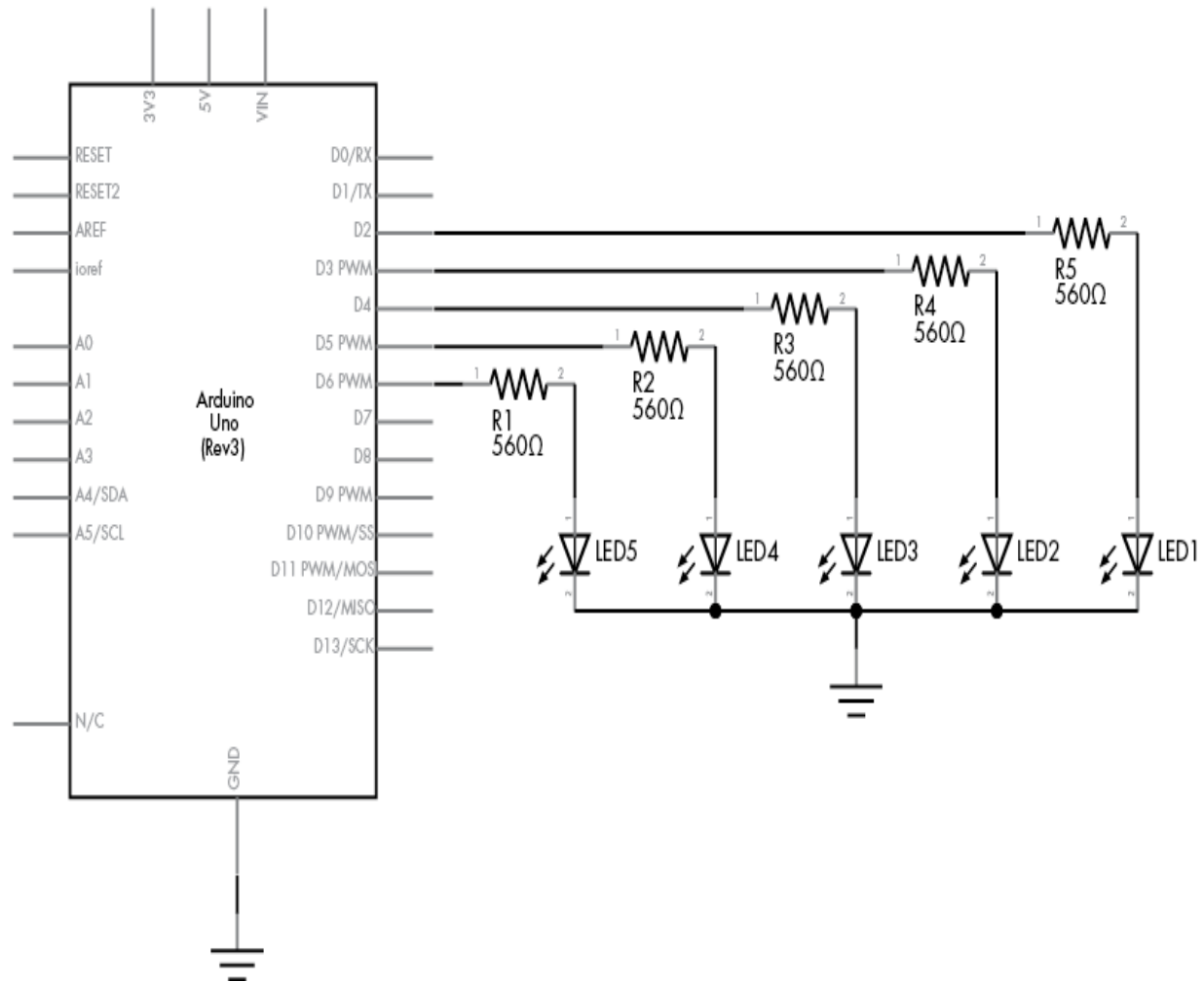


Figure 4-11: Schematic for Project 1

From now on, we'll use schematics to describe circuits, and I'll show you the symbols for new components as they're introduced.

NOTE

If you'd like to create your own computer-drawn schematics, try the *Fritzing* application, available at minimal cost from <http://www.fritzing.org/>.

The Capacitor

A *capacitor* is a device that holds an electric charge. It consists of two conductive plates sandwiching an insulating layer that allows an electric charge to build up between the plates. When the current is stopped, the charge remains and can flow out of the capacitor (called *discharging* the capacitor) as soon as the charge voltage stored in the capacitor is presented with a new path for the current to take.

Measuring the Capacity of a Capacitor

The amount of charge that a capacitor can store is measured in *farads*, and one farad is actually a very large amount. Therefore, you will generally find capacitors with values measured in picofarads or microfarads. One *picofarad* (pF) is 0.000000000001 of a farad, and one *microfarad* (μF) is 0.000001 of a farad. Capacitors are also manufactured to accept certain voltage maximums. In this book we'll be working with low voltages only, so we won't be using capacitors rated at greater than 10 V or so; it's generally fine, however, to use higher-voltage capacitors in lower-voltage circuits. Common voltage ratings are 10, 16, 25, and 50 V.

Reading Capacitor Values

Reading the value of a ceramic capacitor takes some practice, because the value is printed in a sort of code. The first two digits represent the value in picofarads, and the third digit is the multiplier in tens. For example, the capacitor shown in [Figure 4-12](#) is labeled *104*. This equates to 10 followed by four zeros, or 100,000 pF (which is 100 nanofarads [nF] or 0.1 μF).

NOTE

The conversions between units of measure can be a little confusing, but you can print an excellent conversion chart from <http://www.justradios.com/uFnFpF.html>.

Types of Capacitors

Our projects will use two types of capacitors: ceramic and electrolytic.

Ceramic Capacitors

Ceramic capacitors, such as the one shown in [Figure 4-12](#), are very small and therefore hold a small amount of charge. They are not polarized and can be used for current flowing in either direction. The schematic symbol for a non-polarized capacitor is shown in [Figure 4-13](#).

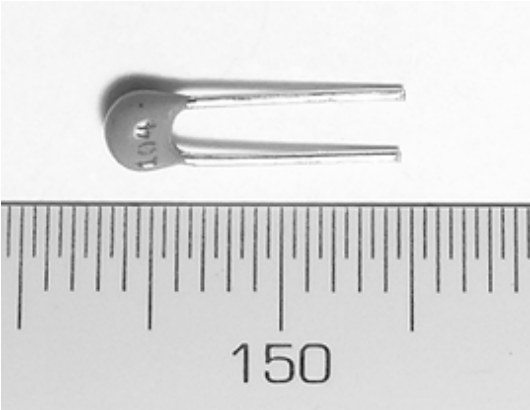


Figure 4-12: A 0.1 μF ceramic capacitor

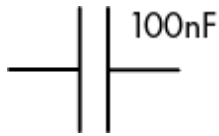


Figure 4-13: Non-polarized capacitor schematic symbol, with the capacitor's value shown at the upper right

Ceramic capacitors work beautifully in high-frequency circuits because they can charge and discharge very quickly due to their small capacitance.

Electrolytic Capacitors

Electrolytic capacitors, like the one shown in [Figure 4-14](#), are physically larger than ceramic types, offer increased capacitance, and are polarized. A marking on the cover shows either the positive (+) side or the negative (–) side. In [Figure 4-14](#), you can see the stripe and the small negative (–) symbol that identifies the negative side. Like resistors, capacitors also have a level of tolerance with their values. The capacitor in [Figure 4-14](#) has a tolerance of 20 percent and a capacitance of 100 μF .

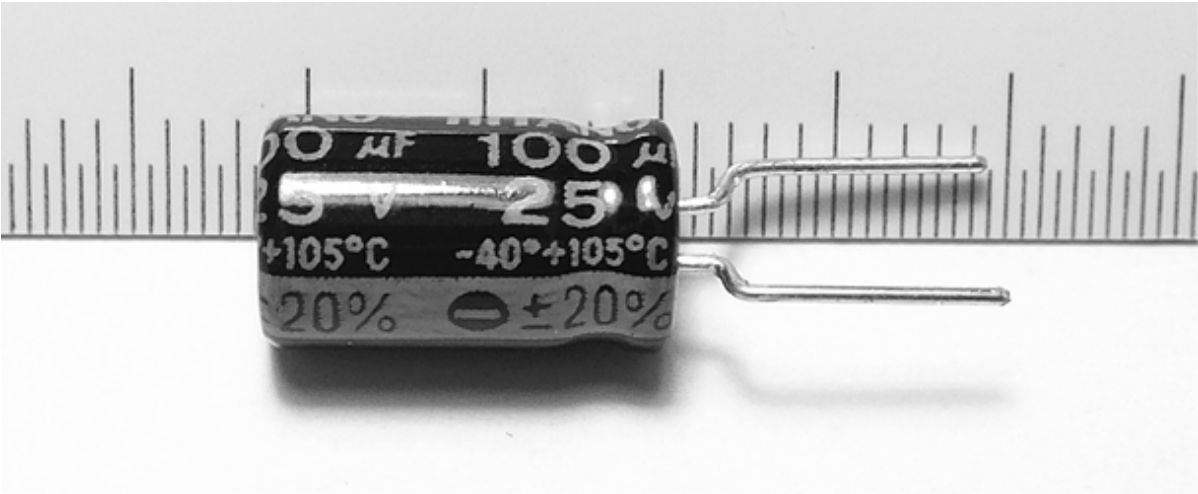


Figure 4-14: An electrolytic capacitor

The schematic symbol for electrolytic capacitors, shown in [Figure 4-15](#), includes the + symbol to indicate the capacitor's polarity.

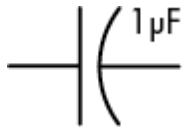


Figure 4-15: Polarized capacitor schematic symbol

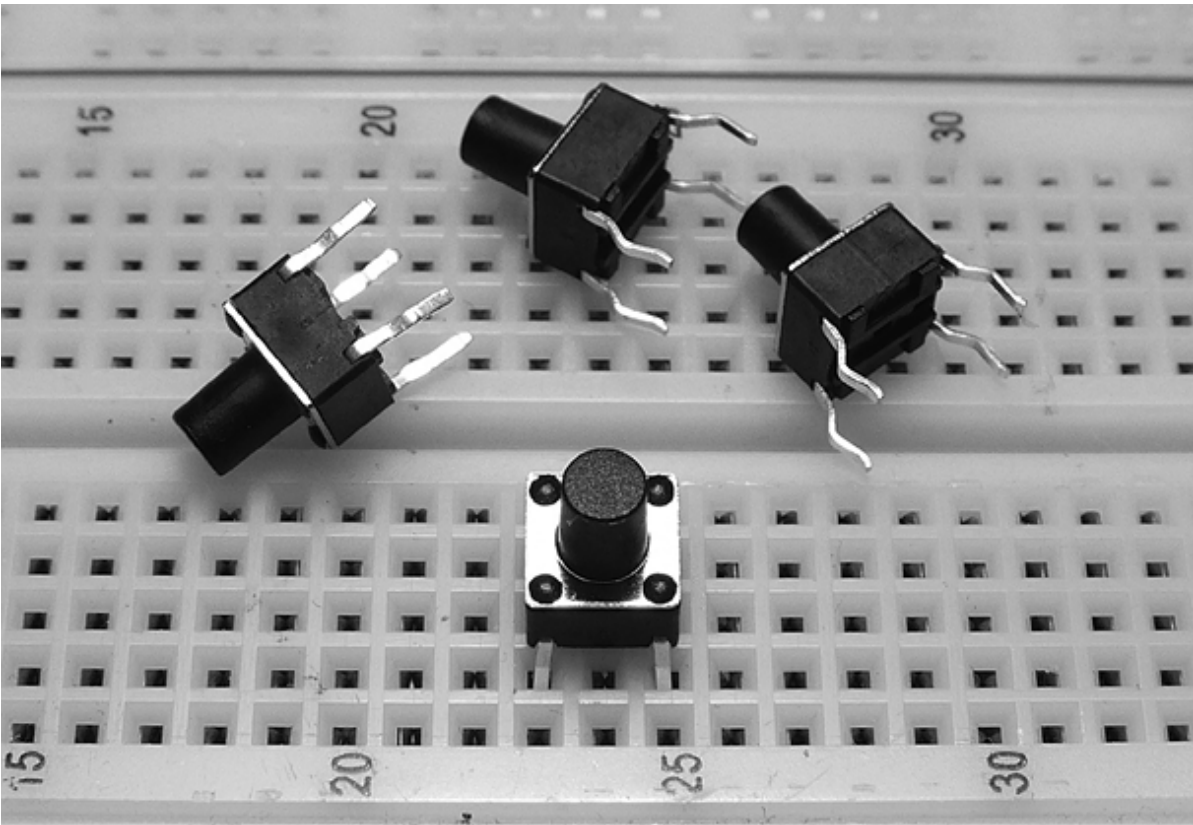
Electrolytic capacitors are often used to store larger electric charges and to smooth power supply voltages. Like a small temporary battery, they can smooth out the power supply and provide stability near circuits or parts that draw high currents quickly from the supply. This prevents unwanted dropouts and noise in your circuits. Luckily, the values of the electrolytic capacitor are printed clearly on the outside and don't require decoding or interpretation.

You already have some experience generating basic forms of output using LEDs with your Arduino. Now it's time to learn how to send input from the outside world into your Arduino using digital inputs, and to make decisions based on that input.

Digital Inputs

In Chapter 3, we used digital I/O pins as outputs to turn LEDs on and off. We can use these same pins to accept input from users—as long as we limit our information to two states, high and low.

The simplest form of digital input is a *push button*; several push buttons are shown in [Figure 4-16](#). You can insert one of these directly into your solderless breadboard and wire it to an Arduino pin. When the button is pressed, current flows through the switch and into the digital input pin, which detects the presence of the voltage.



[Figure 4-16](#): Basic push buttons on a breadboard

Notice that the button at the bottom of the figure is inserted into the breadboard, bridging rows 23 and 25. When the button is pressed, it connects the two rows. The schematic symbol for this push button is shown in [Figure 4-17](#). The symbol represents the two sides of the button, which are numbered with the prefix S. When the button is pressed, the line bridges the two halves and allows voltage or current through.

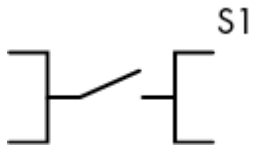
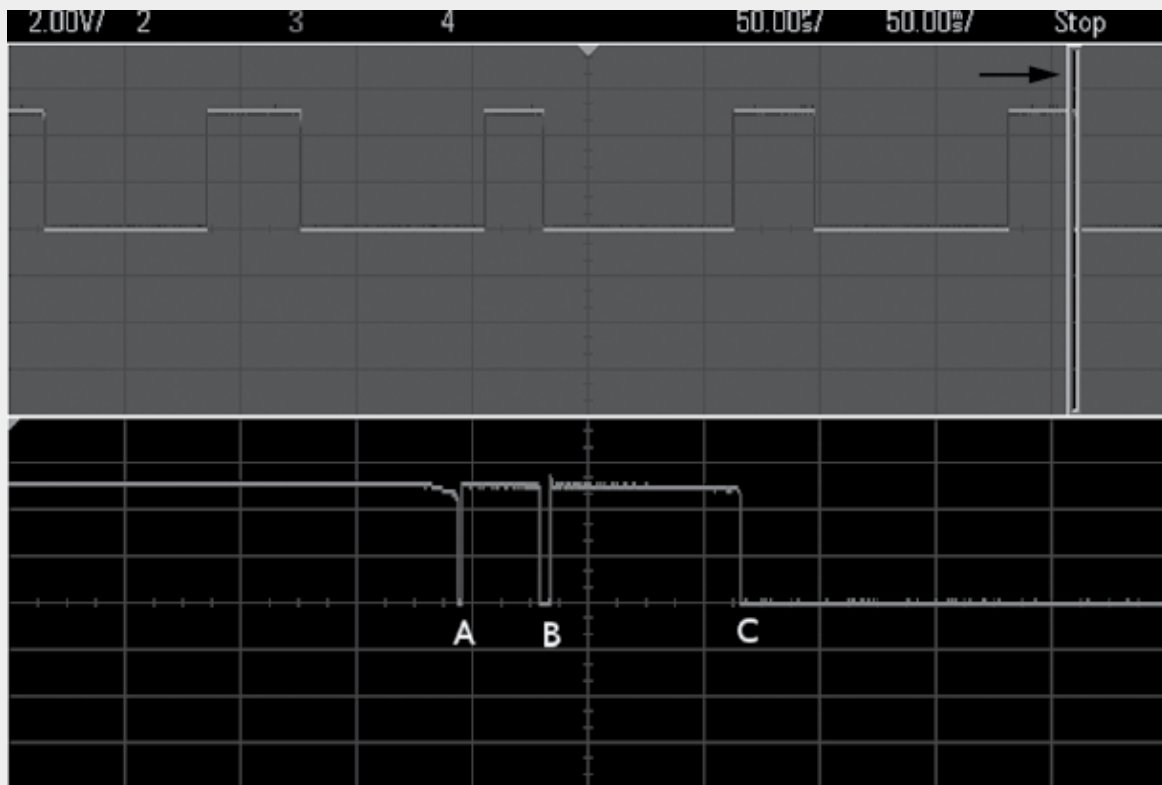


Figure 4-17: Push button schematic symbol

MEASURING SWITCH BOUNCE WITH A DIGITAL STORAGE OSCILLOSCOPE

Push buttons exhibit a phenomenon called *switch bounce*, or *bouncing*, which refers to a button's tendency to turn on and off several times after being pressed only once by the user. This phenomenon occurs because the metal contacts inside a push button are so small that they can vibrate after the button has been released, thereby switching on and off again very quickly.

Switch bounce can be demonstrated with a *digital storage oscilloscope (DSO)*, a device that displays the change in a voltage over a period of time. For example, consider [Figure 4-18](#), a DSO displaying a switch bounce.



[Figure 4-18](#): Measuring switch bounce

The top half of the display in [Figure 4-18](#) shows the results of pressing a button several times. When the voltage line indicated by the arrow is at the higher horizontal position (5 V), the button is in the *on* state, and the voltage is connected through it. Underneath the word *Stop* is a slice of time just after the button was switched off, as shown by two gray vertical lines.

The button voltage during this interval is magnified in the bottom half of the screen. At point A, the button is released by the user, and the line drops to 0 V. However, due to physical vibration, the button returns almost immediately to the higher 5 V position until point B, where it vibrates off and then on again until point C, where it settles in the low

(off) state. In effect, instead of relaying one button press to our Arduino, we have unwittingly sent three.

Project #4: Demonstrating a Digital Input

Our goal in this project is to create a button that turns on an LED for half a second when pressed.

The Algorithm

Here is our algorithm:

- . Test whether the button has been pressed.
- . If the button has been pressed, turn on the LED for half a second and then turn it off.
- . If the button has not been pressed, do nothing.
- . Repeat indefinitely.

The Hardware

Here's what you'll need to create this project:

One push button

One LED

One 560 Ω resistor

One 10 k Ω resistor

One 100 nF capacitor

Various connecting wires

One breadboard

Arduino and USB cable

The Schematic

First, we create the circuit on the breadboard with the schematic shown in [Figure 4-19](#). Notice that the 10 k Ω resistor is connected between GND and digital pin 7. We call this a *pull-down resistor*, because it pulls the voltage at the digital pin almost to zero. Furthermore, by adding a 100 nF capacitor across the 10 k Ω resistor, we create a simple *debounce* circuit to help filter out the switch bounce. When the button is pressed, the digital pin goes immediately to high. But when the button is released, digital pin 7 is pulled down to GND via the 10 k Ω resistor, and the 100 nF capacitor creates a small delay. This effectively covers up the bouncing pulses by slowing the drop of the voltage to GND, thereby eliminating most of the false readings due to floating voltage and erratic button behavior.

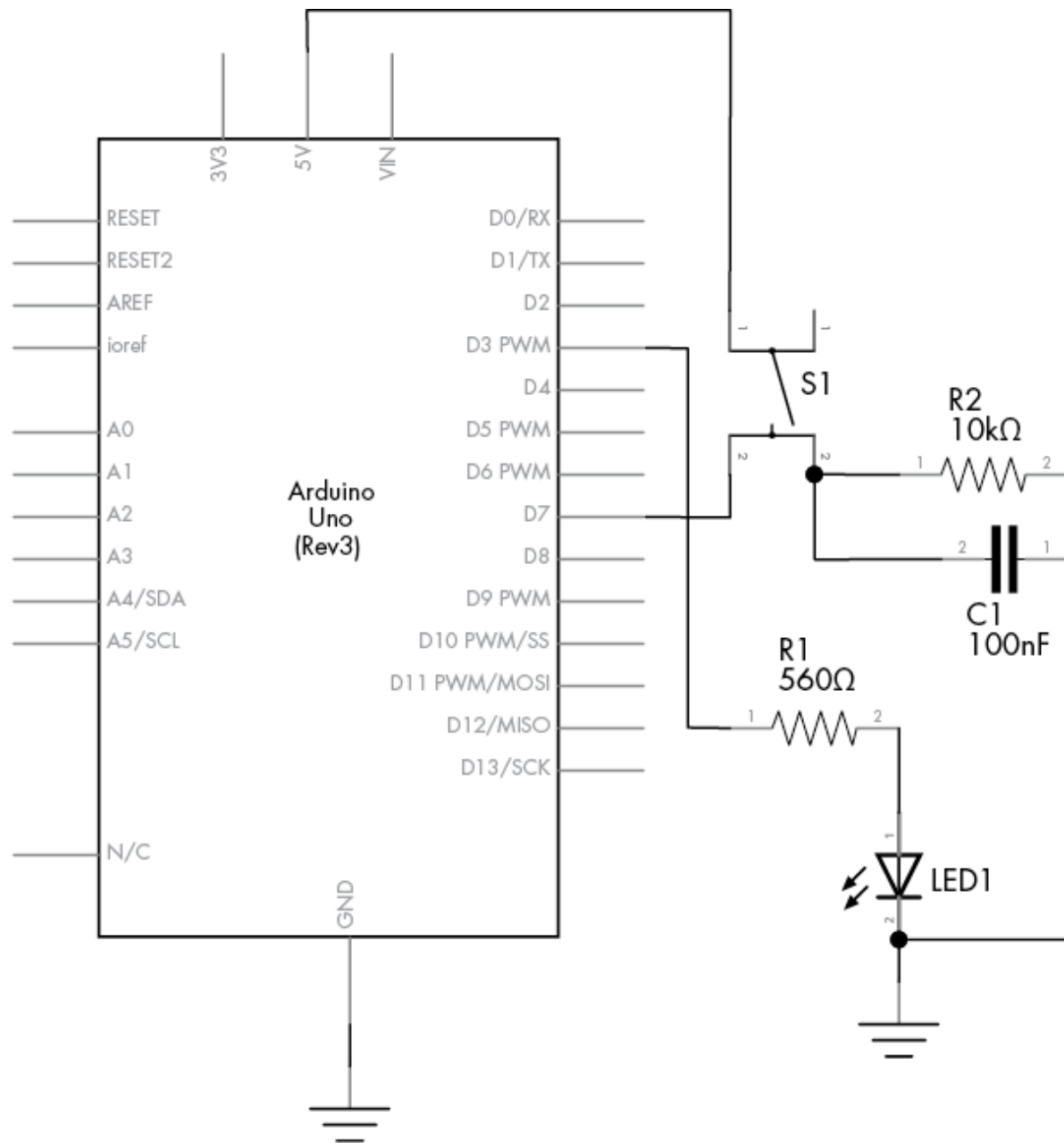


Figure 4-19: Schematic for Project 4

Because this is the first time you're building a circuit with a schematic, follow these step-by-step instructions as you walk through the schematic; this should help you understand how the components connect:

Insert the push button into the breadboard, as shown in [Figure 4-20](#).

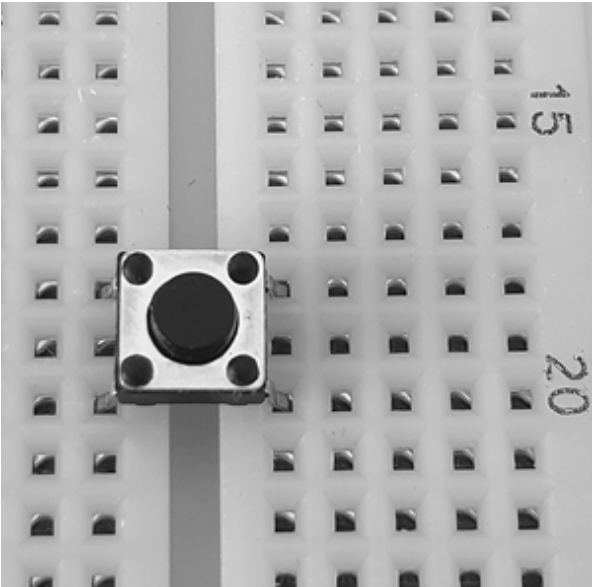


Figure 4-20: The push button inserted into the breadboard

Now insert the 10 k Ω resistor, a short link wire, and the capacitor, as shown in [Figure 4-21](#).

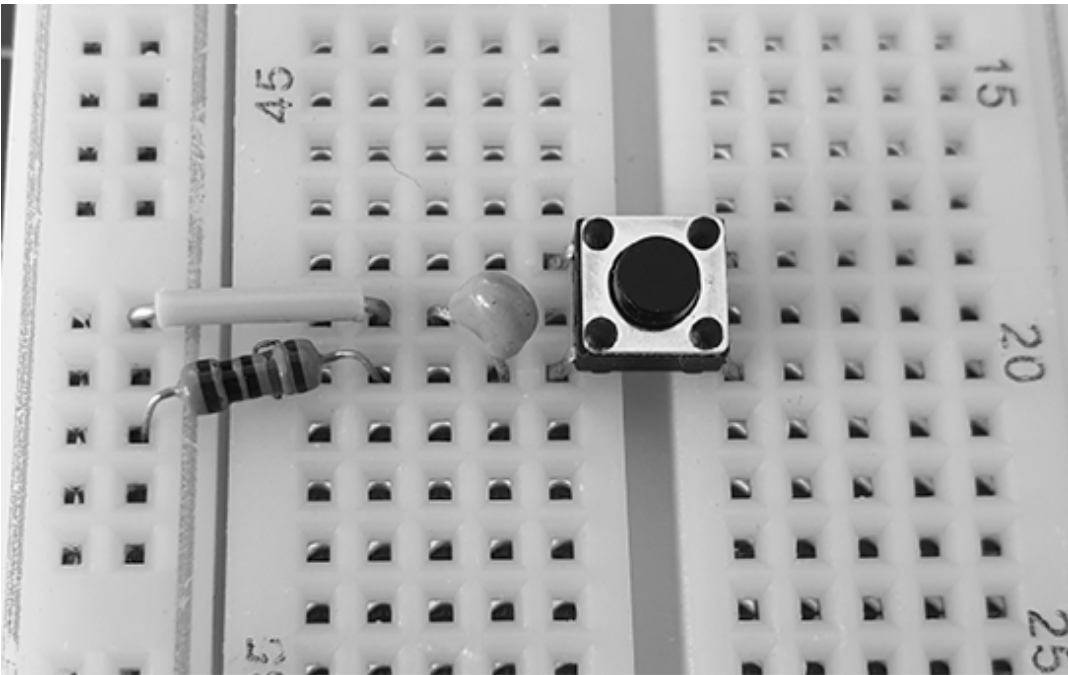


Figure 4-21: Adding the 10 k Ω resistor and the capacitor

Connect one wire from the Arduino 5 V pin to the upper-right row for the button on the breadboard. Connect another wire from the Arduino GND pin

to the same vertical row that connects to the left-hand sides of the wire link and the resistor. This is shown in [Figure 4-22](#).

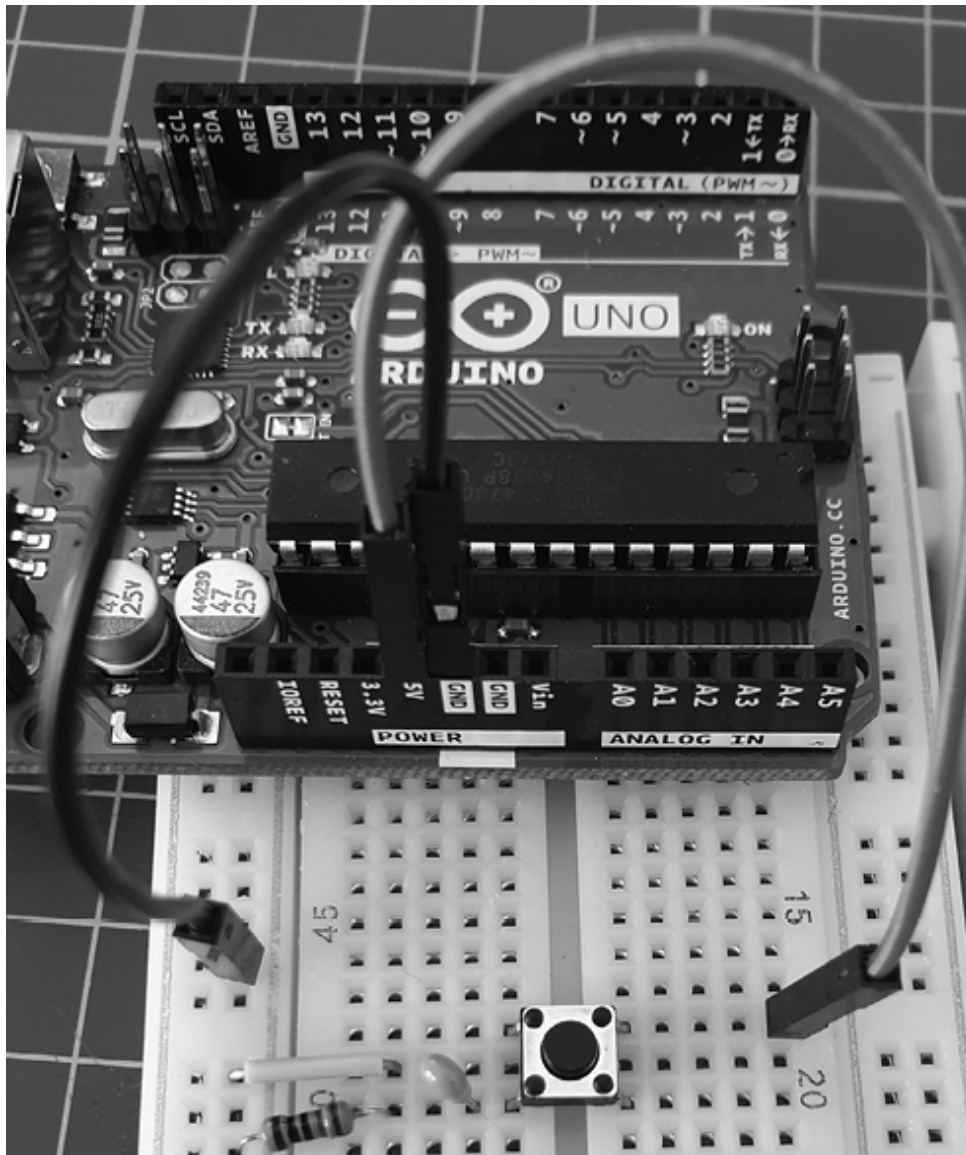


Figure 4-22: Connecting the 5 V (red) and GND (black) wires

. Run a wire from Arduino digital pin 7 to the lower-right row for the button on the breadboard, as shown in [Figure 4-23](#).

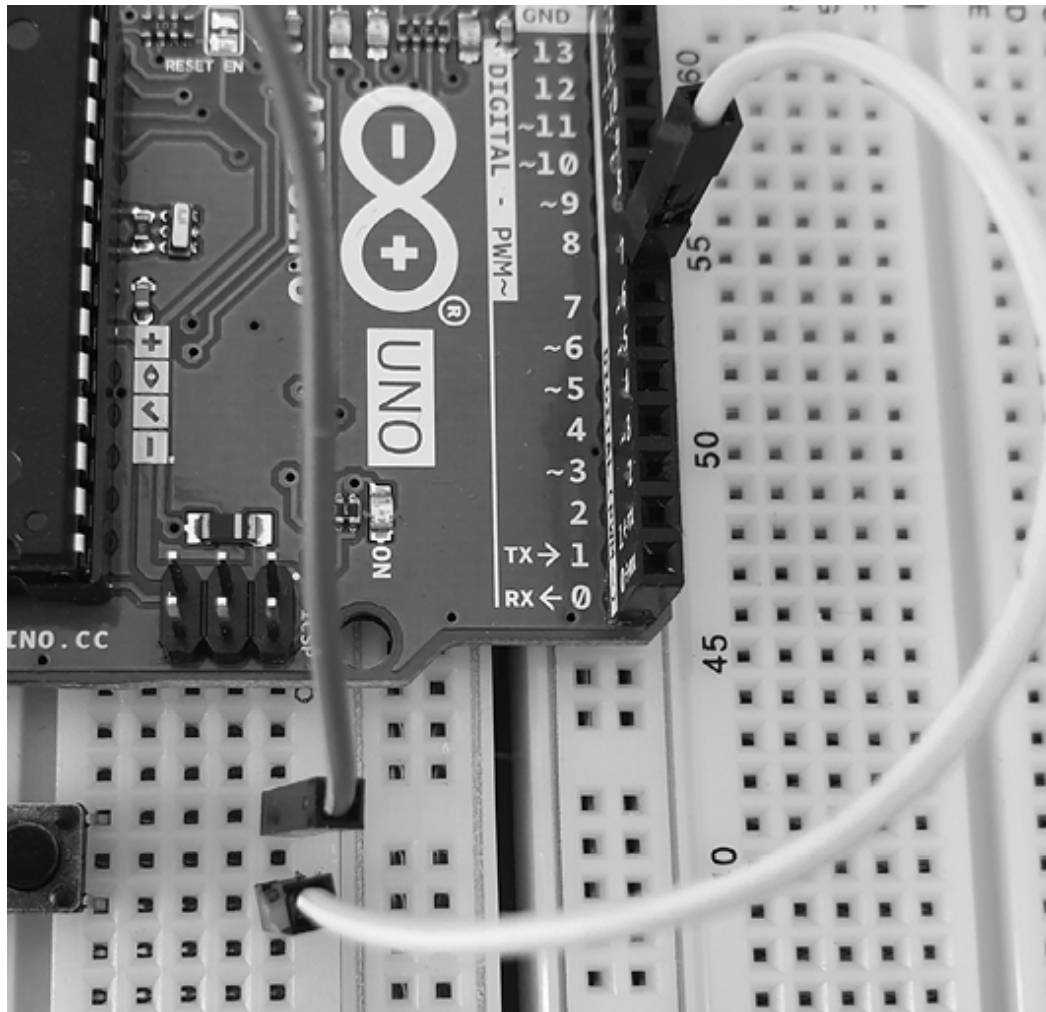


Figure 4-23: Connecting the button to the digital input

Insert the LED into the breadboard with the short leg (the cathode) connected to the GND column and the long leg (the anode) in a row to the right. Next, connect the 560 Ω resistor to the right of the LED, as shown in [Figure 4-24](#).

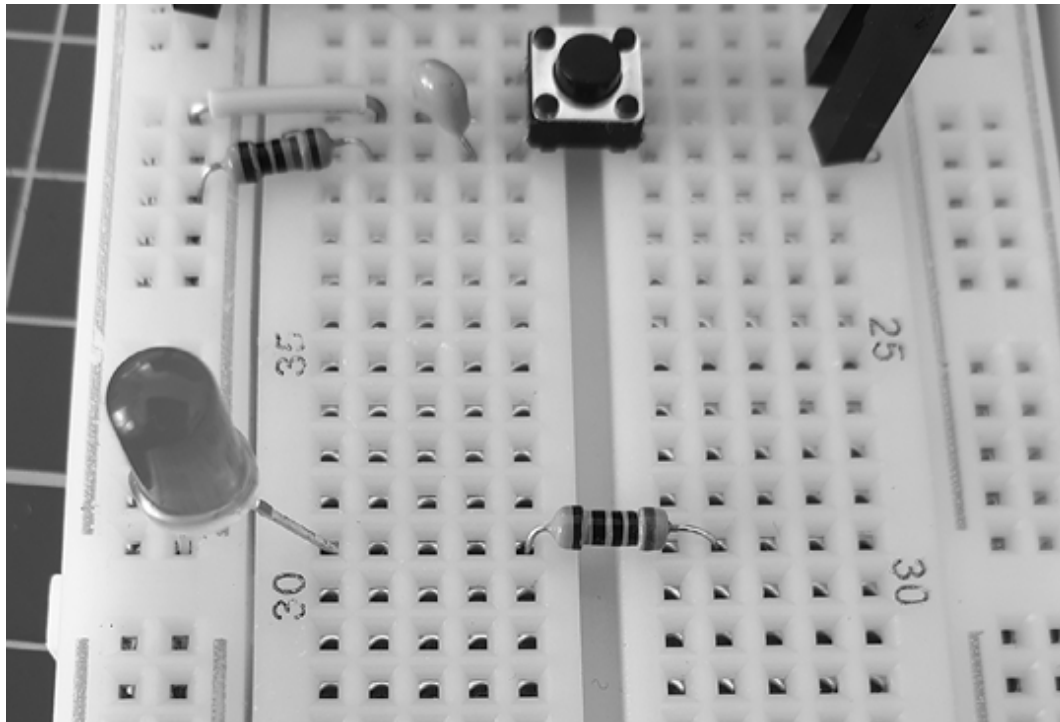


Figure 4-24: Inserting the LED and 560 Ω resistor

Connect a wire from the right side of the 560 Ω resistor to Arduino digital pin 3, as shown in [Figure 4-25](#).

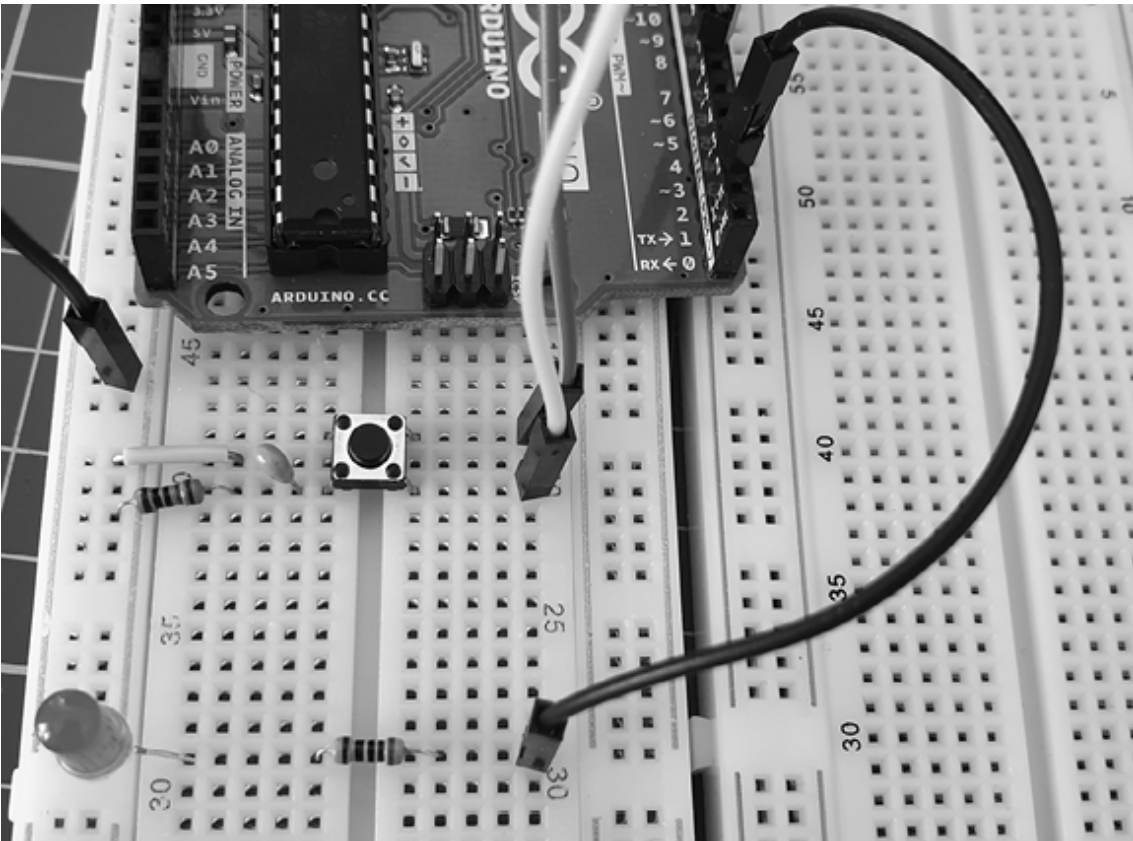


Figure 4-25: Connecting the LED branch to the Arduino

Before continuing, review the schematic for this circuit and check that your components are wired correctly. Compare the schematic against the actual wiring of the circuit.

The Sketch

For the sketch, enter and upload [Listing 4-1](#).

```
// Listing 4-1, Project 4 - Demonstrating a Digital Input
1 #define LED 3
  #define BUTTON 7

  void setup()
  {
2    pinMode(LED, OUTPUT); // output for the LED
    pinMode(BUTTON, INPUT); // input for the button
  }

  void loop()
```



```
{
  if ( digitalRead(BUTTON) == HIGH )
  {
    digitalWrite(LED, HIGH); // turn on the LED
    delay(500);              // wait for 0.5 seconds
    digitalWrite(LED, LOW);  // turn off the LED
  }
}
```

Listing 4-1: Digital input

After you've uploaded your sketch, tap the push button briefly. Your LED should stay on for half a second.

Understanding the Sketch

Let's examine the new items in the sketch for Project 4—specifically, `#define`, digital input pins, and the `if` statement.

Creating Constants with `#define`

Before `void setup()`, we use `#define` statements at 1 to create fixed values: when the sketch is compiled, the IDE replaces any instance of the defined word with the number that follows it. For example, when the IDE sees `LED` in the line at 2, it replaces it with the number 3. Notice that we do not use a semicolon after a `#define` value.

We're basically using the `#define` statements to label the digital pins for the LED and button in the sketch. It's a good idea to label pin numbers and other fixed values (such as a time delay) in this way, because if the value is used repeatedly in the sketch, then you won't have to edit the same item more than once. In this example, `LED` is used three times in the sketch, but to change this value we'd have to edit its definition only once in its `#define` statement.

Reading Digital Input Pins

To read the status of a button, we first define a digital I/O pin as an input in `void setup()` using the following:

```
pinMode(BUTTON, INPUT); // input for button
```

Next, to discover whether the button is connecting a voltage through to the digital input (that is, it's being pressed), we use `digitalRead(pin)`, where *pin* is the digital pin number to read. The function returns either `HIGH` (voltage is close to 5 V at the pin) or `LOW` (voltage is close to 0 V at the pin).

Making Decisions with if

Using `if`, we can make decisions in our sketch and tell the Arduino to run different code depending on the decision. For example, in the sketch for Project 4, we used [Listing 4-2](#).

```
// Listing 4-2
if (digitalRead(BUTTON) == HIGH)
{
    digitalWrite(LED, HIGH); // turn on the LED
    delay(500);              // wait for 0.5 seconds
    digitalWrite(LED, LOW);  // turn off the LED
}
```

Listing 4-2: A simple if-then example

The first line in this code snippet begins with `if` because it tests for a condition. If the condition is true (that is, if the voltage is `HIGH`), then it means that the button is pressed. The Arduino will then run the code that is inside the curly brackets.

To determine whether the button is pressed (`digitalRead(BUTTON)` is set to `HIGH`), we use a *comparison operator*, a double equal sign (`==`). If we were to replace `==` with `!=` (not equal to) in the sketch, then the LED would turn off when the button is pressed instead. Try it and see.

NOTE

A common mistake is to use a single equal sign (`=`), which means “make equal to,” in a test statement instead of a double equal sign (`==`), which says “test whether it is equal to.” You may not get an error message, but your `if` statement may not work!

Once you've had some success, try changing the length of time that the light stays on, or go back to Project 3 on page 38 in Chapter 3 and add a push button control. (Don't disassemble this circuit, though; we'll use it again in the next example.)

Modifying Your Sketch: Making More Decisions with if-else

You can add another action to an `if` statement by using `else`. For example, if we rewrite [Listing 4-1](#) by adding an `else` clause, as shown in [Listing 4-3](#), then the LED will turn on *if* the button is pressed, or *else* it will be off. Using `else` forces the Arduino to run another section of code if the test in the `if` statement is not true.

```
// Listing 4-3
#define LED 3
#define BUTTON 7

void setup()
{
  pinMode(LED, OUTPUT); // output for the LED
  pinMode(BUTTON, INPUT); // input for the button
}

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

Listing 4-3: Adding else

Boolean Variables

Sometimes you need to record whether something is in either of only two states, such as on or off, or hot or cold. A *Boolean variable* is the legendary

computer “bit” whose value can be only a zero (0, false) or one (1, true). As with any other variable, we need to declare it in order to use it:

```
boolean raining = true; // create the variable "raining" and
first make it true
```

Within the sketch, you can change the state of a Boolean with a simple reassignment, such as this:

```
raining = false;
```

Because Boolean variables can only take on the values of true or false, they are well suited to making decisions using `if`. True/false Boolean comparisons work well with the comparison operators `!=` and `==`. Here’s an example:

```
if ( raining == true )
{
    if ( summer != true )
    {
        // it is raining and not summer
    }
}
```

Comparison Operators

We can use various operators to make decisions about two or more Boolean variables or other states. These include the operators *not* (`!`), *and* (`&&`), and *or* (`||`).

The not Operator

The *not* operator is denoted by an exclamation mark (`!`). This operator is used as an abbreviation for checking whether something is *not true*. Here’s an example:

```
if ( !raining )
{
    // it is not raining (raining == false)
}
```

The and Operator

The logical *and* operator is denoted by `&&`. Using *and* helps reduce the number of separate if tests. Here's an example:

```
if (( raining == true ) && ( !summer ))
{
    // it is raining and not summer (raining == true and summer
    == false)
}
```

The or Operator

The logical *or* operator is denoted by `||`. Using *or* is pretty intuitive. Here's an example:

```
if (( raining == true ) || ( summer == true ))
{
    // it is either raining or summer
}
```

Making Two or More Comparisons

You can also make two or more comparisons using the same if statement. Here's an example:

```
if ( snow == true && rain == true && !hot )
{
    // it is snowing and raining and not hot
}
```

And you can use parentheses to set the order of operation. In the next example, the comparison in the parentheses is checked first and given a true or false state, and then that condition is subjected to the remaining test in the if statement:

```
if (( snow == true || rain == true ) && hot == false))
{
    // it is either snowing or raining, and not hot
}
```

Lastly, just like the examples of the not (!) operator before a value, simple true/false tests can be performed without requiring == true or == false in each test. The following code has the same effect as the preceding example:

```
if (( snow || rain ) && !hot )
{
    // it is either snowing or raining, and not hot
    // ( snow is true OR rain is true ) AND it is not hot
}
```

As you can see, it's possible to have the Arduino make a multitude of decisions using Boolean variables and comparison operators. Once you move on to more complex projects, this will become very useful.

Project #5: Controlling Traffic

Now let's put our newfound knowledge to use by solving a hypothetical problem. As the town planners for a rural county, we have a problem with a single-lane bridge that crosses the river. Every week, one or two accidents occur at night, when tired drivers rush across the bridge without first stopping to see if the road is clear. We have suggested that traffic lights be installed, but the mayor wants to see them demonstrated before signing off on the purchase. We could rent temporary lights, but they're expensive. Instead, we've decided to build a model of the bridge with working traffic lights using LEDs and an Arduino.

The Goal

Our goal is to install three-color traffic lights at each end of the single-lane bridge. The lights allow traffic to flow in only one direction at a time. When sensors located at either end of the bridge detect a car waiting at a red light, the lights will change and allow the traffic to flow in the opposite direction.

The Algorithm

We'll use two buttons to simulate the vehicle sensors at each end of the bridge. Each set of lights will have red, yellow, and green LEDs. Initially, the system will allow traffic to flow from west to east, so the west-facing lights will be set to green and the east-facing lights will be set to red.

When a vehicle approaches the bridge (modeled by pressing the button) and the light is red, the system will turn the light on the opposite side from green to yellow to red, and then wait a set period of time to allow any vehicles already on the bridge to finish crossing. Next, the yellow light on the waiting vehicle's side will blink as a “get ready” notice for the driver, and finally the light will change to green. The light will remain green until a vehicle approaches the other side, at which point the process repeats.

The Hardware

Here's what you'll need to create this project:

Two red LEDs (LED1 and LED2)

Two yellow LEDs (LED3 and LED4)

Two green LEDs (LED5 and LED6)

Six 560 Ω resistors (R1 to R6)

Two 10 k Ω resistors (R7 and R8)

Two 100 nF capacitors (C1 and C2)

Two push buttons (S1 and S2)

One medium-sized breadboard

Arduino and USB cable

Various connecting wires

The Schematic

Because we're controlling only six LEDs and receiving input from two buttons, the design will not be too difficult. [Figure 4-26](#) shows the schematic for our project.

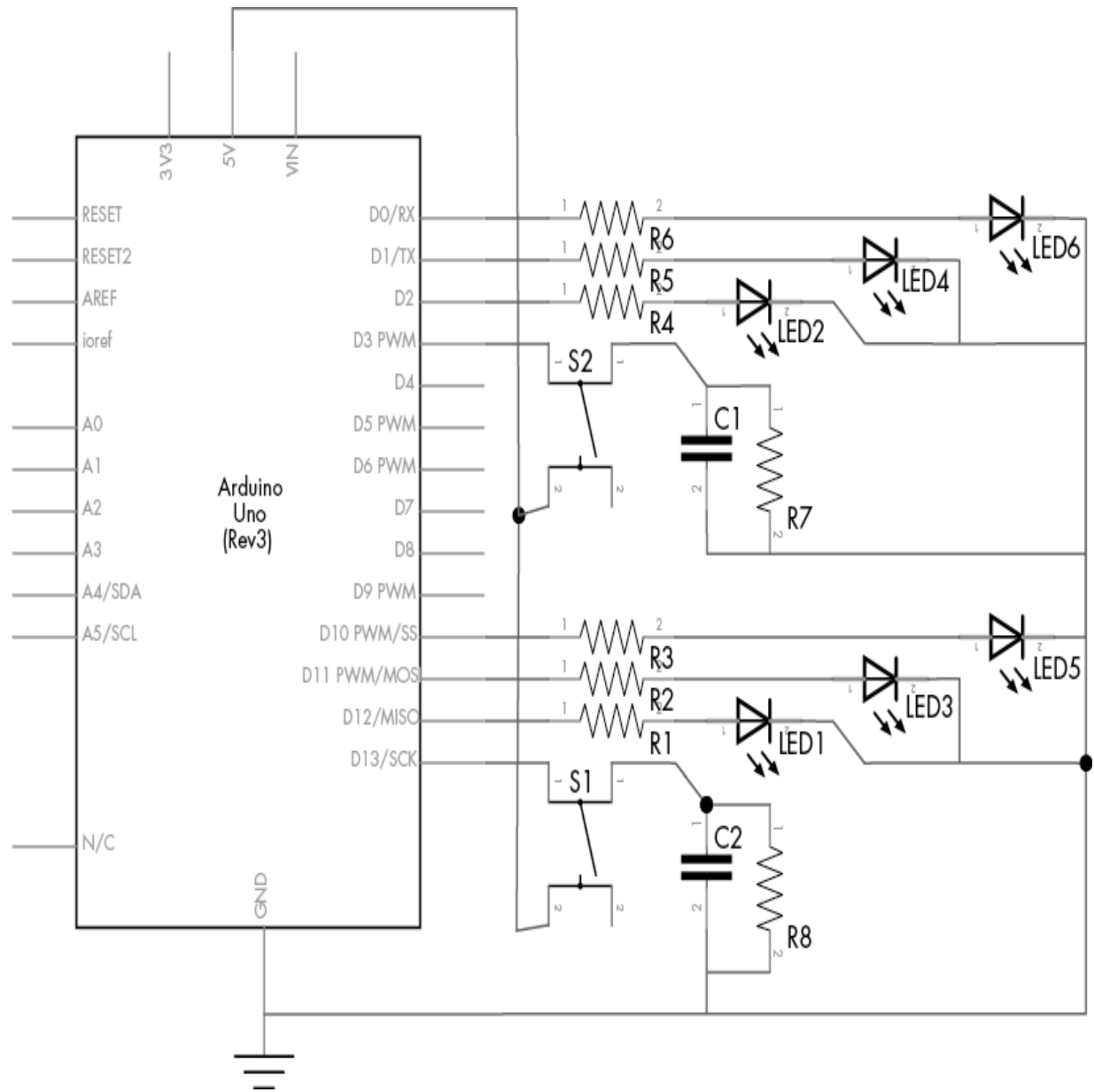


Figure 4-26: Schematic for Project 5

This circuit is basically a more elaborate version of the button and LED circuit in Project 4, with resistors, more LEDs, and another button.

Be sure that the LEDs are inserted in the correct direction: the resistors connect to LED anodes, and the LED cathodes connect to the Arduino GND pin, as shown in [Figure 4-27](#).

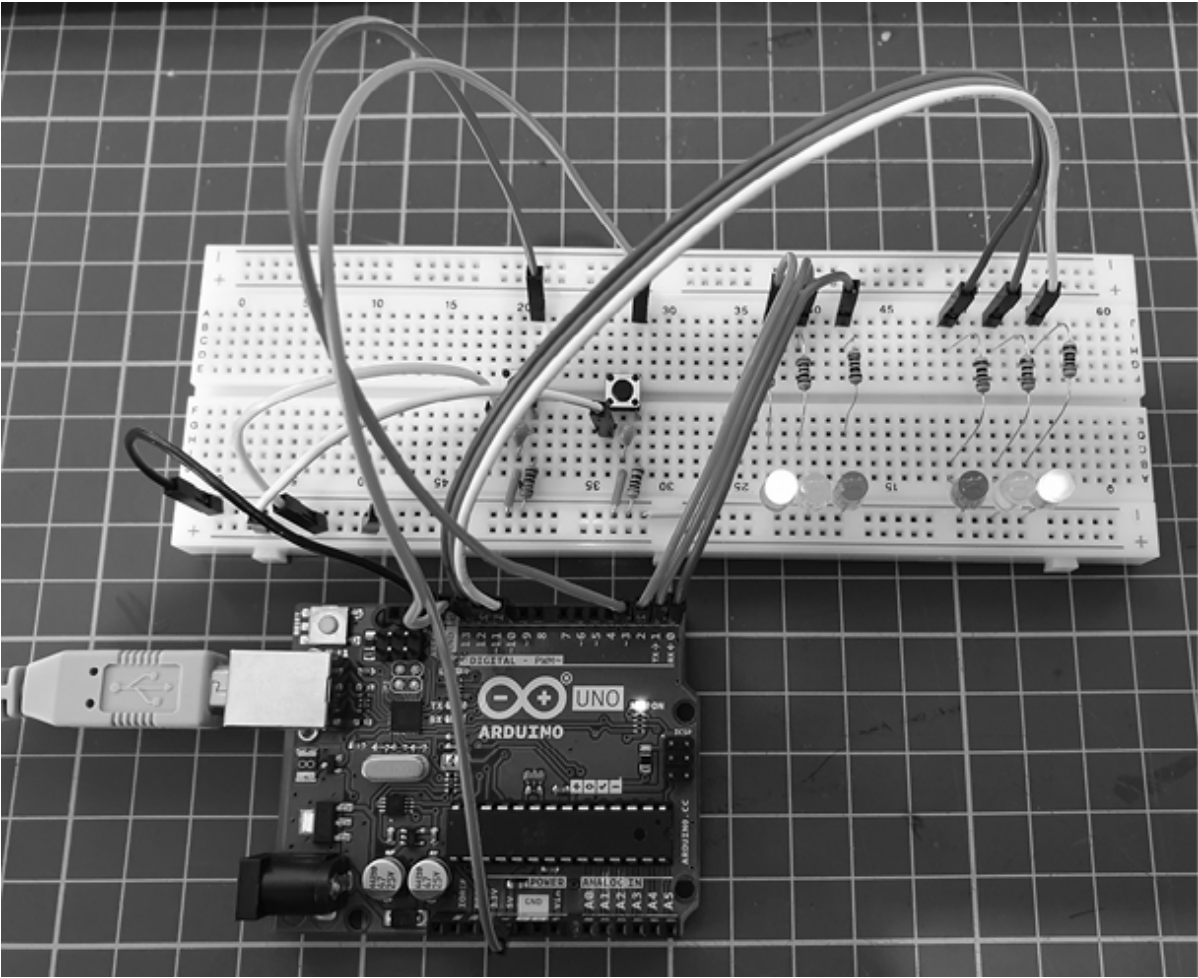


Figure 4-27: The completed circuit

The Sketch

And now for the sketch. Can you see how it matches our algorithm?

```
// Project 5 - Controlling Traffic

// define the pins that the buttons and lights are connected
// to:
1 #define westButton 3
  #define eastButton 13
  #define westRed 2
  #define westYellow 1
  #define westGreen 0
  #define eastRed 12
  #define eastYellow 11
  #define eastGreen 10
```

```

#define yellowBlinkTime 500 // 0.5 seconds for yellow light
blink

2 boolean trafficWest = true; // west = true, east = false
3 int flowTime = 10000;      // amount of time to let traffic
  flow
4 int changeDelay = 2000;    // amount of time between color
  changes

void setup()
{
  // set up the digital I/O pins
  pinMode(westButton, INPUT);
  pinMode(eastButton, INPUT);
  pinMode(westRed, OUTPUT);
  pinMode(westYellow, OUTPUT);
  pinMode(westGreen, OUTPUT);
  pinMode(eastRed, OUTPUT);
  pinMode(eastYellow, OUTPUT);
  pinMode(eastGreen, OUTPUT);
  // set initial state for lights - west side is green first
  digitalWrite(westRed, LOW);
  digitalWrite(westYellow, LOW);
  digitalWrite(westGreen, HIGH);
  digitalWrite(eastRed, HIGH);
  digitalWrite(eastYellow, LOW);
  digitalWrite(eastGreen, LOW);
}

void loop()
{
  if ( digitalRead(westButton) == HIGH ) // request west>east
  traffic flow
  {
    if ( trafficWest != true )
      // only continue if traffic flowing in the opposite
(east) direction
    {
      trafficWest = true; // change traffic flow flag to
west>east
      delay(flowTime);    // give time for traffic to flow
      digitalWrite(eastGreen, LOW); // change east-facing
lights from green
                                   // to yellow to red
      digitalWrite(eastYellow, HIGH);
      delay(changeDelay);
    }
  }
}

```

```

        digitalWrite(eastYellow, LOW);
        digitalWrite(eastRed, HIGH);
        delay(changeDelay);
        for ( int a = 0; a < 5; a++ ) // blink yellow light
        {
            digitalWrite(westYellow, LOW);
            delay(yellowBlinkTime);
            digitalWrite(westYellow, HIGH);
            delay(yellowBlinkTime);
        }
        digitalWrite(westYellow, LOW);
        digitalWrite(westRed, LOW); // change west-facing
lights from red
                                // to green
        digitalWrite(westGreen, HIGH);
    }
    if ( digitalRead(eastButton) == HIGH ) // request east>west
traffic flow
    {
        if ( trafficWest == true )
            // only continue if traffic flow is in the opposite (west)
direction
        {
            trafficWest = false; // change traffic flow flag to
east>west
            delay(flowTime);      // give time for traffic to flow
            digitalWrite(westGreen, LOW);
            // change west-facing lights from green to yellow to red
            digitalWrite(westYellow, HIGH);
            delay(changeDelay);
            digitalWrite(westYellow, LOW);
            digitalWrite(westRed, HIGH);
            delay(changeDelay);
            for ( int a = 0 ; a < 5 ; a++ ) // blink yellow light
            {
                digitalWrite(eastYellow, LOW);
                delay(yellowBlinkTime);
                digitalWrite(eastYellow, HIGH);
                delay(yellowBlinkTime);
            }
            digitalWrite(eastYellow, LOW);
            digitalWrite(eastRed, LOW); // change east-facing lights
from red
                                // to green
            digitalWrite(eastGreen, HIGH);
        }
    }

```

```
}  
}
```

Our sketch starts by using `#define` at 1 to associate digital pin numbers with labels for all the LEDs used, as well as the two buttons. We have red, yellow, and green LEDs and a button each for the west and east sides of the bridge. The Boolean variable `trafficWest` at 2 is used to keep track of which way the traffic is flowing—`true` is west to east, and `false` is east to west.

NOTE

Notice that `trafficWest` is a single Boolean variable with the traffic direction set as either `true` or `false`. Having a single variable like this instead of two (one for east and one for west) ensures that both directions cannot accidentally be true at the same time, which helps avoid a crash!

The integer variable `flowTime` at 3 is the minimum period of time that vehicles have to cross the bridge. When a vehicle pulls up at a red light, the system extends this period to give the opposing traffic time to cross the bridge. The integer variable `changeDelay` at 4 is the elapsed time between changes of color from green to yellow to red.

Before the sketch enters the `void loop()` section, it is set for traffic to flow from west to east in `void setup()`.

Running the Sketch

Once it's running, the sketch does nothing until one of the buttons is pressed. When the east button is pressed, the line:

```
if ( trafficWest == true )
```

ensures that the lights change only if the traffic is heading in the opposite direction. The rest of the `void loop()` section is composed of a simple sequence of waiting and then of turning on and off various LEDs to simulate the traffic lights' operation.

Analog vs. Digital Signals

In this section you'll learn the difference between digital and analog signals, and you'll learn how to measure analog signals with the analog input pins.

Until now, our sketches have been using digital electrical signals, with just two discrete levels. Specifically, we used `digitalWrite(pin, HIGH)` and `digitalWrite(pin, LOW)` to blink an LED and `digitalRead()` to measure whether a digital pin had a voltage applied to it (HIGH) or not (LOW). [Figure 4-28](#) is a visual representation of a digital signal that alternates between high and low.

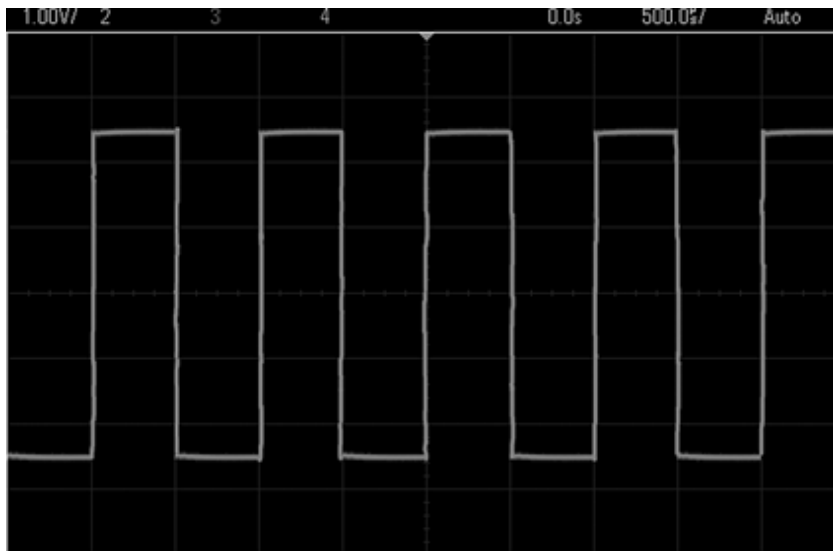


Figure 4-28: A digital signal, with highs appearing as horizontal lines at the top and lows appearing at the bottom

Unlike digital signals, analog signals can vary with an indefinite number of steps between high and low. For example, [Figure 4-29](#) shows the analog signal of a sine wave. Notice that as time progresses, the voltage floats fluidly between high and low levels.

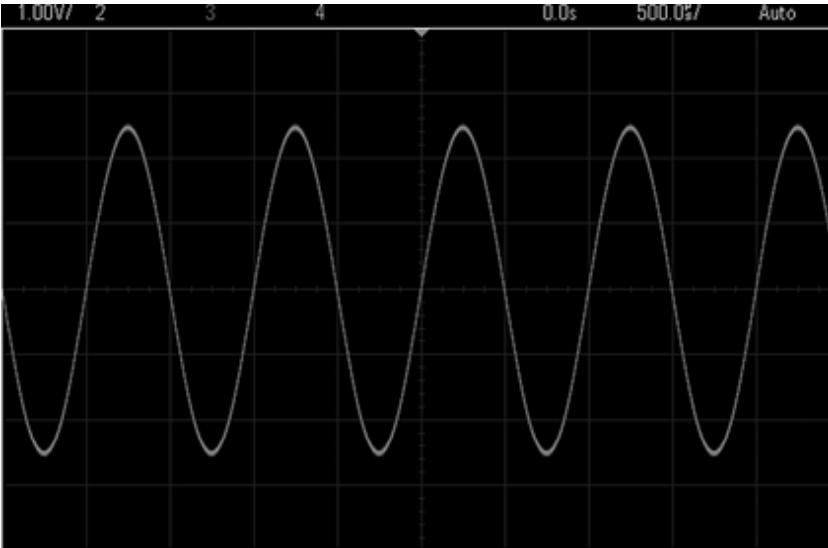


Figure 4-29: An analog signal of a sine wave



Figure 4-30: Analog inputs on the Arduino Uno

With our Arduino, high is closer to 5 V and low is closer to 0 V, or GND. We can measure the voltage values of an analog signal with our Arduino using the six analog inputs shown in [Figure 4-30](#). These analog inputs can safely measure voltages from 0 (GND) to no more than 5 V.

If you use the function `analogRead()`, then the Arduino will return a number between 0 and 1,023 in proportion to the voltage applied to the analog pin. For example, you might use `analogRead()` to store the value of analog pin 0 in the integer variable `a`, as shown here:

```
a = analogRead(0); // read analog input pin 0 (A0)
// returns 0 to 1023, which is usually 0.000 to 4.995 volts
```

Project #6: Creating a Single-Cell Battery Tester

Although the popularity and use of cell batteries has declined, most people still have a few devices around the house, such as remote controls, clocks, or children's toys, that use AA, AAA, C, or D cell batteries. These batteries carry much less than 5 V, so we can measure a cell's voltage with our Arduino to determine the state of the cell. In this project, we'll create a battery tester.

The Goal

Single-cell batteries such as AAs usually have a voltage of about 1.6 V when new, which decreases with use and age. We will measure the voltage and express the battery condition visually with LEDs. We'll use the reading from `analogRead()`, which we will convert to volts. The maximum voltage that can be read is 5 V, so we divide 5 by 1,024 (the number of possible values), which equals 0.0048. We multiply the value returned by `analogRead()` by this number to get the reading in volts. For example, if `analogRead()` returns 512, then we multiply that reading by 0.0048, which equals 2.4576 V.

The Algorithm

Here's the algorithm for our battery tester:

- . Read from analog pin 0.
- . Multiply the reading by 0.0048 to create a voltage value.
- . If the voltage is greater than or equal to 1.6 V, briefly turn on a green LED.
- . If the voltage is greater than 1.4 V *and* less than 1.6 V, briefly turn on a yellow LED.
- . If the voltage is less than 1.4 V, briefly turn on a red LED.
- . Repeat indefinitely.

The Hardware

Here's what you'll need to create this project:

Three 560 Ω resistors (R1 to R3)

One green LED (LED1)

One yellow LED (LED2)

One red LED (LED3)

One breadboard

Various connecting wires

Arduino and USB cable

The Schematic

The schematic for the single-cell battery tester circuit is shown in [Figure 4-31](#). On the left side, notice the two terminals, labeled + and –. Connect the *matching* sides of the single-cell battery to be tested at those points. Positive should connect to positive, and negative should connect to negative.

WARNING

Under no circumstances should you measure anything larger than 5 V, nor should you connect positive to negative, or vice versa. Doing these things would damage your Arduino board.

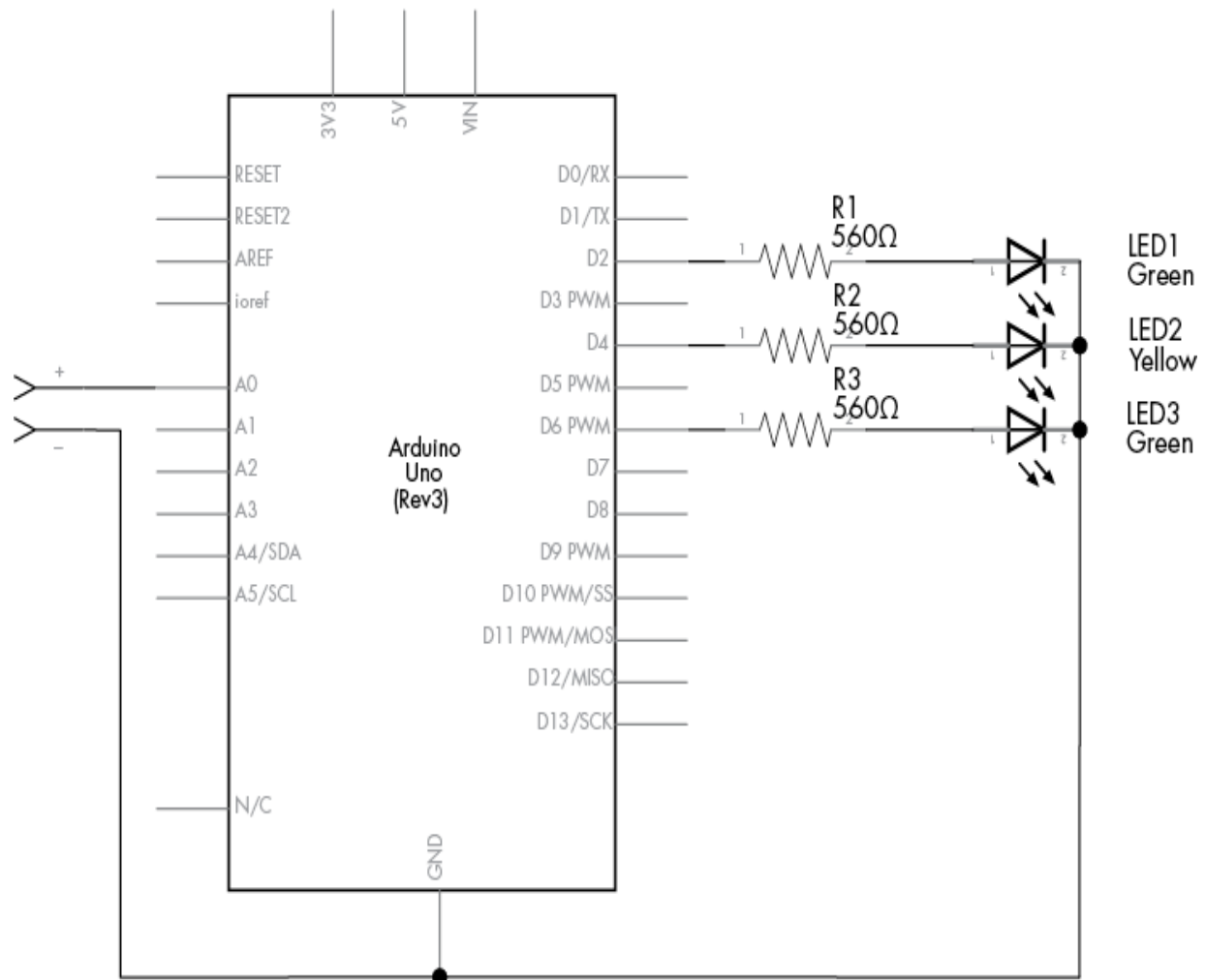


Figure 4-31: Schematic for Project 6

The Sketch

Now for the sketch. Since analog values can drift between integers, we're going to use a new type of variable called a *float*, which can contain fractional or decimal values:

```
// Project 6 - Creating a Single-Cell Battery Tester
#define newLED 2 // green LED
#define okLED 4 // yellow LED
#define oldLED 6 // red LED

int analogValue = 0;
1 float voltage = 0;
  int ledDelay = 2000;
```

```

void setup()
{
    pinMode(newLED, OUTPUT);
    pinMode(okLED, OUTPUT);
    pinMode(oldLED, OUTPUT);
}

void loop()
{
2   analogValue = analogRead(0);
3   voltage = 0.0048*analogValue;
4   if ( voltage >= 1.6 )
    {
        digitalWrite(newLED, HIGH);
        delay(ledDelay);
        digitalWrite(newLED, LOW);
    }
5   else if ( (voltage < 1.6) && (voltage) > 1.4 )
    {
        digitalWrite(okLED, HIGH);
        delay(ledDelay);
        digitalWrite(okLED, LOW);
    }
6   else if ( voltage <= 1.4 )
    {
        digitalWrite(oldLED, HIGH);
        delay(ledDelay);
        digitalWrite(oldLED, LOW);
    }
}

```

In this sketch, the Arduino takes the value measured by analog pin 0 at 2 and converts this to a voltage at 3. You'll learn more about the new type of variable, the `float` at 1, in the next section, which discusses doing arithmetic with an Arduino and using comparison operators to compare numbers.

Doing Arithmetic with an Arduino

Like a pocket calculator, the Arduino can perform calculations such as multiplication, division, addition, and subtraction. Here are some examples:

```
a = 100;  
b = a + 20;  
c = b - 200;  
d = c + 80; // d will equal 0
```

Float Variables

When you need to deal with numbers with a decimal point, you can use the variable type `float`. The values that can be stored in a `float` fall between 3.4028235×10^{38} and $-3.4028235 \times 10^{38}$ and are generally limited to six or seven decimal places of precision. You can mix integers and `float` numbers in your calculations. For example, you could add the `float` number `f` to the integer `a` and store the sum as the `float` variable `g`:

```
int a = 100;  
float f;  
float g;  
  
f = a / 3; // f = 33.333333  
g = a + f; // g = 133.333333
```

Comparison Operators for Calculations

We used comparison operators such as `==` and `!=` with `if` statements and digital input signals in Project 5. In addition to these operators, we can use the following to compare numbers or numerical variables:

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to

We used these operators to compare numbers in lines 4, 5, and 6 in the sketch for Project 6.

Improving Analog Measurement Precision with a Reference Voltage

As demonstrated in Project 6, the `analogRead()` function returns a value proportional to a voltage between 0 and 5 V. The upper value (5 V) is the *reference voltage*, the maximum voltage that the Arduino analog inputs will accept and return the highest value for (1,023).

To increase precision while reading even lower voltages, we can use a lower reference voltage. For example, when the reference voltage is 5 V, `analogRead()` represents this with a value from 0 to 1,023. However, if we needed to measure only a voltage with a maximum of 2 V, then we could alter the Arduino output to represent 2 V using the 0 to 1,023 range to allow for more precise measurement. You can do this with either an external or internal reference voltage, as discussed next.

Using an External Reference Voltage

The first method of using a reference voltage is with the *AREF* (analog reference) pin, as shown in [Figure 4-32](#).

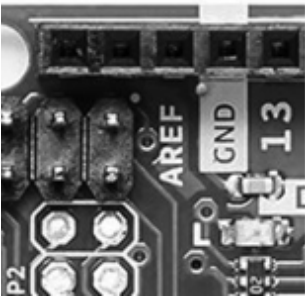


Figure 4-32: The Arduino Uno AREF pin

We can introduce a new reference voltage by connecting the voltage to the AREF pin and the matching GND to the Arduino's GND. Note that this can lower the reference voltage but will not raise it, because the reference voltage connected to an Arduino Uno must not exceed 5 V. A simple way to set a lower reference voltage is by creating a *voltage divider* with two resistors, as shown in [Figure 4-33](#).

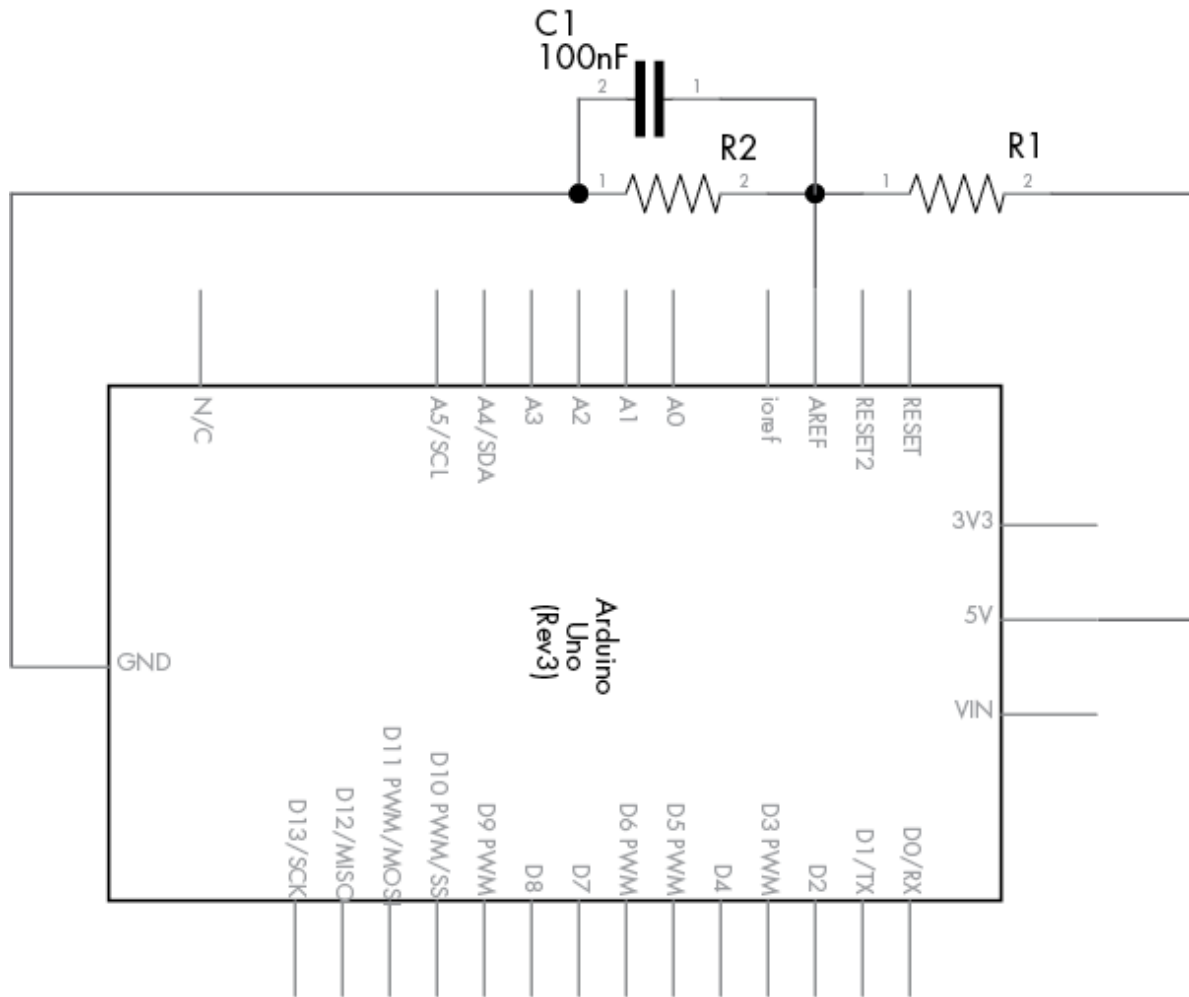


Figure 4-33: A voltage divider circuit

The values of $R1$ and $R2$ will determine the reference voltage according to the formula in [Figure 4-34](#).

$$V_{\text{out}} = V_{\text{in}} \left(\frac{R2}{R1 + R2} \right)$$

Figure 4-34: Reference voltage formula

In the formula, V_{out} is the reference voltage, and V_{in} is the input voltage—in this case, 5 V. $R1$ and $R2$ are the resistor values in ohms.

The simplest way to divide the voltage is to split V_{in} in half by setting $R1$ and $R2$ to the same value—for example, 10 k Ω each. When you're doing this, it's best to use the lowest-tolerance resistors you can find, such as 1 percent; confirm their true resistance values with a multimeter and use those

confirmed values in the calculation. Furthermore, it's a very good idea to place a 100 nF capacitor between AREF and GND to avoid a noisy AREF and prevent unstable analog readings.

When using an external reference voltage, insert the following line in the `void setup()` section of your sketch:

```
analogReference(EXTERNAL); // select AREF pin for reference voltage
```

Using the Internal Reference Voltage

The Arduino Uno also has an internal 1.1 V reference voltage. If this meets your needs, no hardware changes are required. Just add this line to `void setup()`:

```
analogReference(INTERNAL); // select internal 1.1 V reference voltage
```

The Variable Resistor

Variable resistors, also known as *potentiometers*, can generally be adjusted from 0 Ω up to their rated value. Their schematic symbol is shown in [Figure 4-35](#).



Figure 4-35: Variable resistor (potentiometer) symbol

Variable resistors have three pin connections: one in the center pin and one on each side. As the shaft of a variable resistor turns, it increases the resistance between one side and the center and decreases the resistance between the opposite side and the center.

Variable resistors can be either *linear* or *logarithmic*. The resistance of linear models changes at a constant rate as they turn, while the resistance of logarithmic models changes slowly at first and then increases rapidly. Logarithmic potentiometers are used more often in audio amplifier circuits,

because they model the human hearing response. You can generally identify whether a potentiometer is logarithmic or linear via the marking on the rear. Most will have either an *A* or a *B* next to the resistance value: *A* for logarithmic, *B* for linear. Most Arduino projects use linear variable resistors, such as the one shown in [Figure 4-36](#).



[Figure 4-36](#): A typical linear variable resistor

You can also get miniature versions of variable resistors, known as *trimpots* or *trimmers* (see [Figure 4-37](#)). Because of their size, trimpots are useful for making adjustments in circuits, but they're also very useful for breadboard work because they can be slotted in.

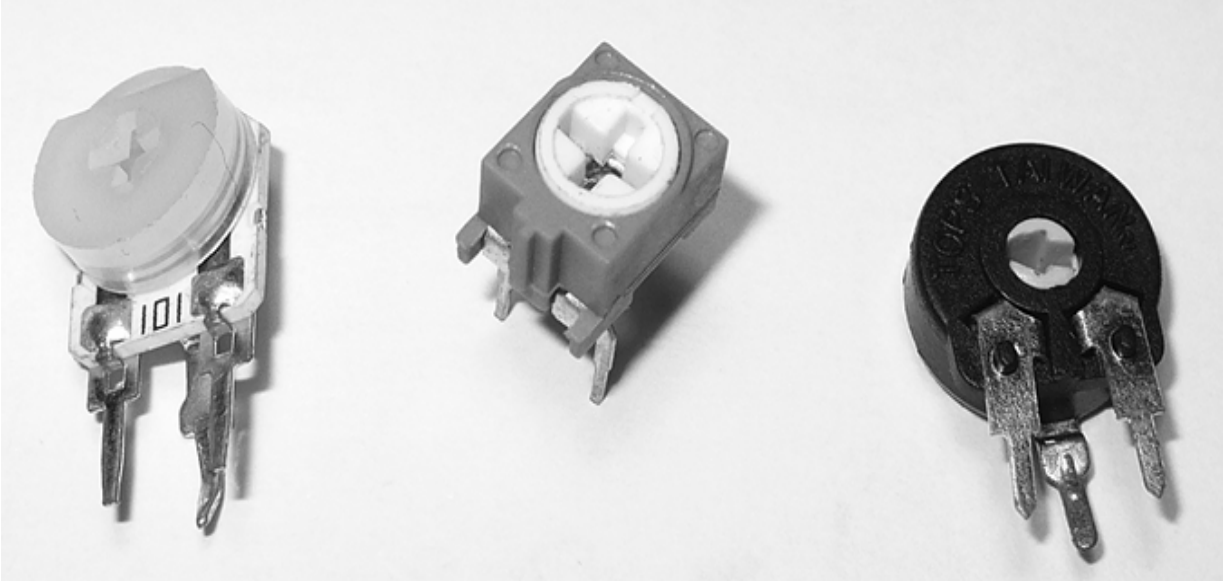


Figure 4-37: Various trimpots

NOTE

When shopping for trimpots, take note of the type. Often you will want one that is easy to adjust with a screwdriver that you have on hand. The enclosed types, pictured in [Figure 4-37](#), last longer than the cheaper, open contact types.

Piezoelectric Buzzers

A *piezoelectric element* (*piezo* for short), or buzzer, is a small, round device that can be used to generate loud and annoying noises that are perfect for alarms—or for having fun. [Figure 4-38](#) shows a common example, the TDK PS1240, next to an American quarter, to give you an idea of its size.

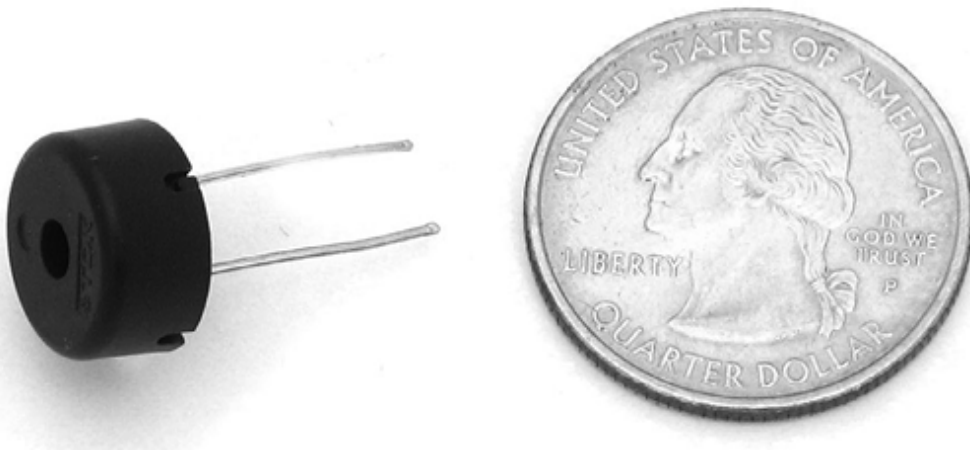


Figure 4-38: The TDK PS1240 piezo

Piezos contain a very thin plate inside the housing that moves when an electrical current is applied. When a pulsed current is applied (such as on . . . off . . . on . . . off), the plate vibrates and generates sound waves.

It's simple to use piezos with the Arduino because they can be turned on and off just like an LED. The piezo elements are not polarized and can be connected in either direction.

Piezo Schematic

The schematic symbol for the piezo looks like a loudspeaker ([Figure 4-39](#)), which makes it easy to recognize.

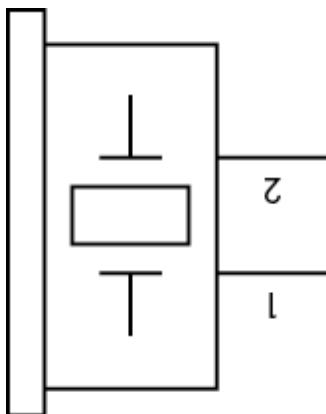


Figure 4-39: Piezo schematic symbol

NOTE

When shopping for a piezo for this project, be sure to get the piezo element only type. Some buzzer types look like [Figure 4-38](#) but include a tone-generating circuit built into the case; we don't want those because we're going to drive our tone directly from the Arduino.

Project #7: Trying Out a Piezo Buzzer

If you have a piezo handy and want to try it out, first connect it between Arduino GND and digital pins D3 to D0 inclusive. Then upload the following demonstration sketch to your Arduino:

```
// Project 7 - Trying Out a Piezo Buzzer
#define PIEZO 3 // pin 3 is capable of PWM output to drive
tones
int del = 500;
void setup()
{
    pinMode(PIEZO, OUTPUT);
}

void loop()
{
1   analogWrite(PIEZO, 128); // 50 percent duty cycle tone to
    the piezo
    delay(del);
    digitalWrite(PIEZO, LOW); // turn the piezo off
    delay(del);
}
```

This sketch uses pulse-width modulation on digital pin 3. If you change the duty cycle in the `analogWrite()` function (currently it's 128, which is 50 percent on) at 1, you can alter the sound of the buzzer.

To increase the volume of your piezo, increase the voltage applied to it. The voltage is currently limited to 5 V, but the buzzer would be much louder at 9 or 12 V. Because higher voltages can't be sourced from the Arduino, you would need to use an external power source for the buzzer, such as a 9 V

The part of the schematic labeled 12 V will be the positive side of the higher-power supply, whose negative side will connect to the Arduino GND pin.



Project #8: Creating a Quick-Read Thermometer

Temperature can be represented by an analog signal. We can measure temperature using the TMP36 voltage output temperature sensor made by Analog Devices (<http://www.analog.com/tmp36/>), shown in [Figure 4-41](#).



Figure 4-41: TMP36 temperature sensor

Notice that the TMP36 looks just like the BC548 transistor we worked with in the relay control circuit in Chapter 3. The TMP36 outputs a voltage that is proportional to the temperature, so you can determine the current temperature using a simple conversion. For example, at 25 degrees Celsius, the output voltage is 750 mV, and each change in temperature of 1 degree results in a change of 10 mV. The TMP36 can measure temperatures between -40 and 125 degrees Celsius.

The function `analogRead()` will return a value between 0 and 1,023, which corresponds to a voltage between 0 and just under 5,000 mV (5 V). If we multiply the output of `analogRead()` by (5,000/1,024), we will get the actual voltage returned by the sensor. Next, we subtract 500 (an offset used by the TMP36 to allow for temperatures below 0) and then divide by 10, which leaves us with the temperature in degrees Celsius. If you work in Fahrenheit, then multiply the Celsius value by 1.8 and add 32 to the result.

The Goal

In this project, we'll use the TMP36 to create a quick-read thermometer. When the temperature falls below 20 degrees Celsius, a blue LED turns on; when the temperature is between 20 and 26 degrees, a green LED turns on; and when the temperature is above 26 degrees, a red LED turns on.

The Hardware

Here's what you'll need to create this project:

Three 560 Ω resistors (R1 to R3)

One red LED (LED1)

One green LED (LED2)

One blue LED (LED3)

One TMP36 temperature sensor

One breadboard

Various connecting wires

Arduino and USB cable

The Schematic

The circuit is simple. When you're looking at the labeled side of the TMP36, the pin on the left connects to the 5 V input, the center pin is the voltage output, and the pin on the right connects to GND, as shown in [*Figure 4-42*](#).

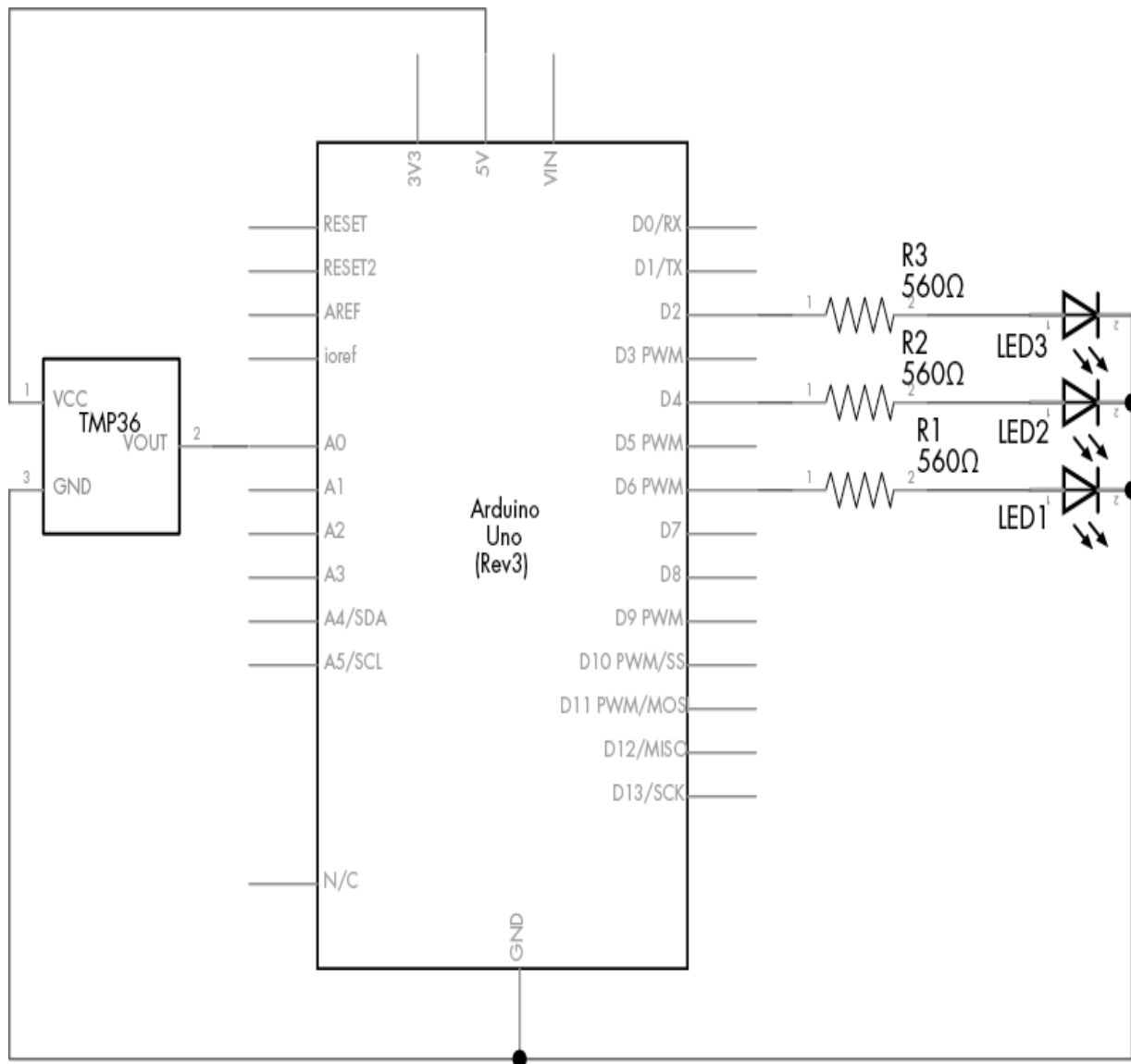


Figure 4-42: Schematic for Project 8

The Sketch

And now for the sketch:

```
// Project 8 - Creating a Quick-Read Thermometer

// define the pins that the LEDs are connected to:
#define HOT 6
#define NORMAL 4
#define COLD 2

float voltage = 0;
```

```

float celsius = 0;
float hotTemp = 26;
float coldTemp = 20;
float sensor = 0;

void setup()
{
    pinMode(HOT, OUTPUT);
    pinMode(NORMAL, OUTPUT);
    pinMode(COLD, OUTPUT);
}

void loop()
{
    // read the temperature sensor and convert the result to
    degrees Celsius
1    sensor = analogRead(0);
    voltage = ( sensor * 5000 ) / 1024; // convert raw sensor
    value to
                                     // millivolts
    voltage = voltage - 500; // remove voltage offset
    celsius = voltage / 10; // convert millivolts to Celsius
    // act on temperature range
2    if ( celsius < coldTemp )
    {
        digitalWrite(COLD, HIGH);
        delay(1000);
        digitalWrite(COLD, LOW);
    }
3    else if ( celsius > coldTemp && celsius <= hotTemp )
    {
        digitalWrite(NORMAL, HIGH);
        delay(1000);
        digitalWrite(NORMAL, LOW);
    }
    else
    {
        // celsius is > hotTemp
        digitalWrite(HOT, HIGH);
        delay(1000);
        digitalWrite(HOT, LOW);
    }
}

```

The sketch first reads the voltage from the TMP36 and converts it to a temperature in degrees Celsius at 1. Next, using the if-else statements at 2

and 3, the code compares the current temperature against the values for hot and cold and turns on the appropriate LED. The `delay(1000)` statements are used to prevent the lights from flashing on and off too quickly if the temperature fluctuates rapidly between two ranges.

You can experiment with the thermometer by blowing cool air over it to lower the temperature or by rubbing two fingers over the TMP36's body to generate heat.

Looking Ahead

And Chapter 4 comes to a close. You now have a lot more tools to work with, including digital inputs and outputs, new types of variables, and various mathematical functions. In the next chapter, you will have a lot more fun with LEDs, learn to create your own functions, build a computer game and electronic dice, and much more.

5

WORKING WITH FUNCTIONS

In this chapter you will

Create your own functions

Learn to make decisions with `while` and `do-while`

Send and receive data between your Arduino and the Serial Monitor window

Learn about `long` variables

You can make your Arduino sketches easier to read and simpler to design by creating your own functions. You can also create modular code that you can reuse in subsequent projects. In addition to these topics, this chapter will introduce a way to have the Arduino make decisions that control blocks of code, and you'll learn about a type of integer variable called the `long`. You'll also use your own functions to create a new type of thermometer.

A *function* consists of a set of instructions, packaged as a unit and given a name, that we can use anywhere in our sketches. Although many functions are already available in the Arduino language, sometimes you won't find one to suit your specific needs—or you may need to run part of a sketch repeatedly to make a function work, which is a waste of memory. In both of these situations, you might wish you had a better function to do what you need to do. The good news is that there is such a function—the one you create yourself.

Project #9: Creating a Function to Repeat an Action

You can write simple functions to repeat actions on demand. For example, the following function will turn the built-in LED on (at 1 and 3) and off (at 2 and 4) twice:

```
void blinkLED()
{
1    digitalWrite(13, HIGH);
    delay(1000);
2    digitalWrite(13, LOW);
    delay(1000);
3    digitalWrite(13, HIGH);
    delay(1000);
4    digitalWrite(13, LOW);
    delay(1000);
}
```

Here is the function being used within a complete sketch, which you can upload to the Arduino:

```
// Project 9 - Creating a Function to Repeat an Action

#define LED 13
#define del 200

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

void blinkLED()
{
    digitalWrite(LED, HIGH);
    delay(del);
    digitalWrite(LED, LOW);
    delay(del);
    digitalWrite(LED, HIGH);
    delay(del);
    digitalWrite(LED, LOW);
    delay(del);
}
```

```
    }  
  
    void loop()  
    {  
1    blinkLED();  
      delay(1000);  
    }
```

When the `blinkLED()` function is called in `void loop()` at 1, the Arduino will run the commands within the `void blinkLED()` section. In other words, you have created your own function and used it when necessary.

Project #10: Creating a Function to Set the Number of Blinks

The function we just created is pretty limited. What if we want to set the number of blinks and the delay? No problem—we can create a function that lets us change values, like this:

```
void blinkLED(int cycles, int del)  
{  
    for ( int z = 0 ; z < cycles ; z++ )  
    {  
        digitalWrite(LED, HIGH);  
        delay(del);  
        digitalWrite(LED, LOW);  
        delay(del);  
    }  
}
```

Our new `void blinkLED()` function accepts two integer values: `cycles` (the number of times we want to blink the LED) and `del` (the delay time between turning the LED on and off). So if we wanted to blink the LED 12 times with a 100-millisecond delay, then we would use `blinkLED(12, 100)`. Enter the following sketch into the IDE to experiment with this function:

```
// Project 10 - Creating a Function to Set the Number of  
Blinks
```

```
#define LED 13

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

void blinkLED(int cycles, int del)
{
  for ( int z = 0 ; z < cycles ; z++ )
  {
    digitalWrite(LED, HIGH);
    delay(del);
    digitalWrite(LED, LOW);
    delay(del);
  }
}

void loop()
{
1  blinkLED(12, 100);
   delay(1000);
}
```

You can see at 1 that the values 12 and 100—for the number of blinks and the delay, respectively—are passed into our custom function `blinkLED()`. Therefore, the LED will blink 12 times with a delay of 100 milliseconds between blinks. The display then pauses for 1,000 milliseconds, or 1 second, before the `loop()` function starts all over again.

Creating a Function to Return a Value

In addition to creating functions that accept values entered as parameters (as `void blinkLED()` did in Project 10), you can create functions that return a value, in the same way that `analogRead()` returns a value between 0 and 1,023 when measuring an analog input, as demonstrated in Project 8 (see page 91 in Chapter 4).

Up until now, all the functions we've seen have started with the word `void`. This tells the Arduino that the function returns nothing, just an empty void. But we can create functions that return any type of value we want. For example, if we wanted a function to return an integer value, we would

create it using `int`. If we wanted it to return a floating point value, it would begin with `float`. Let's create some useful functions that return actual values.

Consider this function that converts degrees Celsius to Fahrenheit:

```
float convertTemp(float celsius)
{
    float fahrenheit = 0;
    fahrenheit = (1.8 * celsius) + 32;
    return fahrenheit;
}
```

In the first line, we define the function name (`convertTemp`), its return value type (`float`), and any variables that we might want to pass into the function (`float celsius`). To use this function, we send it an existing value. For example, if we wanted to convert 40 degrees Celsius to Fahrenheit and store the result in a `float` variable called `tempf`, we would call `convertTemp()` like so:

```
float tempf = convertTemp(40);
```

This would place 40 into the `convertTemp()` variable `celsius` and use it in the calculation `fahrenheit() = (1.8 * celsius) + 32` in the `convertTemp()` function. The result is then returned into the variable `tempf` with the `convertTemp()` line `return fahrenheit;`.

Project #11: Creating a Quick-Read Thermometer That Blinks the Temperature

Now that you know how to create custom functions, we'll make a quick-read thermometer using the TMP36 temperature sensor from Chapter 4 and the Arduino's built-in LED. If the temperature is below 20 degrees Celsius, the LED will blink twice and then pause; if the temperature falls between 20 and 26 degrees, the LED will blink four times and then pause; and if the temperature is above 26 degrees, the LED will blink six times.

We'll make our sketch more modular by breaking it up into distinct functions that will be reusable, as well as making the sketch easier to follow. Our thermometer will perform two main tasks: measure and categorize the temperature, and blink the LED a certain number of times (as determined by the temperature).

The Hardware

The required hardware is minimal:

One TMP36 temperature sensor

One breadboard

Various connecting wires

Arduino and USB cable

The Schematic

The circuit is very simple, as shown in [*Figure 5-1*](#).

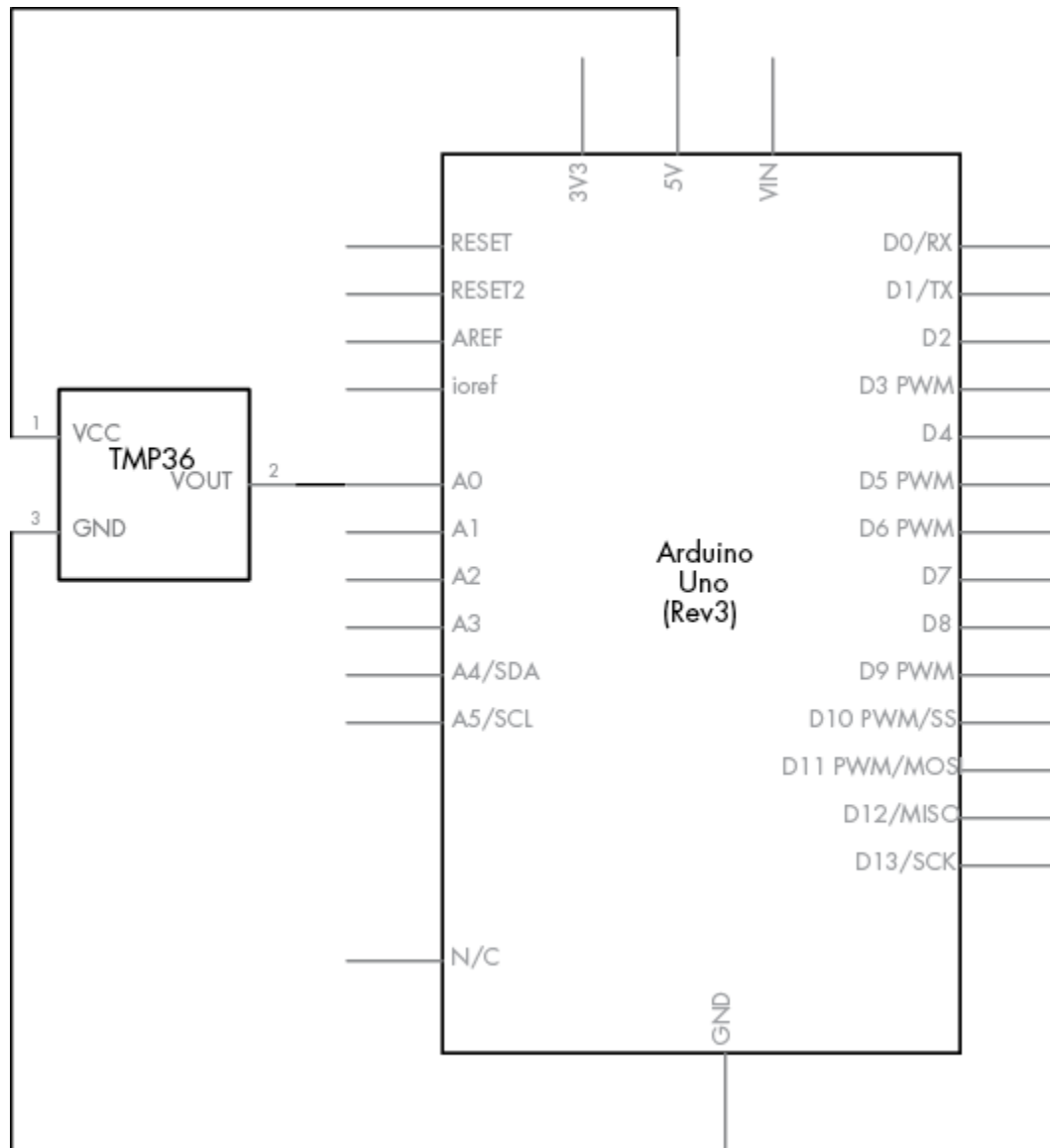


Figure 5-1: Schematic for Project 11

The Sketch

We'll need to create two functions for the sketch. The first one will read the value from the TMP36, convert it to Celsius, and then return a value of 2, 4, or 6, corresponding to the number of times the LED should blink. We'll start with the sketch from Project 8 for this purpose and make minor adjustments.

For our second function, we'll use `blinkLED()` from Project 9. Our void loop will call the functions in order and then pause for 2 seconds before

restarting.

NOTE

Remember to save your modified project sketches with new filenames so that you don't accidentally delete your existing work!

Enter this code into the IDE:

```
// Project 11 - Creating a Quick-Read Thermometer That Blinks
the Temperature

#define LED 13

int blinks = 0;

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

int checkTemp()
{
  float voltage = 0;
  float celsius = 0;
  float hotTemp = 26;
  float coldTemp = 20;
  float sensor = 0;
  int result;
  // read the temperature sensor and convert the result to
  degrees Celsius

  sensor = analogRead(0);
  voltage = (sensor * 5000) / 1024; // convert raw sensor
value to millivolts
  voltage = voltage - 500;           // remove voltage offset
  celsius = voltage / 10;           // convert millivolts to
Celsius

  // act on temperature range
  if (celsius < coldTemp)
  {
    result = 2;
  }
}
```



```

        else if (celsius >= coldTemp && celsius <= hotTemp)
        {
            result = 4;
        }
        else
        {
            result = 6;    // (celsius > hotTemp)
        }
        return result;
    }

    void blinkLED(int cycles, int del)
    {
        for ( int z = 0 ; z < cycles ; z++ )
        {
            digitalWrite(LED, HIGH);
            delay(del);
            digitalWrite(LED, LOW);
            delay(del);
        }
    }

1 void loop()
    {
        blinks = checkTemp();
        blinkLED(blinks, 500);
        delay(2000);
    }

```

Because we use custom functions, all we have to do in `void_loop()` at 1 is call them and set the delay. The function `checkTemp()` returns a value to the integer variable `blinks`, and then `blinkLED()` will blink the LED `blinks` times with a delay of 500 milliseconds. The sketch then pauses for 2 seconds before repeating.

Upload the sketch and watch the LED to see this thermometer in action. As before, see if you can change the temperature of the sensor by blowing on it or holding it between your fingers. Be sure to keep this circuit assembled, since we'll use it in the projects that follow.

Displaying Data from the Arduino in the Serial Monitor

So far, we have sent sketches to the Arduino and used the LEDs to show us output (such as temperatures and traffic signals). Blinking LEDs make it easy to get feedback from the Arduino, but blinking lights can tell us only so much. In this section, you'll learn how to use the Arduino's cable connection and the IDE's Serial Monitor window to display data from the Arduino and send data to the Arduino from your computer's keyboard.

The Serial Monitor

To open the Serial Monitor, start the IDE and click the Serial Monitor icon on the toolbar, shown in [Figure 5-2](#). It appears as a new tab in the IDE with the output window, and should look similar to [Figure 5-3](#).

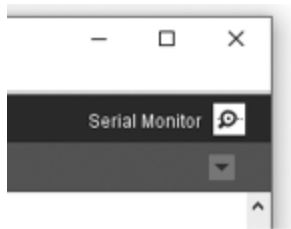


Figure 5-2: The Serial Monitor icon on the IDE toolbar



Figure 5-3: Serial Monitor

As you can see in [Figure 5-3](#), the Serial Monitor displays an input field at the top, consisting of a single row and a Send button, and an output window below it, where data from the Arduino is displayed. When the Autoscroll box is checked (the chevron button next to the clock icon), the most recent

output is displayed, and once the screen is full, older data rolls off the screen as newer output is received. If you uncheck Autoscroll, you can manually examine the data using a vertical scroll bar.

Starting the Serial Monitor

Before we can use the Serial Monitor, we need to activate it by adding this function to our sketch in `void setup()`:

```
Serial.begin(9600);
```

The value 9600 is the speed at which the data will travel between the computer and the Arduino, also known as *baud*. This value must match the speed setting at the bottom right of the Serial Monitor, as shown in [Figure 5-3](#).

Sending Text to the Serial Monitor

To send text from the Arduino to the Serial Monitor to be displayed in the output window, you can use `Serial.print()`:

```
Serial.print("Arduino for Everyone!");
```

This sends the text between the quotation marks to the Serial Monitor's output window.

You can also use `Serial.println()` to display text and then force any following text to start on the next line:

```
Serial.println("Arduino for Everyone!");
```

Displaying the Contents of Variables

You can also display the contents of variables in the Serial Monitor. For example, this would display the contents of the variable `results`:

```
Serial.println(results);
```

If the variable is a `float`, the display will default to two decimal places. You can specify the number of decimal places as a number between 0 and 6 by entering a second parameter after the variable name. For example, to display the `float` variable results to four decimal places, you would enter the following:

```
Serial.print(results,4);
```

Project #12: Displaying the Temperature in the Serial Monitor

Using the hardware from Project 8, we'll display temperature data in Celsius and Fahrenheit in the Serial Monitor window. To do this, we'll create one function to determine the temperature values and another to display them in the Serial Monitor.

Enter this code into the IDE:

```
// Project 12 - Displaying the Temperature in the Serial
Monitor

float celsius    = 0;
float fahrenheit = 0;

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

1 void findTemps()
{
  float voltage = 0;
  float sensor  = 0;

  // read the temperature sensor and convert the result to
  // degrees C and F
  sensor = analogRead(0);
  voltage = (sensor * 5000) / 1024; // convert the raw
  // sensor value to
  // millivolts
  voltage = voltage - 500;          // remove the voltage
```

```

offset
  celsius = voltage / 10;           // convert millivolts to
Celsius
  fahrenheit = (1.8 * celsius) + 32; // convert Celsius to
Fahrenheit
}

2 void displayTemps()
{
  Serial.print("Temperature is ");
  Serial.print(celsius, 2);
  Serial.print(" deg. C / ");
  Serial.print(fahrenheit, 2);
  Serial.println(" deg. F");
  // use .println here so the next reading starts on a new
line
}

void loop()
{
  findTemps();
  displayTemps();
  delay(1000);
}

```

A lot is happening in this sketch, but we've created two functions, `findTemps()` at 1 and `displayTemps()` at 2, to simplify things. These functions are called in `void loop()`, which is quite simple. Thus, you see one reason to create your own functions: to make your sketches easier to understand and the code more modular and possibly reusable.

After uploading the sketch, wait a few seconds and then display the Serial Monitor. The temperature in your area should be displayed in a similar manner to that shown in [Figure 5-4](#).

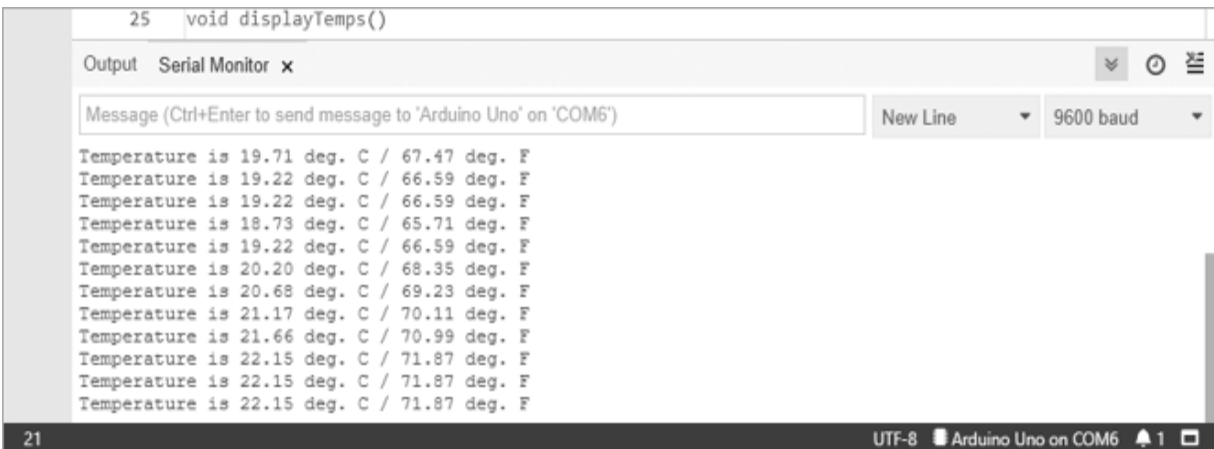


Figure 5-4: Result from Project 12

Debugging with the Serial Monitor

You can use the Serial Monitor to help *debug* (locate and fix errors in) your sketches. For example, if you insert `Serial.println()` statements in your sketch containing brief notes about their location in the sketch, then you can see when the Arduino passes each statement. For example, you might use the line:

```
Serial.println("now in findTemps()");
```

inside the function `findTemps()` to let you know when the Arduino is running that function.

Making Decisions with while Statements

You can use `while` statements in a sketch to repeat instructions, as long as (*while*) a given condition is true.

while

The condition is always tested *before* the code in the `while` statement is executed. For example, `while (temperature > 30)` will test whether the value of `temperature` is greater than 30. You can use any comparison operator, or Boolean variable, within the parentheses to create the condition.

In the following sketch, the Arduino will count up to 10 seconds and then continue with its program:

```
int a = 0; // an integer
while ( a < 10 )
{
    a = a + 1;
    delay(1000);
}
```

This sketch starts by setting the variable `a` to 0. It then checks that the value of `a` is less than 10 (`while (a < 10)`), and, if so, adds 1 to that value, waits 1 second (`delay(1000)`), and checks the value again. It repeats the process until `a` has a value of 10. Once `a` is equal to 10, the comparison in the `while` statement is false; therefore, the Arduino will continue on with the sketch after the `while` loop brackets.

do-while

In contrast to `while`, the `do-while` structure places the test *after* execution of the code within the `do-while` statement. Here's an example:

```
int a = 0; // an integer
do
{
    delay(1000);
    a = a + 1;
} while ( a < 100 );
```

In this case, the code between the curly brackets will execute *before* the conditions of the test (`while (a < 100)`) have been checked. As a result, even if the conditions are not met, the loop will run once. You'll decide whether to use a `while` or a `do-while` statement when designing your particular project.

Sending Data from the Serial Monitor to the Arduino

To send data from the Serial Monitor to the Arduino, we need the Arduino to listen to the *serial buffer*—the part of the Arduino that receives data from the outside world via the serial pins (digital 0 and 1) that are also connected to the USB interface to your computer. The serial buffer holds incoming data from the Serial Monitor's input window.

Project #13: Multiplying a Number by Two

To demonstrate the process of sending and receiving data via the Serial Monitor, let's dissect the following sketch. This sketch accepts a single digit from the user, multiplies it by 2, and then displays the result in the Serial Monitor's output window. After you have uploaded the sketch, when you open the Serial Monitor window, select **No Line Ending** in the window's drop-down menu. When entering data in the Serial Monitor, you need to press CTRL-ENTER to send the data to the Arduino (not just ENTER).

```
// Project 13 - Multiplying a Number by Two

int number;

void setup()
{
  Serial.begin(9600);
}
void loop()
{
  number = 0;      // set the variable to zero, ready for a
  new read
  Serial.flush(); // clear any "junk" out of the serial
  buffer before waiting
1  while (Serial.available() == 0)
    {
      // do nothing until something enters the serial buffer
    }
2  while (Serial.available() > 0)
    {
      number = Serial.read() - '0';
      // read the number in the serial buffer and
      // remove the ASCII text offset for zero: '0'
    }
    // Show me the number!
```



```
Serial.print("You entered: ");  
Serial.println(number);  
Serial.print(number);  
Serial.print(" multiplied by two is ");  
number = number * 2;  
Serial.println(number);  
}
```

The `Serial.available()` test in the first while statement at 1 returns 0 if the user has not yet entered anything into the Serial Monitor. In other words, it tells the Arduino, “Do nothing until the user enters something.” The next while statement at 2 detects the number in the serial buffer and converts the text code into an integer. Afterward, the Arduino displays the number from the serial buffer and the multiplication results.

The `Serial.flush()` function at the start of the sketch clears the serial buffer just in case any unexpected data is in it, readying it to receive the next available data. [Figure 5-5](#) shows the Serial Monitor window after the sketch has run.

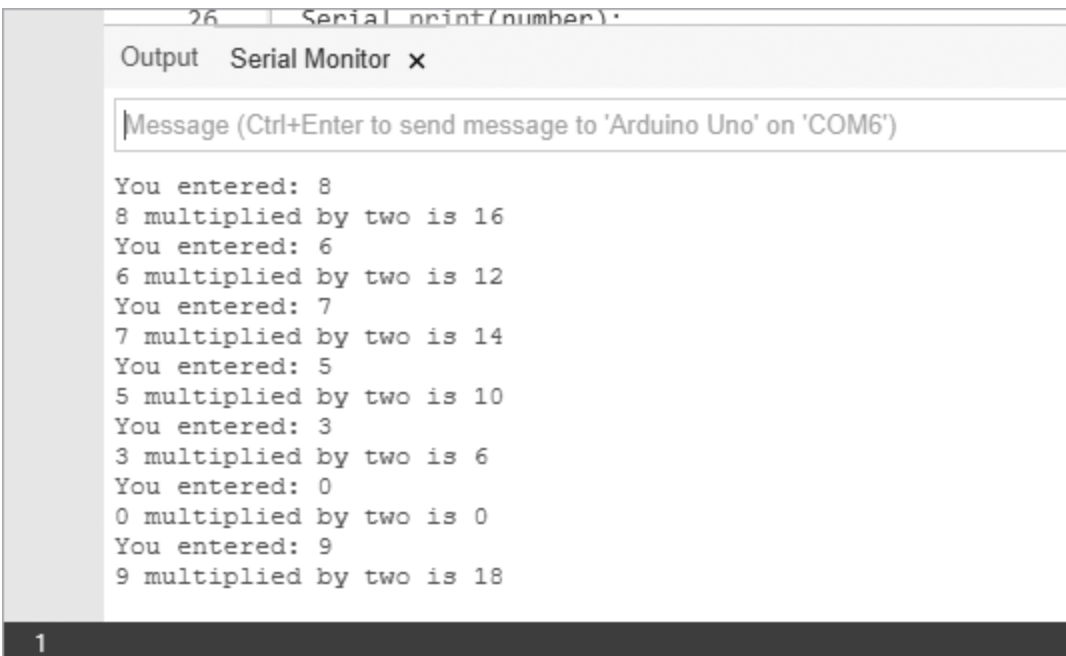


Figure 5-5: Sample input and output for Project 13

Although you can now enter numerical data into the Serial Monitor for the Arduino to process, it currently only accepts inputs of one digit. Even without this restriction, using integer variables limits the range of numbers

available. We can use long variables to increase this range, as discussed next.

long Variables

To use the Serial Monitor to accept numbers with more than one digit, we need to add some new code to our sketch, as you'll see shortly. When working with larger numbers, however, the `int` variable type can be limiting because it has a maximum value of 32,767. Fortunately, we can extend this limitation by using the `long` variable type. A `long` variable is a whole number between -2,147,483,648 and 2,147,483,647, a much larger range than that of an `int` variable (-32,768 to 32,767).

Project #14: Using long Variables

We'll use the Serial Monitor to accept long variables and numbers larger than one digit. This sketch accepts a number of many digits, multiplies that number by 2, and then returns the result to the Serial Monitor:

```
// Project 14 - Using long Variables

long number = 0;
long a = 0;

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

void loop()
{
  number = 0;      // zero the incoming number ready for a new
read
  Serial.flush(); // clear any "junk" out of the serial
buffer before waiting
  while (Serial.available() == 0)
  {
    // do nothing until something comes into the serial
buffer-
    // when something does come in, Serial.available will
```

```

return how many
    // characters are waiting in the buffer to process
}
// one character of serial data is available, begin
calculating
while (Serial.available() > 0)
{
    // move any previous digit to the next column on the
left;
    // in other words, 1 becomes 10 while there is data in
the buffer
    number = number * 10;
    // read the next number in the buffer and subtract the
character 0
    // from it to convert it to the actual integer number
    a = Serial.read() - '0';
    // add this value a into the accumulating number
    number = number + a;
    // allow a short delay for more serial data to come into
Serial.available
    delay(5);
}
Serial.print("You entered: ");
Serial.println(number);
Serial.print(number);
Serial.print(" multiplied by two is ");
number = number * 2;
Serial.println(number);
}

```

In this example, two while loops allow the Arduino to accept multiple digits from the Serial Monitor. When the first digit is entered (the leftmost digit of the number entered), it is converted to a number and then added to the total variable number. If that's the only digit, the sketch moves on. If another digit is entered (for example, the 2 in 42), the total is multiplied by 10 to shift the first digit to the left, and then the new digit is added to the total. This cycle repeats until the rightmost digit has been added to the total. Don't forget to select **No Line Ending** in the Serial Monitor window.

[Figure 5-6](#) shows the input and output of this sketch.

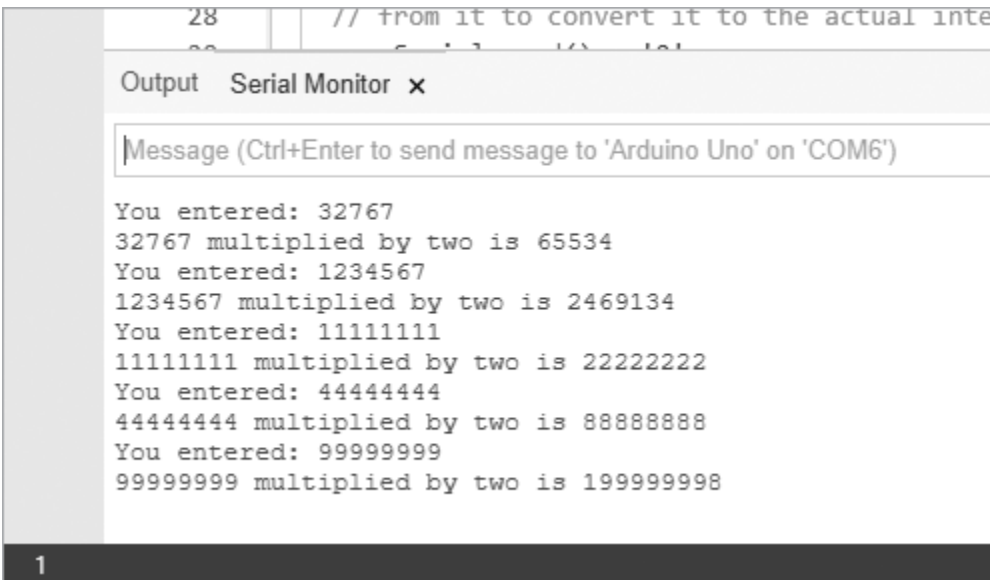


Figure 5-6: Sample input and output from Project 14

Looking Ahead

The ability to create your own functions is an important skill that will simplify your sketches and save you time and effort. You'll make good use of this knowledge in the next chapter, where you'll learn how to do more things with math on the Arduino, including making a game.

6

NUMBERS, VARIABLES, AND ARITHMETIC

In this chapter you will

- Generate random numbers

- Create electronic dice

- Learn about binary numbers

- Use shift-register integrated circuits (ICs) to get more digital output pins

- Test your knowledge of binary numbers with a quiz

- Learn about arrays of variables

- Display numbers on seven-segment LED modules

- Learn how to use the modulo math function

- Create a digital thermometer

You will learn a wide variety of useful new functions that will create more project options, including random number generation, new kinds of math functions, and variable storage in ordered lists called *arrays*. Furthermore, you will learn how to use LED display modules in numeric form to display data and simple images. Finally, we'll combine these tools to create a game, a digital thermometer, and more.

Generating Random Numbers

A program's ability to generate random numbers can be very useful in games and effects. For example, you can use random numbers to play a dice or lottery game, create lighting effects with LEDs, or create visual or auditory effects for a quiz game with the Arduino. Unfortunately, the Arduino can't choose a purely random number by itself. You have to help it by providing a *seed*, an arbitrary starting number used in the calculations to generate a random number.

Using Ambient Current to Generate a Random Number

The easiest way to generate a random number with the Arduino is to write a program that reads the voltage from a free (disconnected) analog pin (for example, analog pin 0) with this line in `void setup()`:

```
randomSeed(analogRead(0));
```

Even when nothing is wired to an analog input on the Arduino, static electricity in the environment creates a tiny, measurable voltage. The amount of this voltage is quite random. We can use this measure of ambient voltage as our seed to generate a random number and then allocate it to an integer variable using the `random(lower, upper)` function. Furthermore, we can use the parameters *lower* and *upper* to set the lower and upper limits of the range for the random number. For example, to generate a random number between 100 and 1,000, you would use the following:

```
int a = 0;  
a = random(100, 1001);
```

We've used the number 1,001 rather than 1,000 because the upper limit is *exclusive*, meaning it's not included in the range.

To generate a random number between 0 and some number, you can just enter the upper limit. Here's how you would generate a random number between 0 and 6:

```
a = random(7);
```

The example sketch in [*Listing 6-1*](#) would generate a random number between 0 and 1,000 and another random number between 10 and 50.

```
// Listing 6-1
int r = 0;

void setup()
{
  randomSeed(analogRead(0));
  Serial.begin(9600);
}

void loop()
{
  Serial.print("Random number between zero and 1000 is: ");
  r = random(0, 1001);
  Serial.println(r);
  Serial.print("Random number between ten and fifty is: ");
  r = random(10, 51);
  Serial.println(r);
  delay(1000);
}
```

Listing 6-1: A random number generator

[Figure 6-1](#) shows the result of [Listing 6-1](#) in the Serial Monitor.

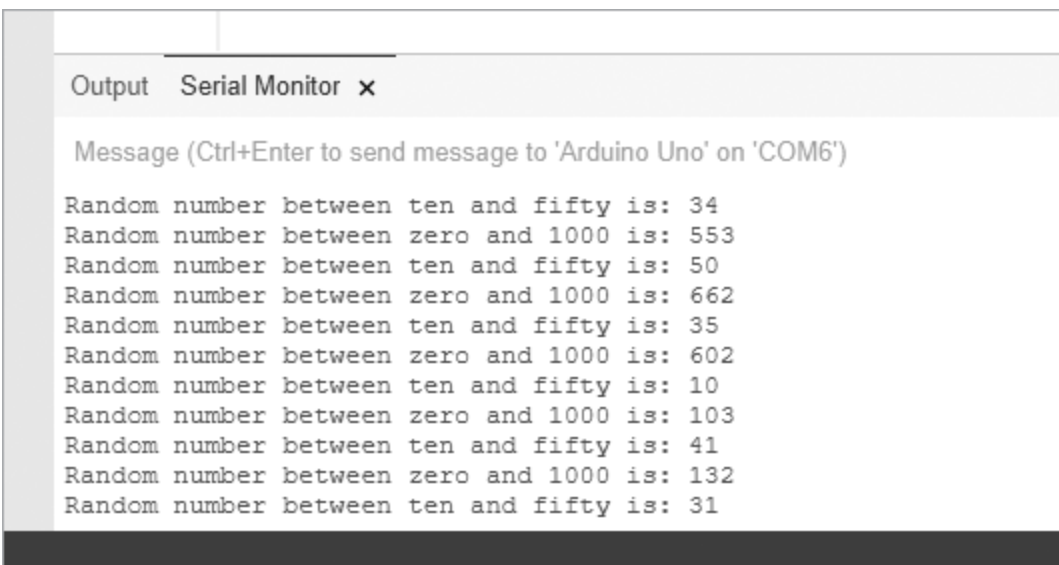


Figure 6-1: Output from [Listing 6-1](#)

Now that you know how to generate random numbers, let's put that knowledge to good use by creating an electronic die.

Project #15: Creating an Electronic Die

Our goal is to light one of six LEDs randomly to mimic the throw of a die. We'll choose a random number between 1 and 6, then turn on the corresponding LED to indicate the result. We'll create a function to select one of six LEDs on the Arduino randomly and to keep the LED on for a certain period of time. When the Arduino running the sketch is turned on or reset, it should show random LEDs rapidly for a specified period of time and then gradually slow the flashing until the final LED is lit. The LED matching the resulting randomly chosen number will stay on until the Arduino is reset or turned off.

The Hardware

To build the die, we'll need the following hardware:

Six LEDs of any color (LED1 to LED6)

One 560 Ω resistor (R1)

Various connecting wires

One medium-sized breadboard

Arduino and USB cable

The Schematic

Because only one LED will be lit at a time, a single current-limiting resistor can go between the cathodes of the LEDs and GND. [Figure 6-2](#) shows the schematic for our die.

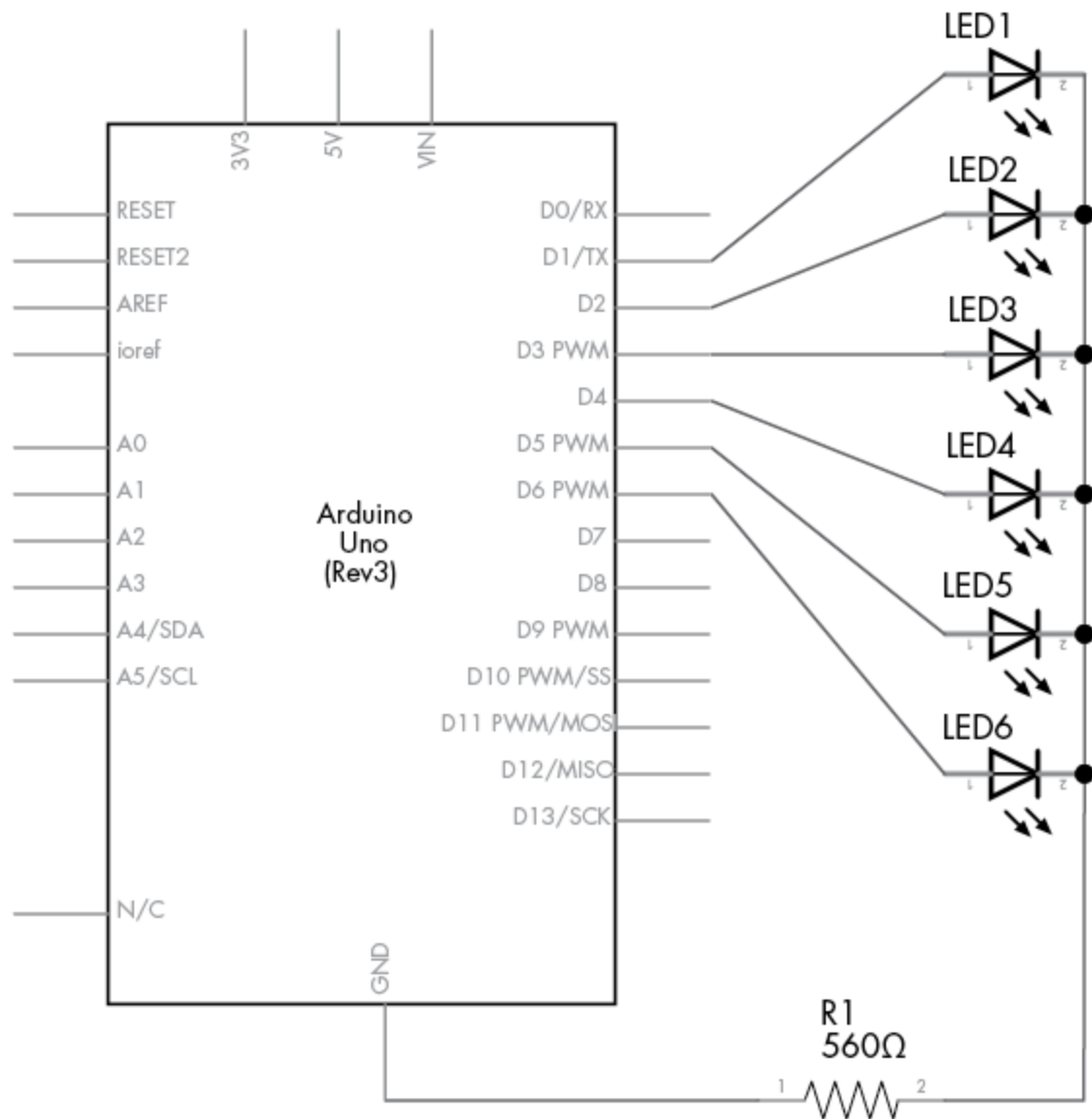


Figure 6-2: Schematic for Project 15

The Sketch

Here's the sketch for our die:

```
// Project 15 - Creating an Electronic Die
void setup()
{
  randomSeed(analogRead(0));    // seed the random number
  generator
  for ( int z = 1 ; z < 7 ; z++ ) // LEDs on pins 1-6 are
  output
  {
```

```

        pinMode(z, OUTPUT);
    }
}

void randomLED(int del)
{
    int r;
    r = random(1, 7);    // get a random number from 1 to 6
    digitalWrite(r, HIGH); // output to the matching LED on
digital pin 1-6
    if (del > 0)
    {
1      delay(del);          // hold the LED on for the delay
received
    }
2      else if (del == 0)
    {
        do                // delay entered was zero, hold the
LED on forever
        {}
3      while (1);
    }
    digitalWrite(r, LOW); // turn off the LED
}

void loop()
{
    int a;
    // cycle the LEDs around for effect
    for ( a = 0 ; a < 100 ; a++ )
    {
        randomLED(50);
    }
    // slow down
4    for ( a = 1 ; a <= 10 ; a++ )
    {
        randomLED(a * 100);
    }
    // and stop at the final random number and LED
    randomLED(0);
}

```

Here we use a loop in void `setup()` to activate the digital output pins. The function `randomLED()` receives an integer that is used in the `delay()` function at 1 to keep the LED turned on for the selected time. If the value of

the delay received at 2 is 0, then the function keeps the LED turned on indefinitely, because we use

```
do {} while (1);
```

at 3, which loops forever, because 1 is always 1.

To “roll the die,” we reset the Arduino to restart the sketch. To gradually slow the change in the LEDs before the final value is displayed, we first display a random LED 100 times for 50 milliseconds each time. Then, at 4 we slow down the flashing by increasing the delay between LED flashes from 100 to 1,000 milliseconds, with each flash lasting 100 milliseconds. The purpose of this is to simulate the “slowing down” of a die before it finally settles on a value. With the last line, the Arduino displays the outcome of the roll by keeping one LED lit:

```
randomLED(0);
```

Modifying the Sketch

We can tinker with this project in many ways. For example, we could add another six LEDs to roll two dice at once. Or display the result using only the built-in LED, by blinking it a number of times to indicate the result of the roll. Or use a button to roll the dice again. Use your imagination and new skills to have some fun!

A Quick Course in Binary

Most children learn to count using the base-10 system, but computers (including the Arduino) count using the binary number system.

Binary Numbers

Binary numbers consist of only 1s and 0s—for example, 10101010. In binary, each digit from right to left represents 2 to the power of the column number in which it appears (which increases from right to left). The products in each column are then added to determine the value of the number.

For example, consider the binary number 10101010, as shown in [Table 6-1](#). To convert the number 10101010 in binary to base 10, we add the totals in each column as listed in the bottom row of the table:

$$128 + 0 + 32 + 0 + 8 + 0 + 2 + 0$$

The sum is 170, and therefore the binary number 10101010 equals 170 in base 10. A binary number with eight columns (or *bits*) holds 1 *byte* of data; 1 byte of data can have a numerical value between 0 and 255. The leftmost bit is referred to as the *most significant bit (MSB)*, and the rightmost is the *least significant bit (LSB)*.

Table 6-1: Binary to Base-10 Number Conversion Example

2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰	
1	0	1	0	1	0	1	0	Binary
128	64	32	16	8	4	2	1	Base 10

Binary numbers are great for storing certain types of data, such as on/off patterns for LEDs, true/false settings, and the statuses of digital outputs. Binary numbers are the building blocks of all types of data in computers.

Byte Variables

One way we can store binary numbers is by using a *byte variable*. For example, we can create the byte variable outputs using the following code:

```
byte outputs = B11111111;
```

The B in front of the number tells Arduino to read the number as a binary number (in this case, 11111111) instead of its base-10 equivalent of 255.

[Listing 6-2](#) demonstrates this further.

```
// Listing 6-2

byte a;

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

```

}

void loop()
{
  for ( int count = 0 ; count < 256 ; count++ )
  {
    a = count;
    Serial.print("Base-10 = ");
1    Serial.print(a, DEC);
    Serial.print(" Binary = ");
2    Serial.println(a, BIN);
    delay(1000);
  }
}

```

Listing 6-2: Binary number demonstration

We display byte variables as base-10 numbers using DEC 1 or as binary numbers using BIN 2 as part of the `Serial.print()` function. After uploading the sketch, you should see output in the Serial Monitor similar to that shown in [Figure 6-3](#).

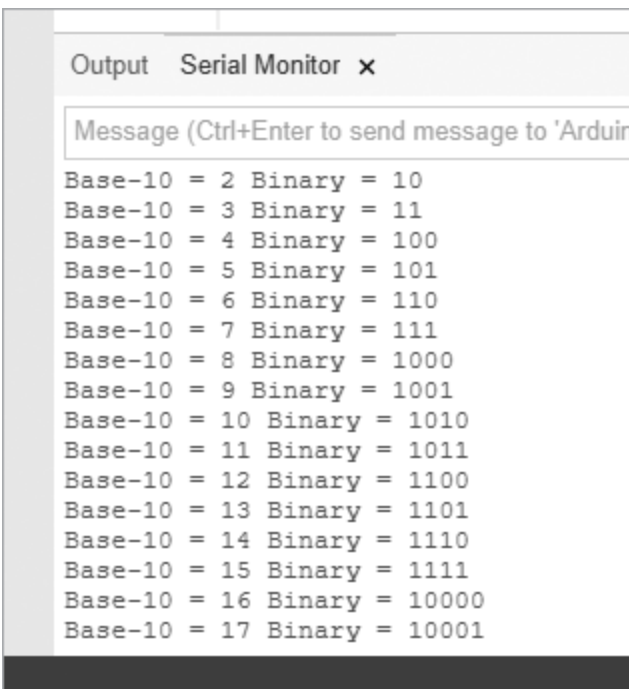
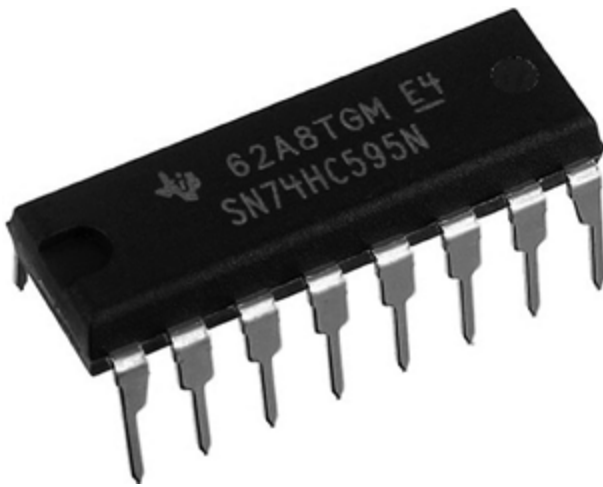


Figure 6-3: Output from [Listing 6-2](#)

Increasing Digital Outputs with Shift Registers

The Arduino board has 13 digital pins that we can use as outputs—but sometimes 13 just isn't enough. To add outputs, we can use a *shift register* and still have plenty of room left on the Arduino for outputs. A shift register is an integrated circuit (IC) with eight digital output pins that can be controlled by sending a byte of data to the IC. For our projects, we will be using the 74HC595 shift register shown in [Figure 6-4](#).



[Figure 6-4](#): The 74HC595 shift register IC

The 74HC595 shift register has eight digital outputs that can operate like the Arduino digital output pins. The shift register itself takes up three Arduino digital output pins, so the net gain is five output pins.

The principle behind the shift register is simple: we send 1 byte of data (8 bits) to the shift register, and it turns on or off the matching eight outputs based on the 1 byte of data. The bits representing the byte of data match the output pins in order from highest to lowest, so the leftmost bit of the data represents output pin 7 of the shift register, and the rightmost bit of the data represents output pin 0. For example, if we send `B10000110` to the shift register, then it will turn on outputs 1, 2, and 7 and will turn off outputs 0 and 3 to 6 until the next byte of data is received or the power is turned off.

More than one shift register can be connected together to provide an extra eight digital output pins for every shift register attached to the same three

Arduino pins; this makes shift registers very convenient when you want to control lots of LEDs. Let's do that now by creating a binary number display.

Project #16: Creating an LED Binary Number Display

In this project, we'll use eight LEDs to display binary numbers from 0 to 255. Our sketch will use a `for` loop to count from 0 to 255 and will send each value to the shift register, which will use LEDs to display the binary equivalent of each number.

The Hardware

The following hardware is required:

One 74HC595 shift register IC

Eight LEDs (LED1 to LED8)

Eight 560 Ω resistors (R1 to R8)

One breadboard

Various connecting wires

Arduino and USB cable

The Schematic

[Figure 6-5](#) shows the schematic symbol for the 74HC595.

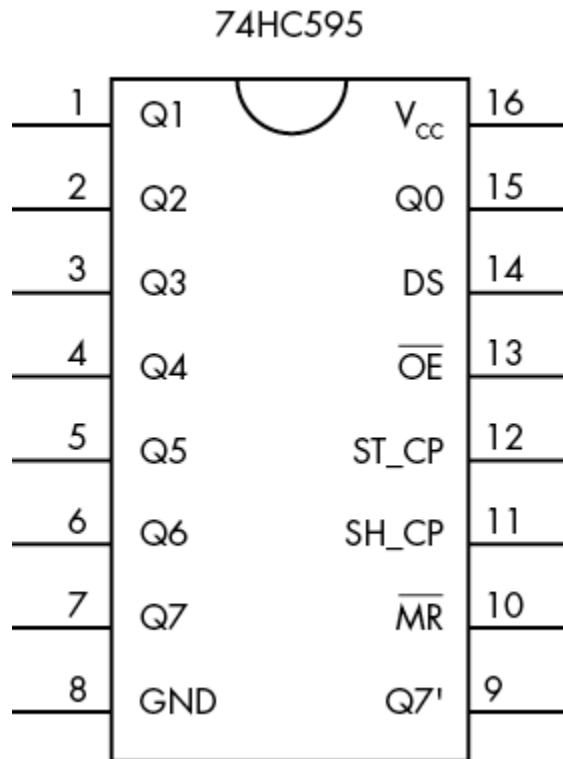


Figure 6-5: 74HC595 schematic symbol

There are 16 pins on our shift register:

Pins 15 and 1 to 7 are the eight output pins that we control (labeled Q0 to Q7, respectively).

Q7 outputs the first bit sent to the shift register and Q0 outputs the last.

Pin 8 connects to GND.

Pin 9 is called *data out* and is used to send data to another shift register if one is present.

Pin 10 is always connected to 5 V (for example, the 5 V connector on the Arduino).

Pins 11 and 12 are called *clock* and *latch*.

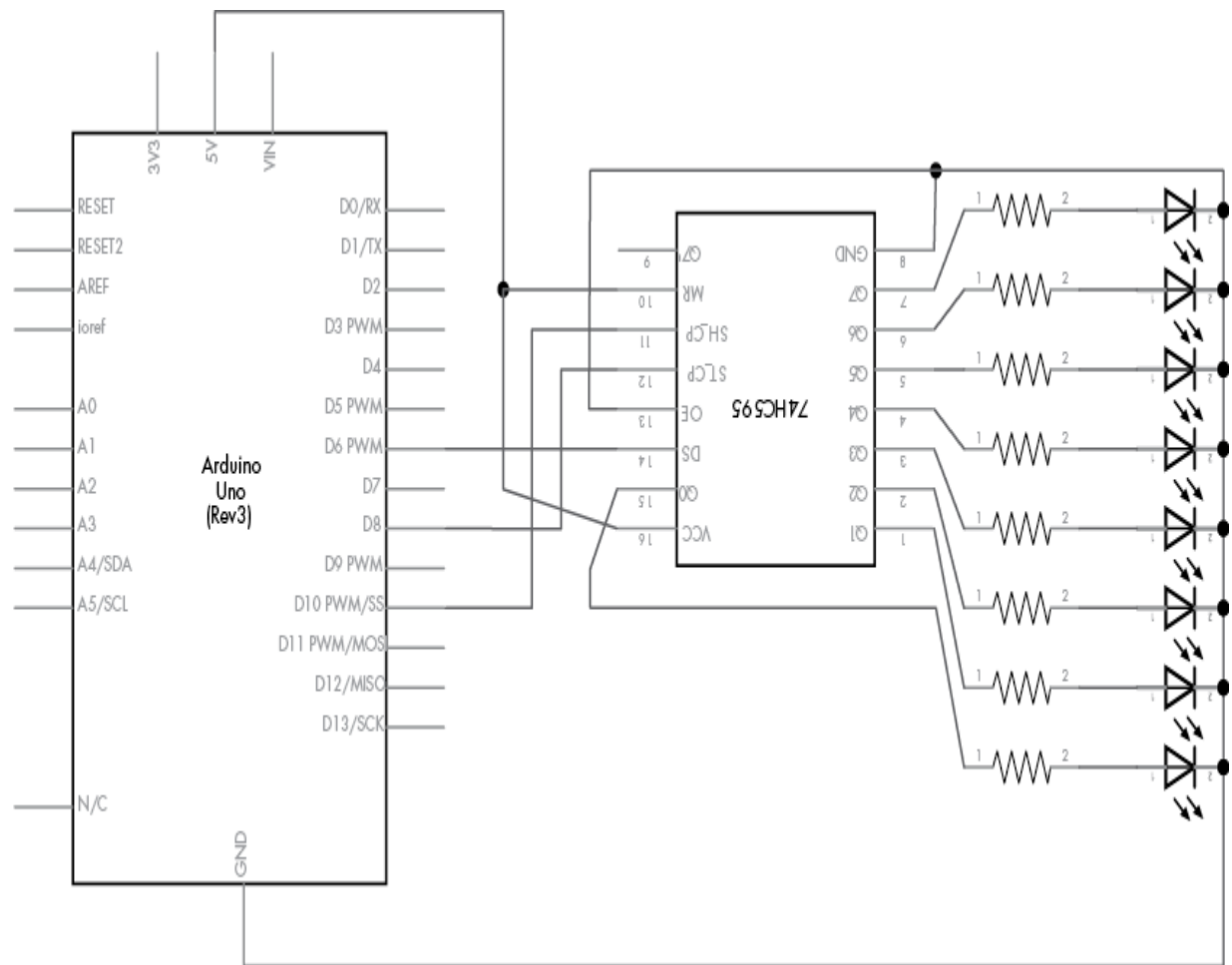
Pin 13 is called *output enable* and is usually connected to GND.

Pin 14 is for incoming bit data sent from the Arduino.

Pin 16 is used for power: 5 V from the Arduino.

To give you a sense of the way the pins are oriented, the semicircular notch on the left end of the body of the shift register IC shown in [Figure 6-4](#) lies between pins 1 and 16.

The pins are numbered sequentially around the body in a counterclockwise direction, as shown in [Figure 6-6](#), the schematic for our LED binary number display.



[Figure 6-6](#): Schematic for Project 16

NOTE

Once you have finished with this example circuit, keep it assembled. We'll use it again for the next project.

The Sketch

And now for the sketch:

```
// Project 16 - Creating an LED Binary Number Display
#define DATA 6          // digital 6 to pin 14 on the
74HC595
#define LATCH 8          // digital 8 to pin 12 on the
74HC595
#define CLOCK 10         // digital 10 to pin 11 on the
74HC595

void setup()
{
  pinMode(LATCH, OUTPUT);
  pinMode(CLOCK, OUTPUT);
  pinMode(DATA, OUTPUT);
}

void loop()
{
  int i;
  for ( i = 0; i < 256; i++ )
  {
    digitalWrite(LATCH, LOW);
    shiftOut(DATA, CLOCK, MSBFIRST, i);
    digitalWrite(LATCH, HIGH);
    delay(200);
  }
}
```

In this sketch, we set the three pins connected to the shift register as outputs in `void setup()` and then add a loop in `void loop()` that counts from 0 to 255 and repeats. The magic lies inside the loop. When we send a byte of data (for example, 240, or B11110000) to the shift register in the `for` loop, three things happen:

The latch pin 12 is set to LOW (that is, a low signal is applied to it from the Arduino digital output pin 8). This is preparation for setting output pin 12 to HIGH, which latches the data to the output pins after `shiftOut()` has completed its task.

We send the byte of data (for example, B11110000) from Arduino digital pin 6 to the shift register and tell the `shiftOut()` function from which direction to interpret the byte of data. For example, if we selected `LSBFIRST`, then

LEDs 1 to 4 would turn on and the others would turn off. If we used MSBFIRST, then LEDs 5 to 8 would turn on and the others would turn off.

Finally, the latch pin 12 is set to HIGH (5 V is applied to it). This tells the shift register that all the bits are shifted in and ready. At this point, the shift register alters its output to match the data received.

Project #17: Making a Binary Quiz Game

In this project, we'll use random numbers, the Serial Monitor, and the circuit created in Project 16 to create a binary quiz game. The Arduino will display a random binary number using the LEDs, and then you will enter the decimal version of the binary number using the Serial Monitor. The Serial Monitor will tell you whether your answer is correct, and the game will continue with a new number.

The Algorithm

The algorithm can be divided into three functions. The `displayNumber()` function will display a binary number using the LEDs. The `getAnswer()` function will receive a number from the Serial Monitor and display it to the user. Finally, the `checkAnswer()` function will compare the user's number to the random number generated and display the correct/incorrect status, as well as the correct answer if the guess was incorrect.

The Sketch

The sketch generates a random number between 0 and 255, displays it in binary using the LEDs, asks the user for their answer, and then displays the result in the Serial Monitor. You've already seen all the functions used in the sketch, so although there's a lot of code here, it should look familiar. We'll dissect it with comments within the sketch and some commentary following:

```
// Project 17 - Making a Binary Quiz Game

#define DATA    6                // connect to pin 14 on the
74HC595
#define LATCH    8                // connect to pin 12 on the
```

```

74HC595
#define CLOCK 10                // connect to pin 11 on the
74HC595

int number = 0;
int answer = 0;

1 void setup()
{
    pinMode(LATCH, OUTPUT);      // set up the 74HC595 pins
    pinMode(CLOCK, OUTPUT);
    pinMode(DATA, OUTPUT);
    Serial.begin(9600);
    randomSeed(analogRead(0));   // initialize the random
number generator
    displayNumber(0);           // clear the LEDs
}

2 void displayNumber(byte a)
{
    // send byte to be displayed on the LEDs
    digitalWrite(LATCH, LOW);

    shiftOut(DATA, CLOCK, MSBFIRST, a);
    digitalWrite(LATCH, HIGH);
}

3 void getAnswer()
{
    // receive the answer from the player
    int z = 0;
    Serial.flush();
    while (Serial.available() == 0)
    {
        // do nothing until something comes into the serial
buffer
    }
    // one character of serial data is available, begin
calculating
    while (Serial.available() > 0)
    {
        // move any previous digit to the next column on the
left; in
        // other words, 1 becomes 10 while there is data in the
buffer
        answer = answer * 10;
        // read the next number in the buffer and subtract the

```

```

character '0'
    // from it to convert it to the actual integer number
    z = Serial.read() - '0';
    // add this digit into the accumulating value
    answer = answer * 10 + z;
    // allow a short delay for any more numbers to come into
Serial.available
    delay(5);
}
Serial.print("You entered: ");
Serial.println(answer);
}

```

```

4 void checkAnswer()
{
    // check the answer from the player and show the results
    if (answer == number)    // Correct!
    {
        Serial.print("Correct! ");
        Serial.print(answer, BIN);
        Serial.print(" equals ");
        Serial.println(number);
        Serial.println();
    }
    else                        // Incorrect
    {
        Serial.print("Incorrect, ");
        Serial.print(number, BIN);
        Serial.print(" equals ");
        Serial.println(number);
        Serial.println();
    }
    answer = 0;
    delay(10000); // give the player time to review their
answer
}

```

```

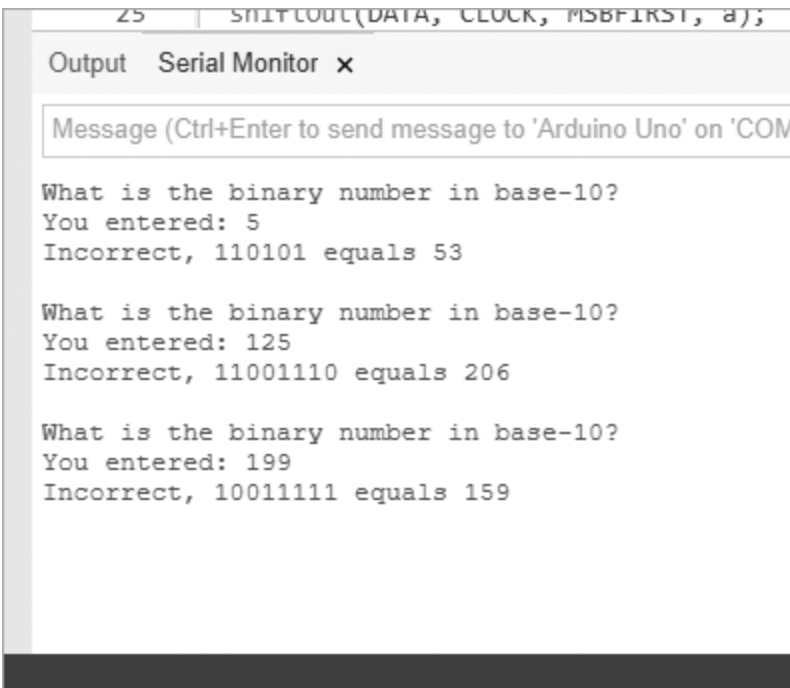
5 void loop()
{
    number = random(256);
    displayNumber(number);
    Serial.println("What is the binary number in base 10? ");
    getAnswer();
    checkAnswer();
}

```

Let's review how the sketch works. At 1, `void setup()` configures the digital output pins to use the shift register, starts the Serial Monitor, and seeds the random number generator. At 2, the custom function `displayNumber()` accepts a byte of data and sends it to the shift register, which uses LEDs to display the byte in binary form via the attached LEDs (as in Project 16). At 3, the custom function `getAnswer()` accepts a number from the user via the Serial Monitor (as in Project 14 in Chapter 5) and displays it, as shown in [Figure 6-7](#).

The function `checkAnswer()` at 4 compares the number entered by the player in `getAnswer()` against the random number generated by the sketch in `void loop()`. The player is then advised of a correct or incorrect answer with corresponding binary and decimal values. Finally, in the main `void loop()` at 5 from which the program runs, the Arduino generates the random binary number for the quiz, calls the matching functions to display it with hardware, and then receives and checks the player's answer.

[Figure 6-7](#) shows the game in play in the Serial Monitor.



[Figure 6-7](#): Project 17 in play

Arrays

An *array* is a set of variables or values grouped together so that they can be referenced as a whole. When dealing with lots of related data, you'll find it a good idea to use arrays to keep your data organized.

Defining an Array

Each item in an array is called an *element*. For example, suppose six `float` variables contain temperatures taken over the last six hours; instead of giving them all separate names, we can define an array called `temperatures` with six elements like this:

```
float temperatures[6];
```

We can also insert values when defining the array. When we do that, we don't need to define the array size. Here's an example:

```
float temperatures[]={11.1, 12.2, 13.3, 14.4, 15.5, 16.6};
```

Notice that this time, we didn't explicitly define the size of the array within the square brackets (`[]`); instead, its size is deduced based on the number of elements set by the values inside the curly brackets (`{}`). Note that arrays of any size can only contain one type of variable.

Referring to Values in an Array

We count the elements in an array beginning from the left and starting from 0; the `temperatures[]` array has elements numbered 0 to 5. We can refer to individual values within an array by inserting the number of the element in the square brackets. For example, to change the first element in `temperatures[]` (currently 11.1) to 12.34, we would use this:

```
temperatures[0] = 12.34;
```

Writing to and Reading from Arrays

In [Listing 6-3](#), we demonstrate writing values to and reading values from an array of five elements. The first `for` loop in the sketch writes a random

number into each of the array's elements, and the second for loop retrieves the elements and displays them in the Serial Monitor.

```
// Listing 6-3

void setup()
{
  Serial.begin(9600);
  randomSeed(analogRead(0));
}
int array[5];    // define our array of five integer elements
void loop()
{
  int i;
  Serial.println();
  for ( i = 0 ; i < 5 ; i++ )    // write to the array
  {
    array[i] = random(10);      // random numbers from 0 to 9
  }
  for ( i = 0 ; i < 5 ; i++ )    // display the contents of
the array
  {
    Serial.print("array[");
    Serial.print(i);
    Serial.print("] contains ");
    Serial.println(array[i]);
  }
  delay(5000);
}
```

Listing 6-3: Array read/write demonstration

[Figure 6-8](#) shows the output of this sketch in the Serial Monitor.

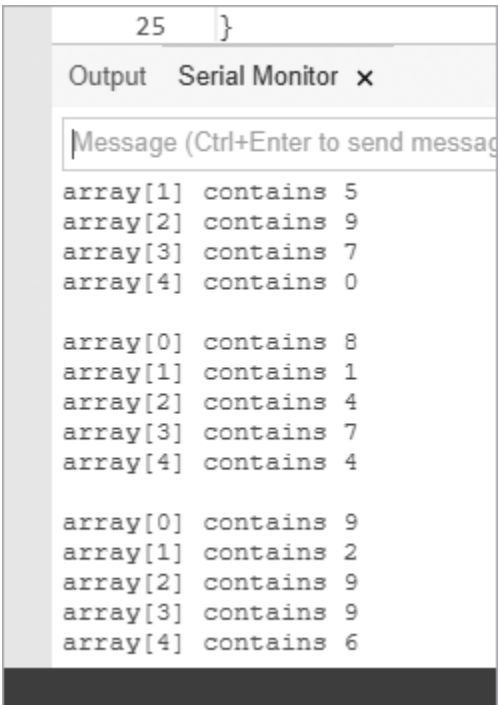


Figure 6-8: [Listing 6-3](#) in action

Now that you know how to use with binary numbers, shift registers, and arrays, it's time to put that knowledge to use. In our next project, we'll wire up some digital number displays.

Seven-Segment LED Displays

LEDs are fun, but there are limits to the kinds of data that can be displayed with individual lights. In this section, we'll begin working with numeric digits in the form of seven-segment LED displays, as shown in [Figure 6-9](#).

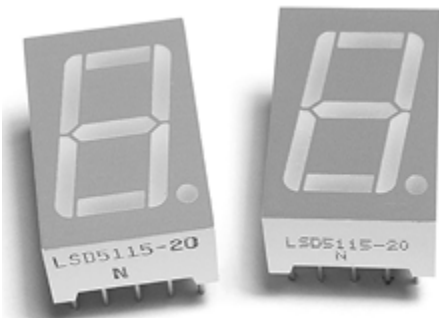


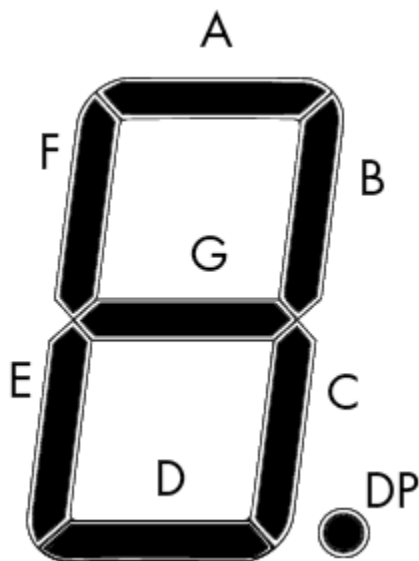
Figure 6-9: Seven-segment display modules

These displays are perfect for displaying numbers, and that's why you'll find them used in digital alarm clocks, speedometers, and the like. Each module in a seven-segment LED display consists of eight LEDs. The modules are also available in different colors. To reduce the number of pins used by the display, all of the anodes or cathodes of the LEDs are connected together—these are called *common-anode* or *common-cathode* modules, respectively. Our projects will use common-cathode modules.

The display's LEDs are labeled *A* to *G* and *DP* (for the decimal point). There is an anode pin for each LED segment, and the cathodes are connected to one common cathode pin. The layout of seven-segment LED displays is always described as shown in [Figure 6-10](#), with LED segment *A* at the top, *B* to its right, and so on. So, for example, if you wanted to display the number 7, then you would apply current to segments *A*, *B*, and *C*.

The pins on each LED display module can vary, depending on the manufacturer, but they always follow the basic pattern shown in [Figure 6-10](#). When you use one of these modules, always get the data sheet for the module from the retailer to help save you time determining which pins are which.

We'll use the schematic symbol shown in [Figure 6-11](#) for our seven-segment LED display modules.



[Figure 6-10](#): LED map for a typical seven-segment display module

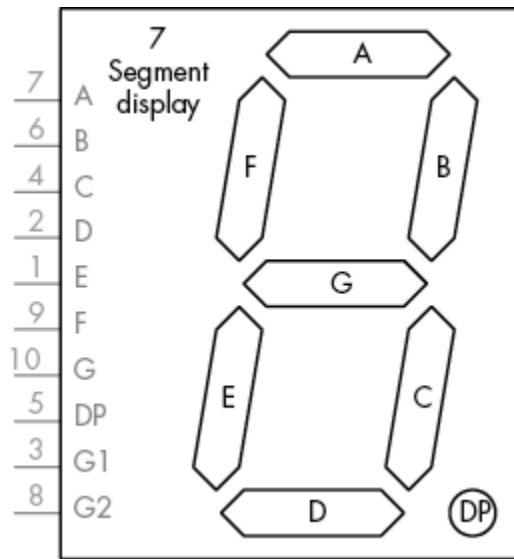


Figure 6-11: Schematic symbol for a seven-segment display module

Controlling the LED

We'll control the LED display using the method discussed in Project 17, by connecting pins A through DP to the shift register outputs Q0 to Q7. Use the matrix shown in [Table 6-2](#) as a guide to help determine which segments to turn on and off to display a particular number or letter.

The top row in the matrix is the shift register output pin that controls the segments on the second row. Each row below this shows the digit that can be displayed with the corresponding binary and decimal value to send to the shift register.

Table 6-2: *Display Segment Matrix*

SR	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7	
Segment	A	B	C	D	E	F	G	DP	Decimal
0	1	1	1	1	1	1	0	0	252
1	0	1	1	0	0	0	0	0	96
2	1	1	0	1	1	0	1	0	218
3	1	1	1	1	0	0	1	0	242
4	0	1	1	0	0	1	1	0	102
5	1	0	1	1	0	1	1	0	182
6	1	0	1	1	1	1	1	0	190
7	1	1	1	0	0	0	0	0	224
8	1	1	1	1	1	1	1	0	254
9	1	1	1	1	0	1	1	0	246
A	1	1	1	0	1	1	1	0	238
B	0	0	1	1	1	1	1	0	62
C	1	0	0	1	1	1	0	0	156
D	0	1	1	1	1	0	1	0	122
E	1	0	0	1	1	1	1	0	158
F	1	0	0	0	1	1	1	0	142

For example, to display the digit 7, as shown in [Figure 6-12](#), we need to turn on LED segments A, B, and C, which correspond to the shift register outputs Q0, Q1, and Q2. Therefore, we will send the byte B1110000 into the shift register (with `shiftOut()` set to `LSBFIRST`) to turn on the first three outputs that match the desired LEDs on the module.

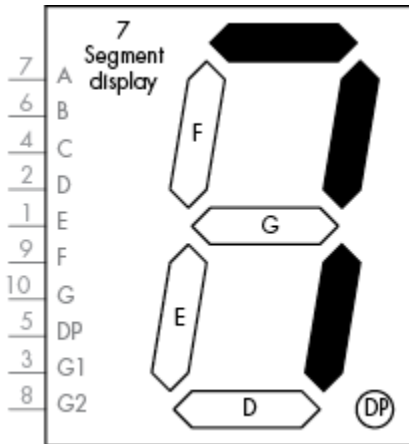


Figure 6-12: Displaying the digit 7

In our next project, we'll create a circuit that displays, in turn, the digits 0 through 9 and then the letters A through F. The cycle repeats with the decimal-point LED turned on.

Project #18: Creating a Single-Digit Display

In this project we'll assemble a circuit to use a single-digit display.

The Hardware

The following hardware is required:

One 74HC595 shift register IC

One common-cathode seven-segment LED display

One 560 Ω resistor (R1)

One large breadboard

Various connecting wires

Arduino and USB cable

The Schematic

The schematic is shown in [Figure 6-13](#).

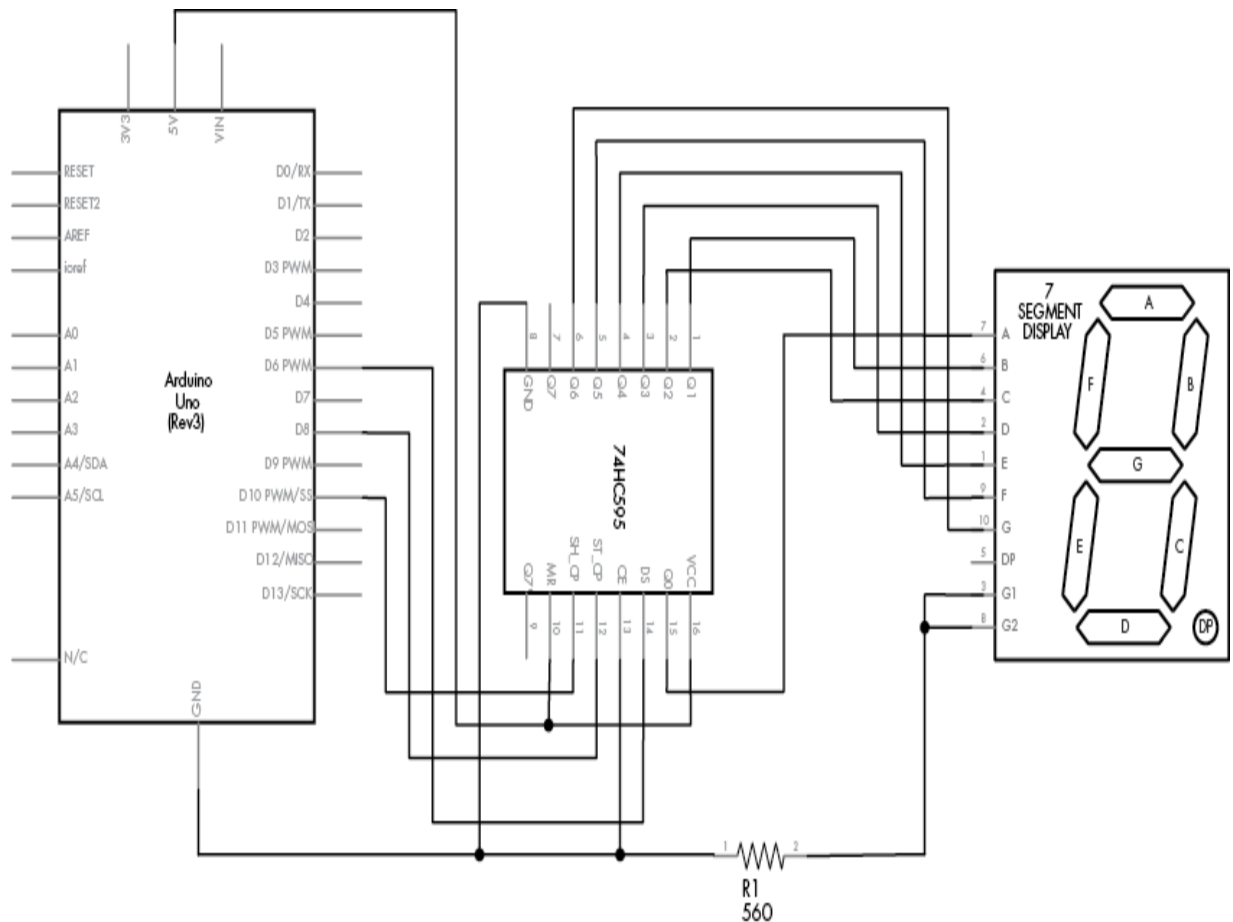


Figure 6-13: Schematic for Project 18

When you're wiring the LED module to the shift register, LED pins A through G connect to pins Q0 through Q6, respectively, and DP connects to Q7.

The Sketch

In the sketch for Project 18, we store the decimal values (see [Table 6-2](#)) in the `int digits[]` array. In the `void loop()`, we send these values to the shift register in sequential order at 1 and then repeat the process with the decimal point on by adding 1 to the value sent to the shift register at 2:

```
// Project 18 - Creating a Single-Digit Display
#define DATA 6 // connect to pin 14 on
the 74HC595
#define LATCH 8 // connect to pin 12 on
the 74HC595
```

```

#define CLOCK 10                                // connect to pin 11 on
the 74HC595

// set up the array with the segments for 0 to 9, A to F
(from Table 6-2)
int digits[] = {252, 96, 218, 242, 102, 182, 190, 224, 254,
246, 238, 62, 156, 122, 158, 142};

void setup()
{
  pinMode(LATCH, OUTPUT);
  pinMode(CLOCK, OUTPUT);
  pinMode(DATA, OUTPUT);
}

void loop()
{
  int i;
  for ( i = 0 ; i < 16 ; i++ )    // display digits 0-9, A-F
  {
    digitalWrite(LATCH, LOW);
1  shiftOut(DATA, CLOCK, LSBFIRST, digits[i]);
    digitalWrite(LATCH, HIGH);
    delay(250);
  }
  for ( i = 0 ; i < 16 ; i++ )    // display digits 0-9, A-F
  with DP
  {
    digitalWrite(LATCH, LOW);
2  shiftOut(DATA, CLOCK, LSBFIRST, digits[i]+1); // +1 is to turn
    on the DP bit
    digitalWrite(LATCH, HIGH);
    delay(250);
  }
}

```

Seven-segment LED displays are bright and easy to read. For example, [Figure 6-14](#) shows the result when this sketch is asked to display the digit 9 with the decimal point.



Figure 6-14: Digit displayed by Project 18

Modifying the Sketch: Displaying Double Digits

To use more than one shift register to control additional digital outputs, connect pin 9 of the 74HC595 (which receives data from the Arduino) to pin 14 of the second shift register. Once you've made this connection, two bytes of data will be sent: the first to control the second shift register and the second to control the first shift register. Here's an example:

```
digitalWrite(LATCH, LOW);  
shiftOut(DATA, CLOCK, MSBFIRST, 254); // data for second  
74HC595  
shiftOut(DATA, CLOCK, MSBFIRST, 254); // data for first  
74HC595  
digitalWrite(LATCH, HIGH);
```

Project #19: Controlling Two Seven-Segment LED Display Modules

This project will show you how to control two seven-segment LED display modules so that you can display two-digit numbers.

The Hardware

The following hardware is required:

Two 74HC595 shift register ICs

Two common-cathode seven-segment LED displays

Two 560 Ω resistors (R1 to R2)

One large breadboard or two smaller units

Various connecting wires

Arduino and USB cable

The Schematic

[Figure 6-15](#) shows the schematic for two display modules.

Note that the shift registers' data and clock pins are connected to each other and then to the Arduino. The data line from Arduino digital pin 6 runs to shift register 1, and then a link from pin 9 of shift register 1 runs to pin 14 of shift register 2.

To display a number between 0 and 99, we'll need a more complicated sketch. If a number is less than 10, we can just send the number followed by a 0, as the right digit will display the number and the left digit will display 0. However, if the number is greater than 10, then we need to determine each of the number's two digits and send each to the shift registers separately. To make this process easier, we'll use the math function modulo.

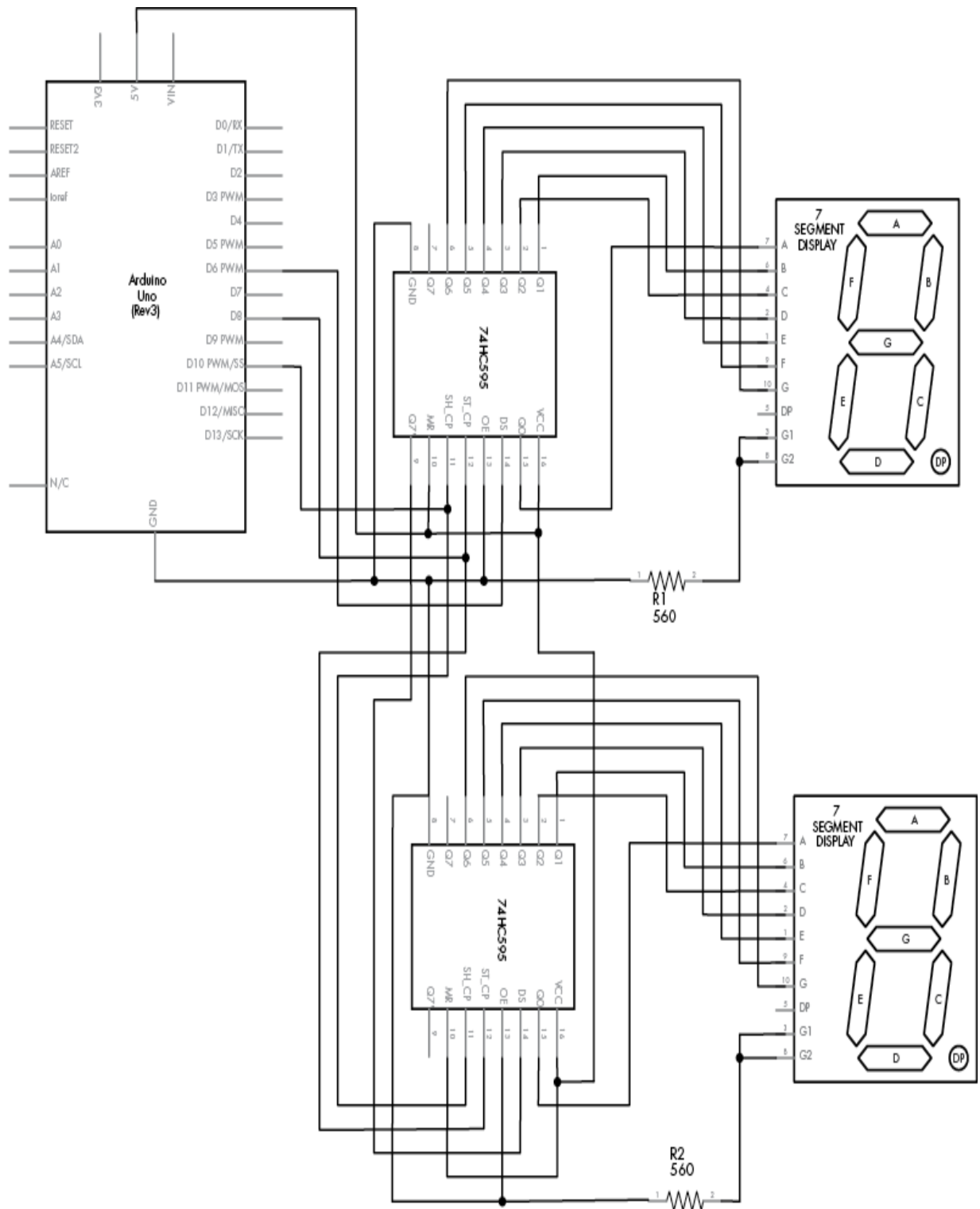


Figure 6-15: Schematic for Project 19

Modulo

Modulo is a function that returns the remainder of a division operation. For example, 10 modulo (or *mod*) 7 equals 3—in other words, the remainder of 10 divided by 7 equals 3. We use the percent sign (%) to represent modulo. The following example uses modulo in a sketch:

```
int a = 8;
int b = 3;
int c = a % b;
```

In this example, the value of *c* will be 2. So, to determine a two-digit number's right-hand digit, we use the modulo function, which returns the remainder when dividing the two numbers.

To automate displaying a single- or double-digit number, we'll create the function `displayNumber()` for our sketch. We use modulo as part of this function to separate the digits of a two-digit number. For example, to display the number 23, we first isolate the left-hand digit by dividing 23 by 10, yielding 2 (and a fraction that we can ignore). To isolate the right-hand digit, we perform 23 modulo 10, which equals 3:

```
// Project 19 - Controlling Two Seven-Segment LED Display
Modules
// set up the array with the segments for 0 to 9, A to F
(from Table 6-2)

#define DATA 6           // connect to pin 14 on the
74HC595
#define LATCH 8           // connect to pin 12 on the
74HC595
#define CLOCK 10          // connect to pin 11 on the
74HC595

void setup()
{
  pinMode(LATCH, OUTPUT);
  pinMode(CLOCK, OUTPUT);
  pinMode(DATA, OUTPUT);
}

int digits[] = {252, 96, 218, 242, 102, 182, 190, 224, 254,
246, 238, 62, 156, 122, 158, 142};
void displayNumber(int n)
{
```

```

    int left, right=0;
1  if (n < 10)
    {
        digitalWrite(LATCH, LOW);
        shiftOut(DATA, CLOCK, LSBFIRST, digits[n]);
        shiftOut(DATA, CLOCK, LSBFIRST, 0);
        digitalWrite(LATCH, HIGH);
    }
    else if (n >= 10)
    {
2      right = n % 10; // remainder of dividing the number to
display by 10
        left = n / 10; // quotient of dividing the number to
display by 10
        digitalWrite(LATCH, LOW);
        shiftOut(DATA, CLOCK, LSBFIRST, digits[right]);
        shiftOut(DATA, CLOCK, LSBFIRST, digits[left]);
        digitalWrite(LATCH, HIGH);
    }
}
3 void loop()
{
    int i;
    for ( i = 0 ; i < 100 ; i++ )
    {
        displayNumber(i);
        delay(100);
    }
}

```

At 1, the function checks whether the number to be displayed is less than 10. If so, it sends the data for the number and a blank digit to the shift registers. However, if the number is greater than 10, the function uses modulo and division at 2 to separate the digits and then sends them to the shift registers separately. Finally, in `void loop()` at 3, we set up and call the function to display the numbers from 0 to 99.

Project #20: Creating a Digital Thermometer

In this project, we'll add the TMP36 temperature sensor we created in Project 8 in Chapter 4 to the double-digit circuit constructed for Project 19 to create a digital thermometer that displays values for 0 degrees and above.

The algorithm is simple: we read the voltage returned from the TMP36 (using the method from Project 12 in Chapter 5) and convert the reading to degrees Celsius.

The Hardware

The following hardware is required:

The double-digit circuit from Project 19

One TMP36 temperature sensor

Connect the center output lead of the TMP36 to analog pin 5, the left lead to 5 V, and the right lead to GND, and you're ready to measure.

The Sketch

Here is the sketch:

```
// Project 20 - Creating a Digital Thermometer
#define DATA 6           // connect to pin 14 on the
74HC595
#define LATCH 8           // connect to pin 12 on the
74HC595
#define CLOCK 10          // connect to pin 11 on the
74HC595

int temp = 0;
float voltage = 0;
float celsius = 0;
float sensor = 0;
int digits[]={
    252, 96, 218, 242, 102, 182, 190, 224,
    254, 246, 238, 62, 156, 122, 158, 142
};

void setup()
{
    pinMode(LATCH, OUTPUT);
    pinMode(CLOCK, OUTPUT);
    pinMode(DATA, OUTPUT);
}

void displayNumber(int n)
{
```

```

int left, right = 0;
if (n < 10)
{
    digitalWrite(LATCH, LOW);
    shiftOut(DATA, CLOCK, LSBFIRST, digits[n]);
    shiftOut(DATA, CLOCK, LSBFIRST, digits[0]);
    digitalWrite(LATCH, HIGH);
}
if (n >= 10)
{
    right = n % 10;
    left = n / 10;
    digitalWrite(LATCH, LOW);
    shiftOut(DATA, CLOCK, LSBFIRST, digits[right]);
    shiftOut(DATA, CLOCK, LSBFIRST, digits[left]);
    digitalWrite(LATCH, HIGH);
}
}

void loop()
{
    sensor = analogRead(5);
    voltage = (sensor * 5000) / 1024; // convert raw sensor
value to millivolts
    voltage = voltage - 500;           // remove voltage offset
    celsius = voltage / 10;           // convert millivolts to
Celsius
    temp = int(celsius); // change the floating-point
temperature to an int
    displayNumber(temp);
    delay(500);
}

```

As indicated, the sketch borrows code from previous projects: `displayNumber()` from Project 19 and the temperature calculations from Project 12. The `delay(500)` function in the second-to-last line of the sketch keeps the display from changing too quickly when the temperature fluctuates.

Looking Ahead

In this chapter, you have learned a lot of fundamental skills that you'll use over and over in your own projects. LED displays are relatively hardy, so

enjoy experimenting with them. However, there is a limit to the display effects they can be used for, so in the next chapter, we make use of much more detailed display methods for text and graphics.

7

EXPANDING YOUR ARDUINO

In this chapter you will

Learn about the broad variety of Arduino shields

Make your own Arduino shield using a ProtoShield

See how Arduino libraries can expand the available functions

Use a memory card module to record data that can be analyzed in a spreadsheet

Build a temperature-logging device

Learn how to make a stopwatch using `micros()` and `millis()`

Understand Arduino interrupts and their uses

Another way to expand the capabilities of your Arduino is by using shields. A *shield* is a circuit board that connects via pins to the sockets on the sides of an Arduino. In the first project in this chapter, you'll learn how to make your own shield. Over time, as you experiment with electronics and Arduino, you can make your circuits more permanent by building them onto a *ProtoShield*, a blank printed circuit board that you can use to mount custom circuitry.

Next, I'll introduce a memory card module. We'll use it in this chapter to create a temperature-logging device to record temperatures over time; the shield will be used to record data from the Arduino to be transferred elsewhere for analysis.

You'll learn about the functions `micros()` and `millis()`, which are very useful for keeping time, as you'll see in the stopwatch project. Finally, we'll examine interrupts.

Shields

You can add functionality to your Arduino board by attaching shields. Hundreds of shields are available on the market, and they can be combined, or stacked, to work together. One popular project, for example, combines a GPS shield with a microSD memory card shield to create a device that logs and stores position over time, for example to record a car's path of travel or the location of a new hiking trail. Other projects include Ethernet network adapters that let the Arduino access the internet ([Figure 7-1](#)).

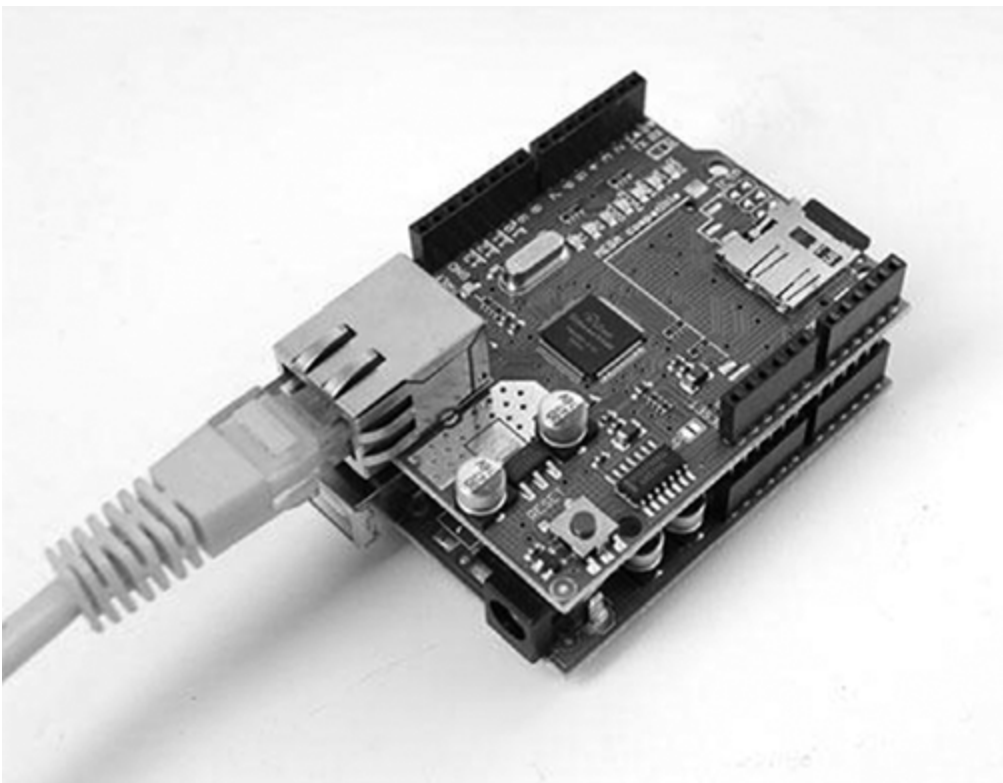
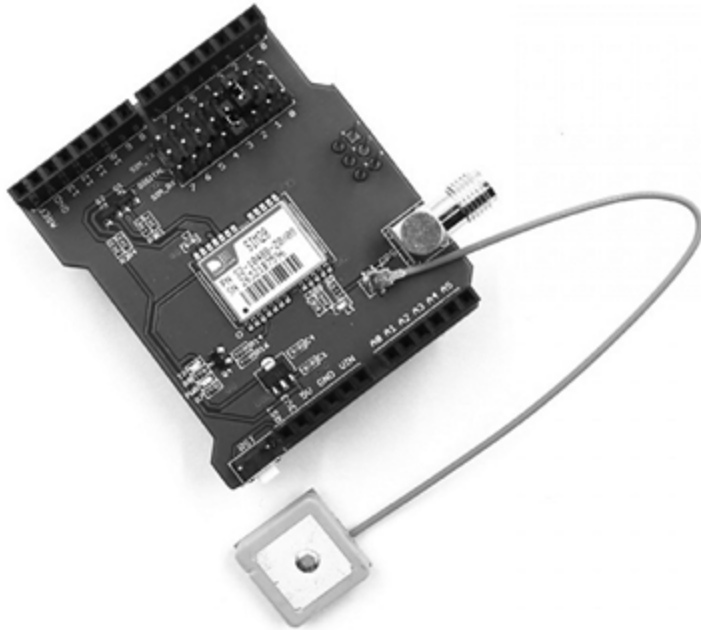


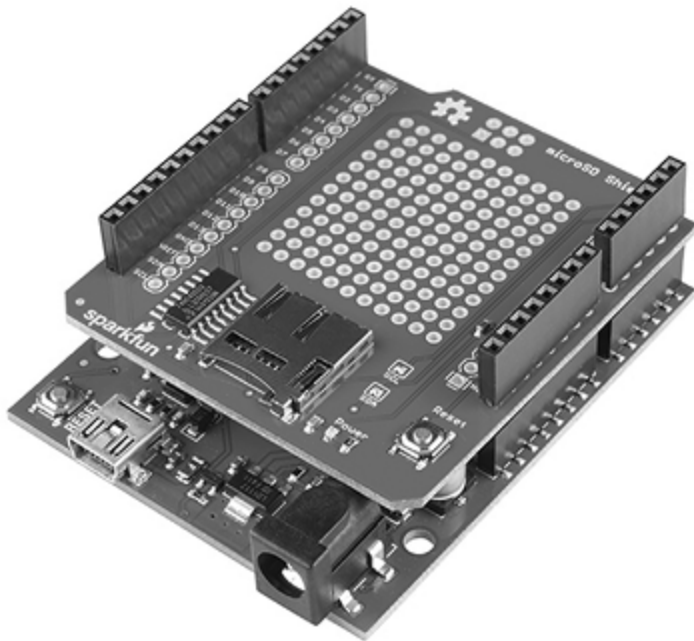
Figure 7-1: An Ethernet shield on an Arduino Uno

GPS satellite receivers let you track the location of the Arduino ([Figure 7-2](#)). MicroSD memory card interfaces let the Arduino store data on a memory card ([Figure 7-3](#)).

[*Figure 7-4*](#) shows a stack that includes an Arduino Uno, a microSD memory card shield to which data can be recorded, an Ethernet shield for connecting to the internet, and an LCD shield to display information.



[*Figure 7-2*](#): A GPS receiver shield (with separate GPS module)



[*Figure 7-3*](#): A MicroSD card shield



Figure 7-4: Three stacked shields with an Arduino Uno

WARNING

When stacking shields, make sure that no shield uses the same digital or analog pins used by another shield at the same time. If you share pins between shields that use the same pin(s) for different functions, you may damage your entire creation, so take care. The suppliers of each shield should provide information showing which pins are used by their shields.

ProtoShields

You can buy a variety of shields online, or make your own using a ProtoShield. A ProtoShield is a blank circuit board that you can use to make your own permanent Arduino shields. ProtoShields come preassembled or in kit form, similar to the one shown in [Figure 7-5](#).

A ProtoShield also makes a good base for a solderless breadboard, because it keeps a small circuit within the physical boundary of your Arduino

A collection of electronic components including a breadboard, integrated circuits, resistors, LEDs, and jumper wires.

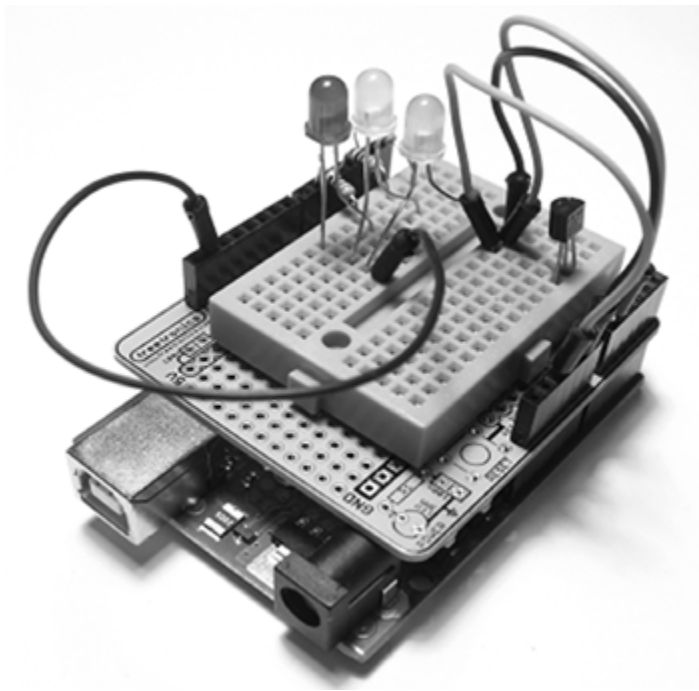


Figure 7-6: An example of small project mounted on a ProtoShield's solderless breadboard

Building custom circuits on a ProtoShield requires some forward planning. You have to design the circuit, make a schematic, and then plan the layout of the components as they will sit on the ProtoShield. Finally, you will solder the completed circuit onto your custom shield, but you should always test it first using a solderless breadboard to ensure that it works. Some ProtoShields come with a PDF schematic file that you can download and print, intended specifically for drawing your project schematic.

Project #21: Creating a Custom Shield

In this project, you'll create a custom shield containing two LEDs and current-limiting resistors. This custom shield will make it easy to experiment with LEDs on digital outputs.

The Hardware

The following hardware is required for this project:

One blank Arduino ProtoShield with stacking headers

Two LEDs of any color

Two 560 Ω resistors

Two 10 k Ω resistors

Two push buttons

Two 100 nF capacitors

The Schematic

The circuit schematic is shown in [Figure 7-7](#).

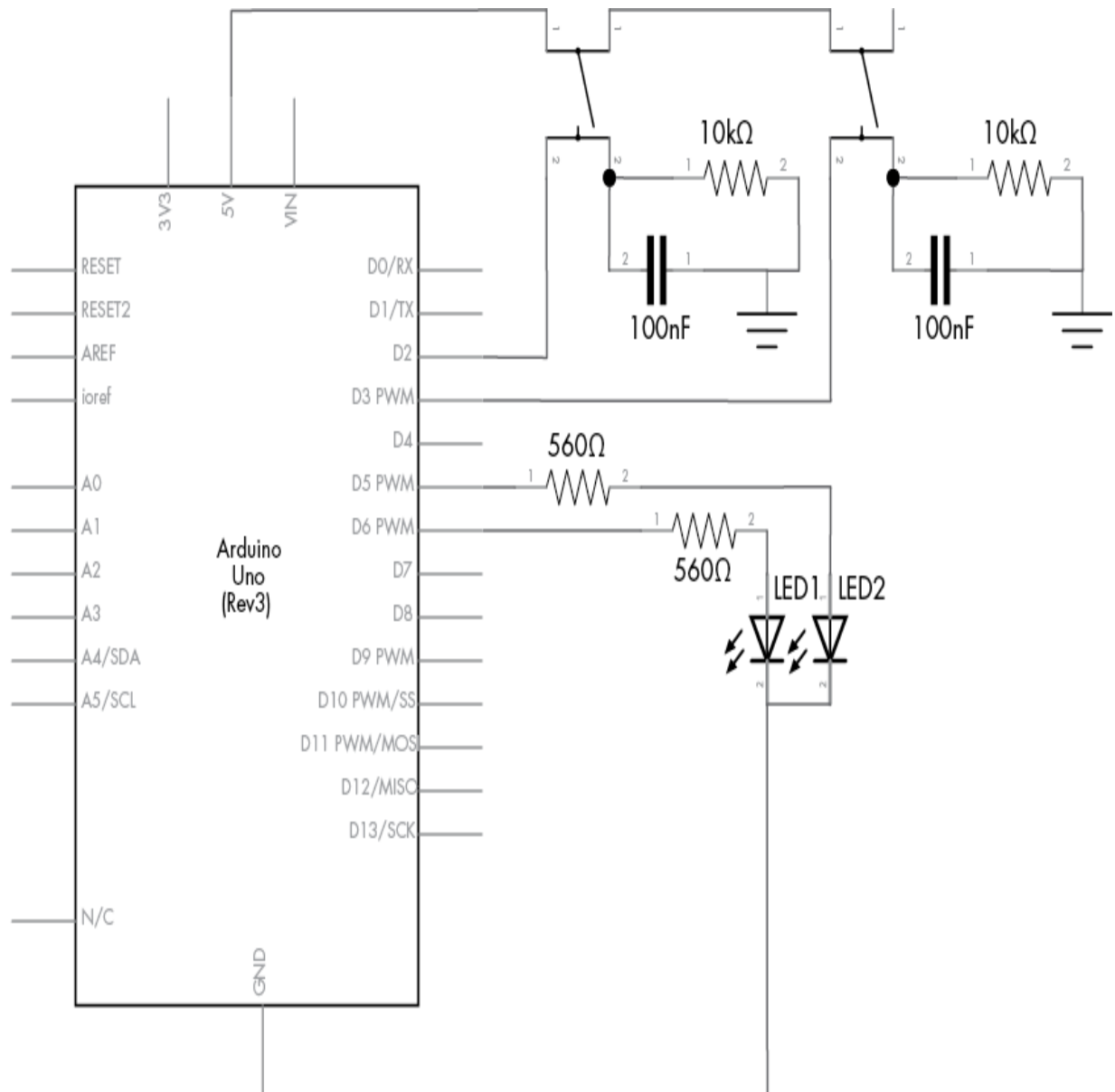


Figure 7-7: Schematic for Project 21

The Layout of the ProtoShield Board

The next step is to learn the layout of the holes on the ProtoShield. The rows and columns of holes on the ProtoShield generally match those of a solderless breadboard. However, each ProtoShield may vary, so take the time to determine how the holes are connected. On the example ProtoShield shown in [Figure 7-8](#), some holes are connected, as shown by the solid lines between the holes, but a lot of holes have been left unconnected. This design gives you a lot of flexibility in how you use your ProtoShield.

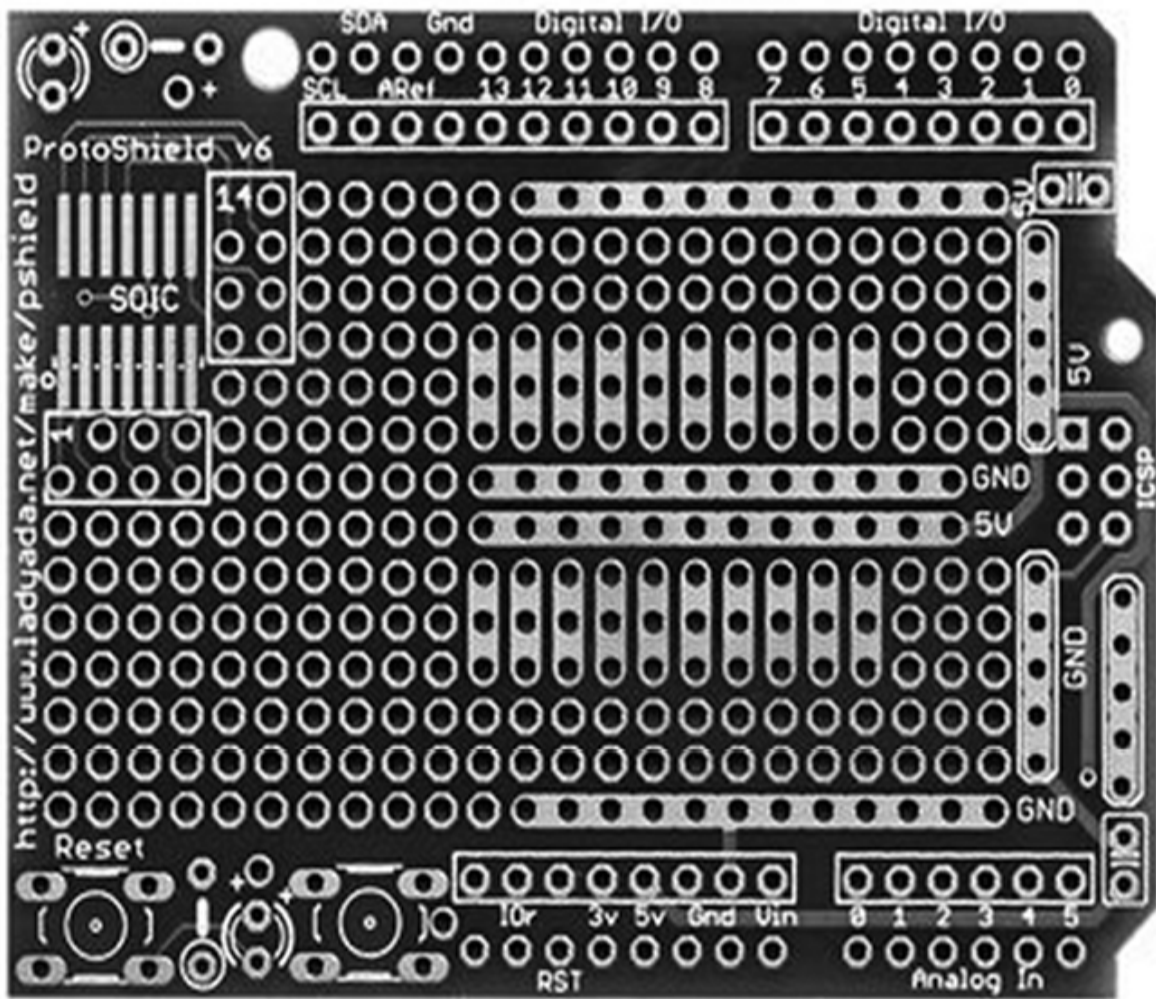


Figure 7-8: A blank ProtoShield shown from above

Note the two groups of holes surrounded by rectangles along the top and bottom of the ProtoShield: this is where we solder the stackable headers that allow the ProtoShield to slot into the Arduino board.

The Design

You need to convert the circuit shown in [Figure 7-7](#) into a physical layout that's suitable for your ProtoShield. A good way to do this is to lay out your circuit using graph paper, as shown in [Figure 7-9](#). You can then mark the connected holes on the graph paper and easily experiment until you find a layout that works for your particular ProtoShield. If you don't have any

graph paper, you can generate and print your own at <http://www.printfreegraphpaper.com/>.

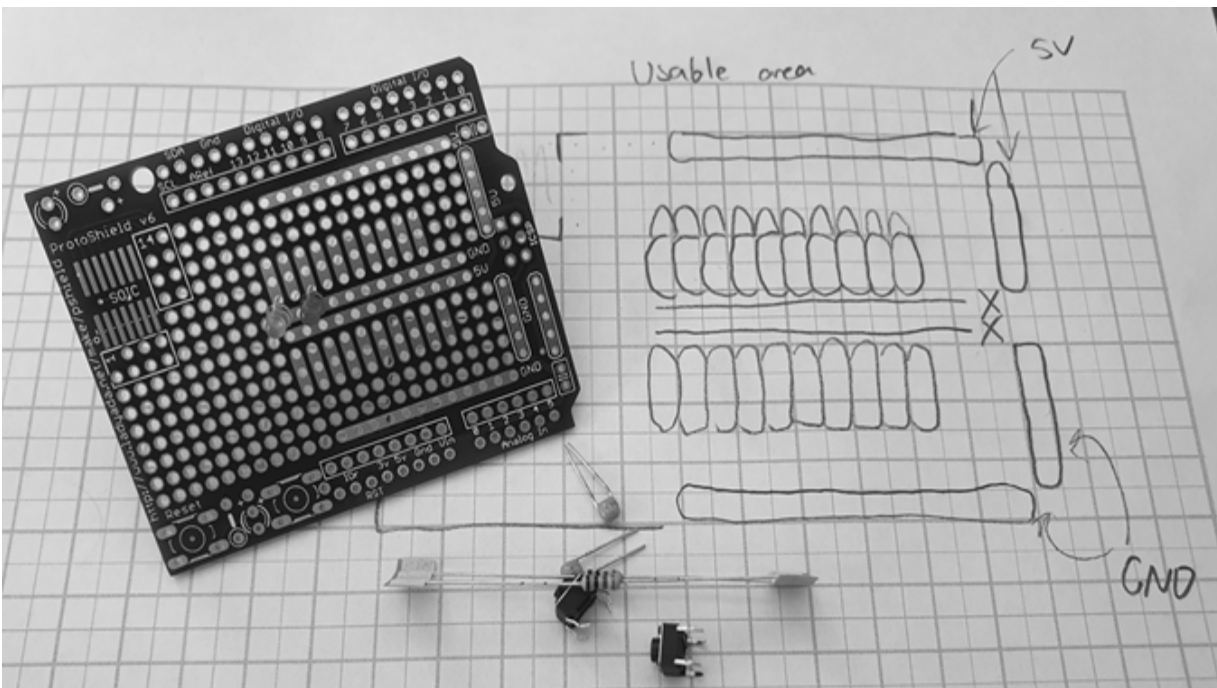


Figure 7-9: Planning our custom shield

After you've drawn a plan for your circuit, test-fit the components into the ProtoShield to make sure that they'll fit and that they aren't too crowded. If the ProtoShield has space for a reset button, always include one, because the shield will block access to your Arduino's RESET button.

Soldering the Components

Once you're satisfied with the layout of the circuit on your ProtoShield and you've tested the circuit to make sure it works, you can solder the components. Using a soldering iron is not that difficult, and you don't need to buy an expensive soldering station for this type of work. A simple iron rated at 25 to 40 watts, like the one shown in [Figure 7-10](#), should do the job.



Figure 7-10: Soldering iron

NOTE

If soldering is new to you, download and read the instructional comic book from <http://mightyohm.com/soldercomic/>.

When soldering the components, you may need to bridge them together with a small amount of solder and wire cutoffs, as shown in [Figure 7-11](#).

Check each solder connection as you go, because mistakes are easier to locate and repair *before* the project is finished. When the time comes to solder the four header sockets or header pins, keep them aligned by using an existing shield to hold the new pins, as shown in [Figure 7-12](#).



Figure 7-11: A solder bridge

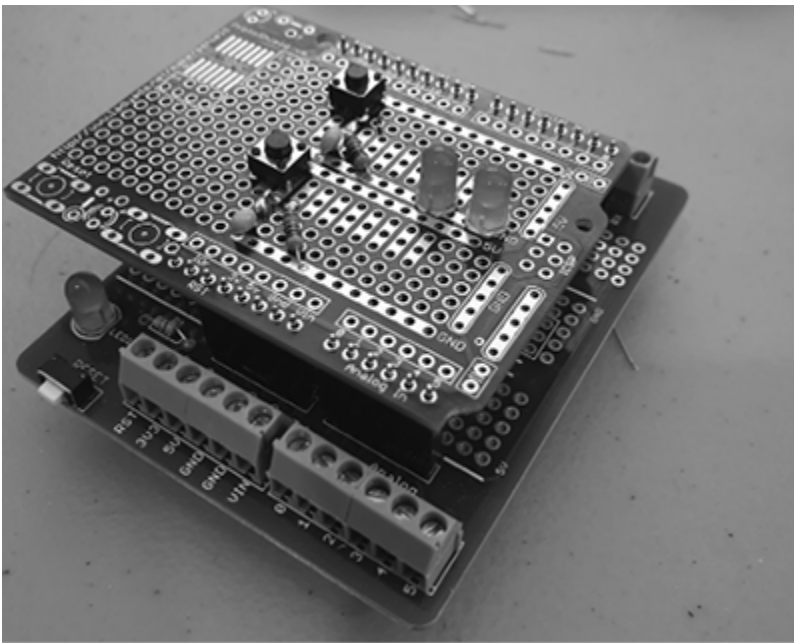


Figure 7-12: Soldering header pins

[Figure 7-13](#) shows the finished product: a custom Arduino shield with two LEDs and two buttons.

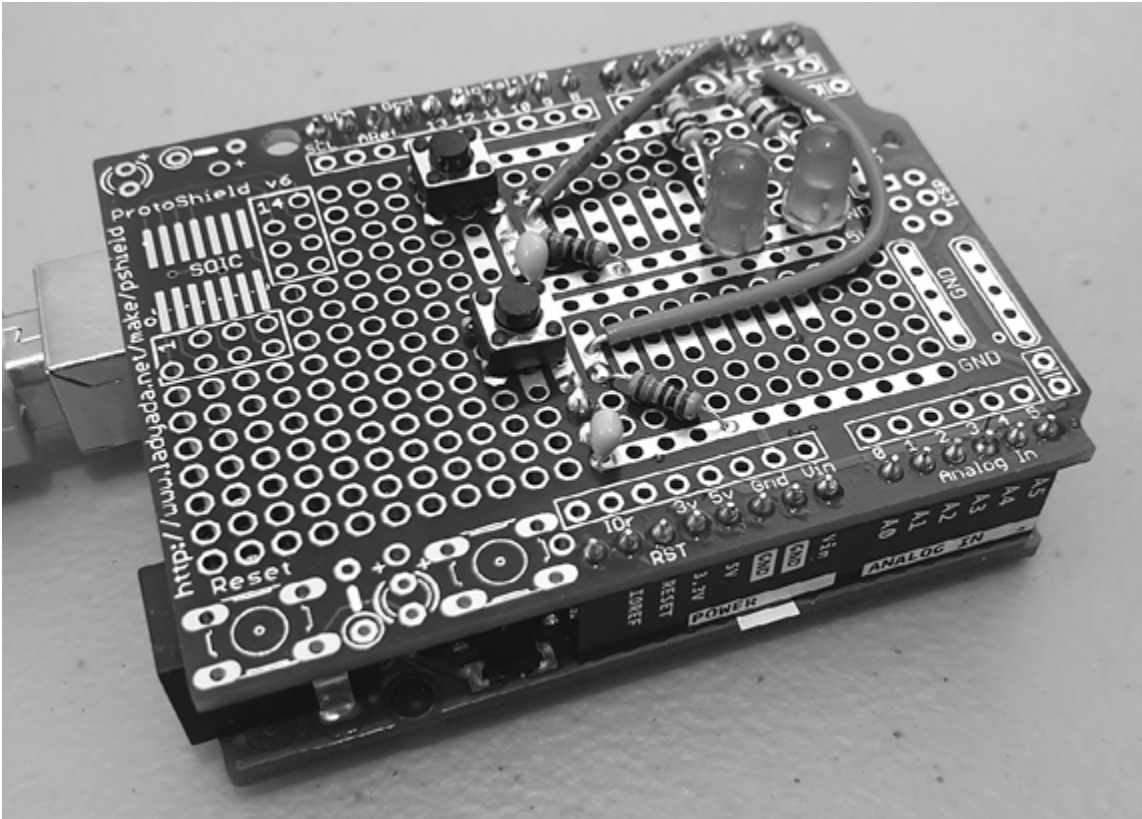


Figure 7-13: The completed custom shield

Testing Your ProtoShield

Before moving on, it's a great idea to test your ProtoShield's buttons and LEDs. The sketch in [Listing 7-1](#) uses the two buttons to turn the LEDs on or off.

```
// Listing 7-1: ProtoShield test
void setup()
{
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(5, OUTPUT);
  pinMode(6, OUTPUT);
}

void loop()
{
  if (digitalRead(2) == HIGH)
  {
    digitalWrite(5, HIGH);
```

```
    digitalWrite(6, HIGH);  
  }  
  if (digitalRead(3) == HIGH)  
  {  
    digitalWrite(5, LOW);  
    digitalWrite(6, LOW);  
  }  
}
```

Listing 7-1: Testing the ProtoShield's buttons and lights

Expanding Sketches with Libraries

Just as an Arduino shield can expand our hardware, a *library* can add useful functions to our sketches. These functions can allow us to use hardware specific to a manufacturer's shield. Anyone can create a library, just as suppliers of various Arduino shields often write their own libraries to match their hardware.

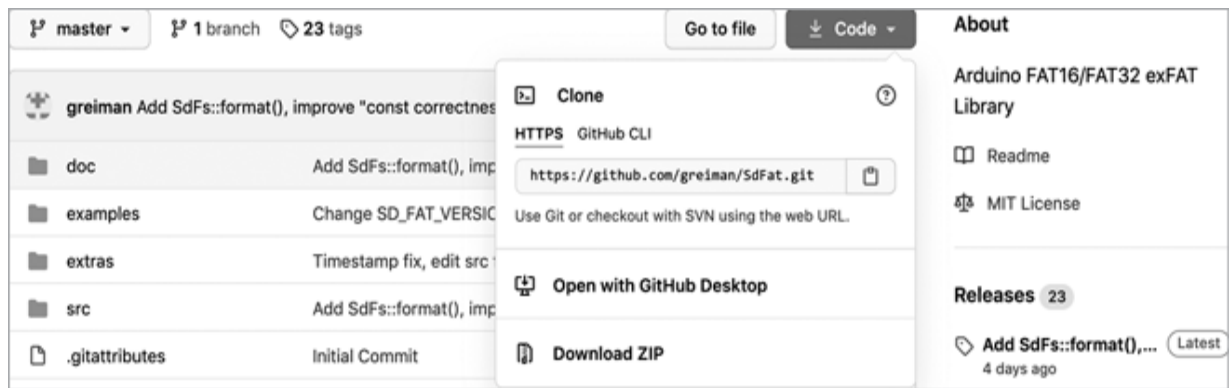
The Arduino IDE already includes a set of preinstalled libraries. To include them in your sketches, choose **Sketch►Include Library**. You should see the collection of preinstalled libraries with names such as Ethernet, LiquidCrystal, Servo, and so on. Many of these names will be self-explanatory. (If a library is required for a project in this book, it will be explained in detail in these pages.)

If you buy a new piece of hardware, you'll generally need to download and install its libraries from the hardware vendor's site or from a provided link. There are two methods for installing an Arduino library: downloading the library in a ZIP file or using the Arduino Library Manager. Let's see how both methods work by walking through a download of the library required by the microSD card shield ([Figure 7-3](#)).

Downloading an Arduino Library as a ZIP File

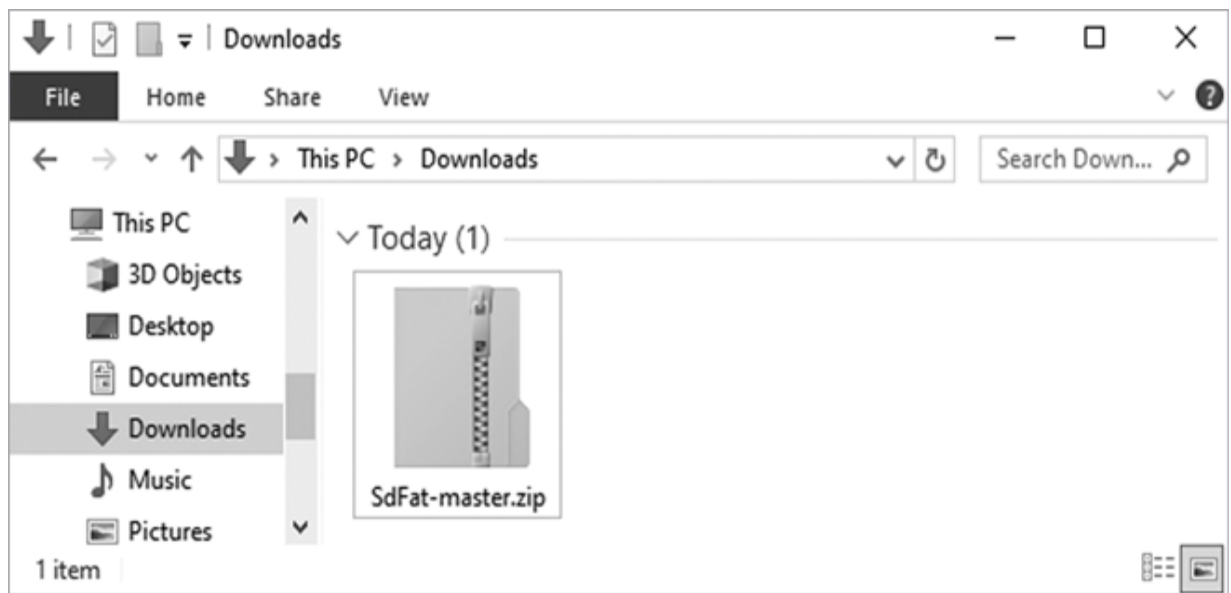
First, let's try downloading and installing a library in ZIP format. You'll download the advanced library used by memory card modules to allow you to read and write data to microSD and SD cards:

Visit <https://github.com/greiman/SdFat/> and click **Code**. Make sure **HTTPS** is selected and then click **Download ZIP**, as demonstrated in [Figure 7-14](#).



[Figure 7-14](#): Library download page

After a moment, the file *SdFat-master.zip* will appear in your *Downloads* folder, as shown in [Figure 7-15](#). If you are using an Apple computer, the ZIP file may be extracted automatically.



[Figure 7-15](#): Downloads folder containing SdFat-master.zip

Open the Arduino IDE and choose **Sketch**►**Include Library**►**Add .ZIP Library**, as shown in [Figure 7-16](#).

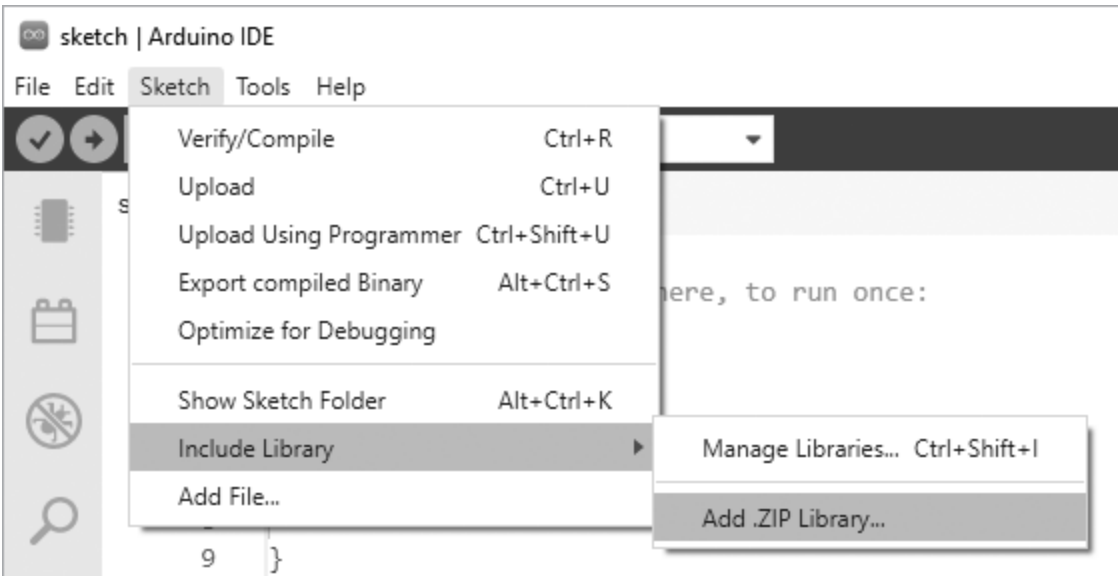


Figure 7-16: Starting the library installation process

You will be presented with a file manager dialog, as shown in [Figure 7-17](#). Navigate to your *Downloads* folder (or wherever you saved the ZIP file) and click **Open**.

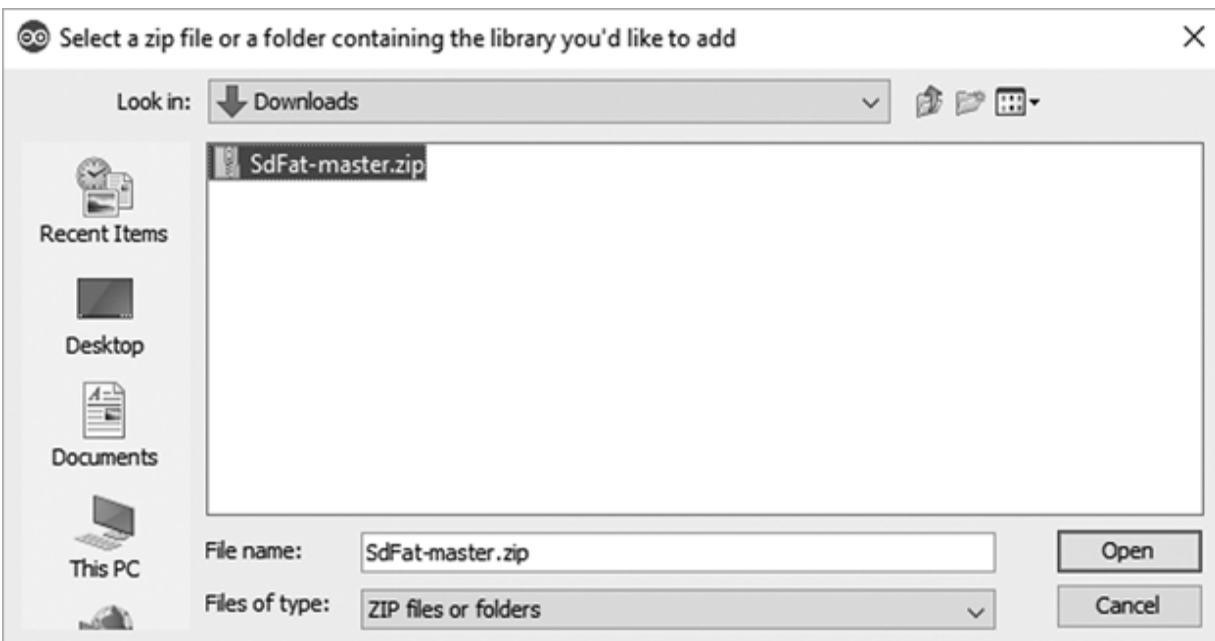


Figure 7-17: Locating the ZIP file

The Arduino IDE will now take care of the library installation. After a few moments, you will be notified that the library has been installed by a

message in the IDE output window, as shown in [Figure 7-18](#).

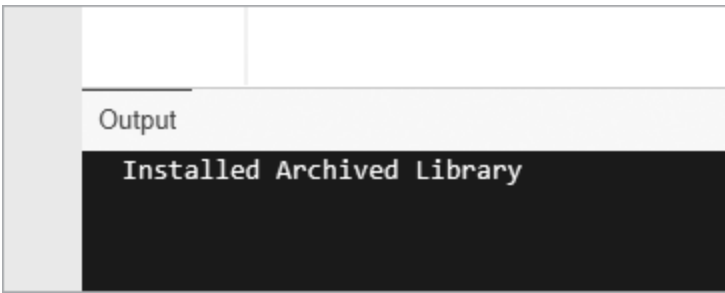


Figure 7-18: Arduino library installation success

You can verify that the SdFat library has been installed and is available by searching through the IDE's Library Manager. To do this, click the Library Manager icon in the vertical group on the left of the IDE, then search using the box at the top, or scroll down until you see your library. For example, in [Figure 7-19](#) you can see that SdFat appears in the Library Manager.

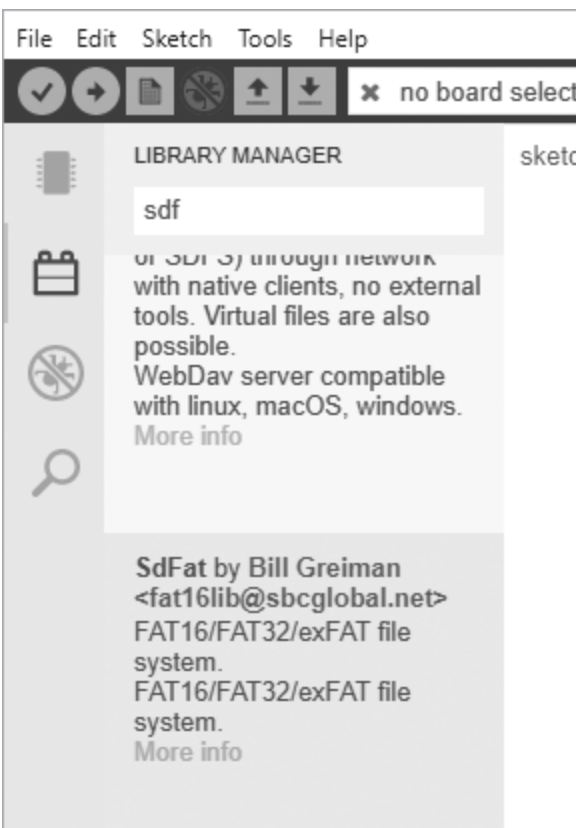


Figure 7-19: Successful installation of the SdFat library

Importing an Arduino Library with Library Manager

The alternative method of installing an Arduino library is via the Arduino IDE's built-in Library Manager. This is a tool for accessing an online repository of libraries that are available for use by the wider public and have been personally approved by the Arduino team, or are just very popular. You will generally access the Library Manager when instructed to by a hardware supplier.

As an example, we'll download the FastLED Arduino library, which is used by a popular type of RGB LED.

To do this, open the Arduino IDE if you've not already done so, then open the Library Manager. Enter **FastLED** in the search box at the top of the manager, as shown in [Figure 7-20](#). As you type, the manager will return libraries that match your search data, and you can see the required library has appeared.

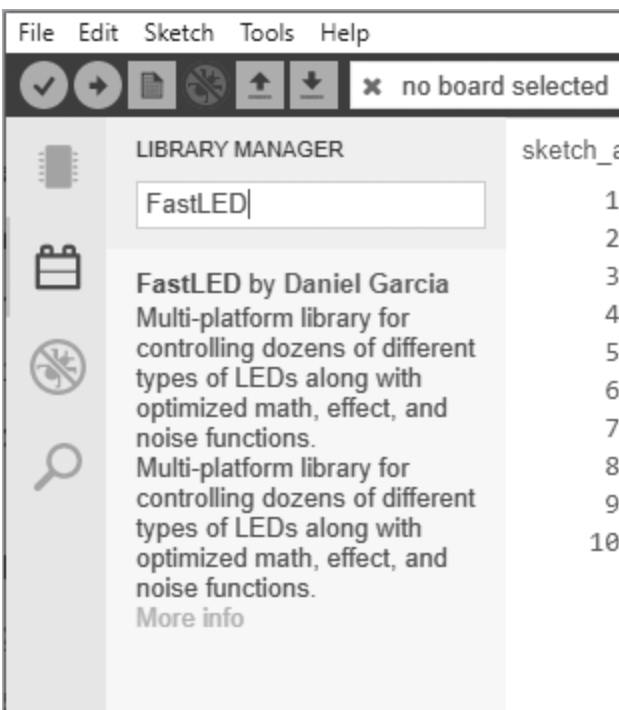


Figure 7-20: Searching the Library Manager

Once the library is found and displayed in the Library Manager, move your mouse cursor over the library description. You may have the option to select a version number. Generally, the latest version is displayed by default, so

you simply need to click **Install** and wait for installation to complete. Installation progress is shown in the output window, as shown in [Figure 7-21](#).

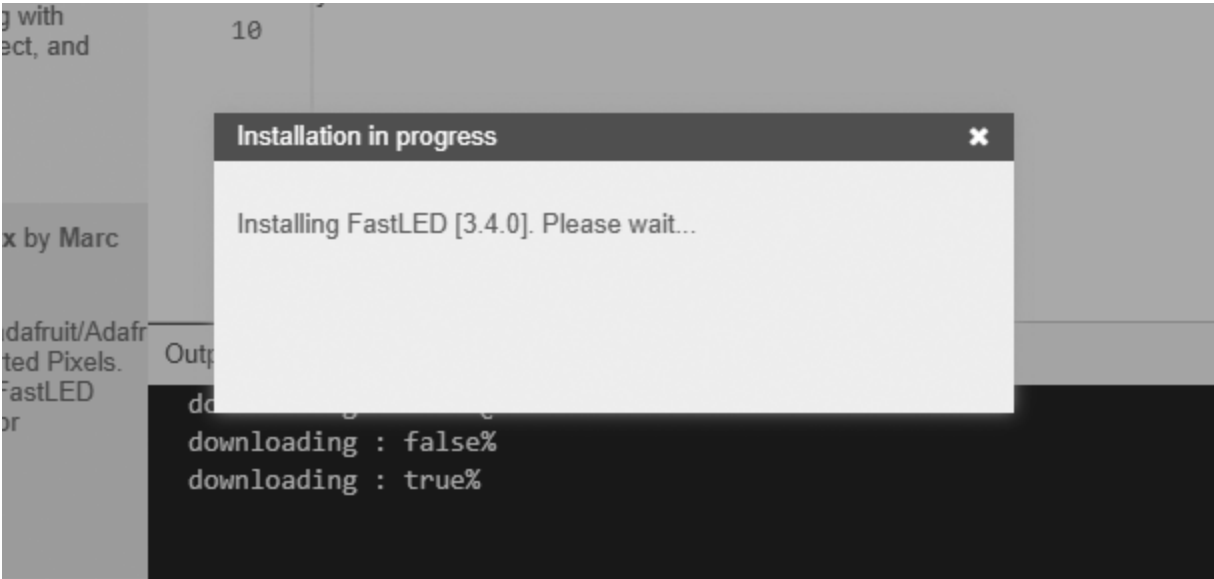


Figure 7-21: Library installation process

You can check that the library has been installed using the method described earlier in this chapter.

SD Memory Cards

By using SD or microSD cards with your Arduino, you can capture data from many sources, such as the TMP36 temperature sensor we used in Chapter 4. You can also use the cards to store web server data or any files that your project might use. To record and store the data you collect, you can use a memory card like the one shown in [Figure 7-22](#).



Figure 7-22: A microSD card with 16GB capacity

Both microSD and SD memory cards are available to work with your Arduino.

NOTE

If your memory card isn't brand new, you'll need to format it before you can use it. To format the card, plug it into a computer and follow your operating system's instructions for formatting memory cards.

Connecting the Card Module

Before you can use the memory card, you'll need to connect six wires from the card reader module to your Arduino. Both card reader types (microSD and SD) will have the same pins, which should be labeled as shown in [Figure 7-23](#).

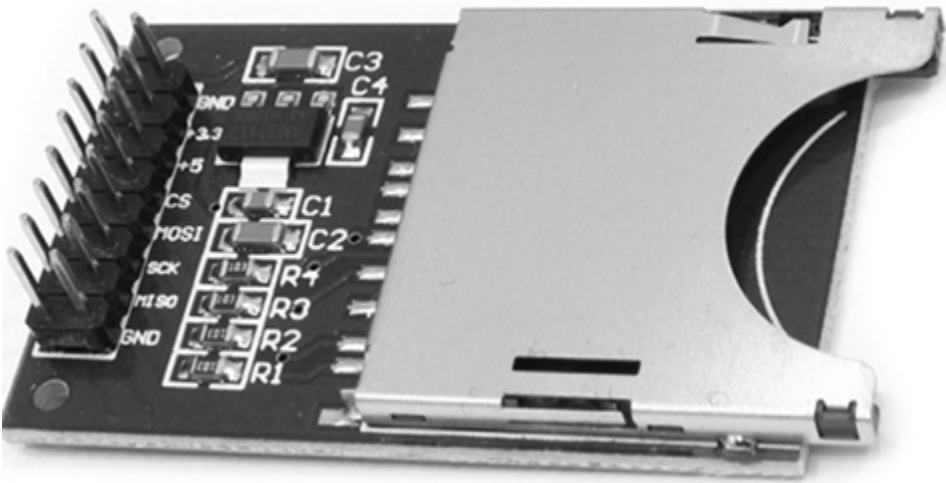


Figure 7-23: SD card module

Make the connections between your Arduino and the card reader as shown in [Table 7-1](#).

Table 7-1: *Connections Between the Card Module and Arduino*

Module pin label	To Arduino pin	Module pin functions
5 V or Vcc	5 V	Power
GND	GND	GND
CS	D10	Chip select
MOSI	D11	Data in from Arduino
MISO	D12	Data out to Arduino
SCK	D13	Clock

Testing Your SD Card

After you have finished connecting the card module to your Arduino—and you have a new or newly formatted card—now is the time to make sure the card is working correctly. To do so, follow these steps:

- Insert the memory card into the card module. Then connect the module to the Arduino and the Arduino to your PC via the USB cable.
- Open the IDE and select **File►Examples►SdFat►SdInfo**. This will load an example sketch.
- Scroll down to line 36 in the sketch and change the value of `const int chipSelect` from 4 to 10, as shown in [Figure 7-24](#). This is necessary as the pin used varies depending on the SD card hardware. Now upload this sketch to your Arduino.

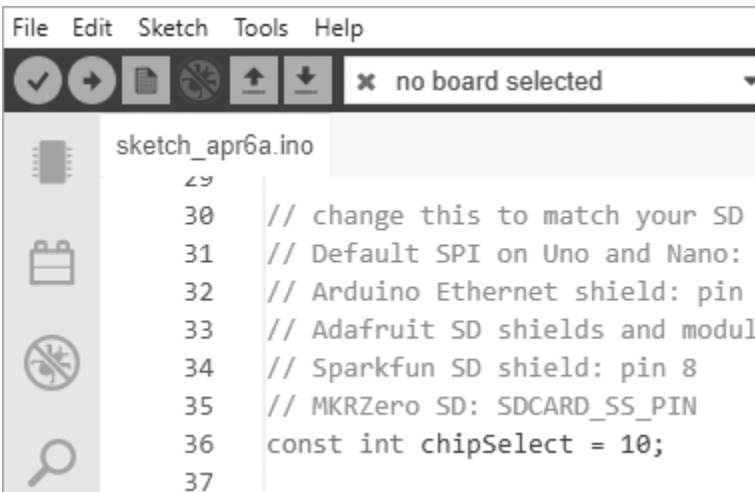


Figure 7-24: Altering the test sketch

- Finally, open the Serial Monitor window, set it to 9,600 baud, press any key on the keyboard, and press ENTER. After a moment, you should see some data about the microSD card, as shown in [Figure 7-25](#).

```
Initializing SD card...Wiring is correct and a card is present.

Card type:          SDHC
Clusters:           490192
Blocks x Cluster:   64
Total Blocks:       31372288

Volume type is:     FAT32
Volume size (Kb):    15686144
Volume size (Mb):    15318
Volume size (Gb):    14.96

Files found on the card (name, date and size in bytes):
SYSTEM~1/           2019-09-24 16:52:30
INDEXE~1            2019-09-24 16:52:30 76
```

Figure 7-25: Results of a successful memory card test

If the test results don't appear in the Serial Monitor, try the following:

Remove the USB cable from your Arduino and remove and reinsert the microSD card.

Make sure that the wiring connections match those in [Table 7-1](#).

Check that the Serial Monitor baud rate is 9,600 and that a regular Arduino Uno-compatible board is being used. The Mega and some other board models have the SPI pins in different locations.

Reformat your memory card.

Failing all else, try a new name-brand memory card.

Finally, before either inserting your memory card or removing it, ensure the entire project is disconnected from the USB and/or power supply.

Project #22: Writing Data to the Memory Card

In this project, you'll use a memory card to store data—specifically, a multiplication table.

The Sketch

To write data to the memory card, connect your shield, insert a microSD card, and then enter and upload the following sketch:

```
// Project 22 - Writing Data to the Memory Card
#include <SD.h>
int b = 0;
void setup()
{
  Serial.begin(9600);
  Serial.println("Initializing SD card...");
  pinMode(10, OUTPUT);

  // check that the memory card exists and is usable
  if (!SD.begin(10))
  {
    Serial.println("Card failed, or not present");
    // stop sketch
    return;
  }
  Serial.println("memory card is ready");
}

void loop()
{
1  // create the file for writing
  File dataFile = SD.open("DATA.TXT", FILE_WRITE);
  // if the file is ready, write to it:
  if (dataFile)
2  {
    for ( int a = 0 ; a < 11 ; a++ )
    {
      dataFile.print(a);
      dataFile.print(" multiplied by two is ");
      b = a * 2;
3    dataFile.println(b, DEC);
    }
4    dataFile.close(); // close the file once the system has
    finished with it
                        // (mandatory)
  }
  // if the file isn't ready, show an error:
  else
  {
    Serial.println("error opening DATA.TXT");
  }
}
```

```
Serial.println("finished");  
do {} while (1);  
}
```

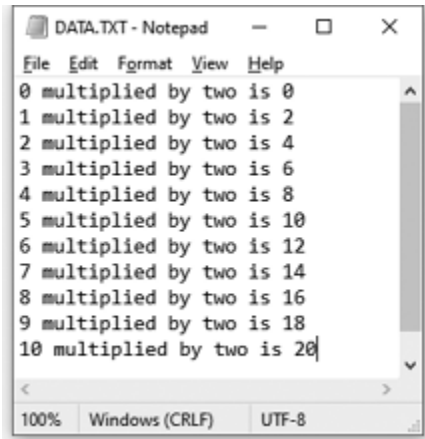


Figure 7-26: Output from Project 22

The sketch creates a text file called *DATA.TXT* on the microSD card, as shown in [Figure 7-26](#).

Let's review the `void loop()` section of the sketch to see how it created the text file. The code in `void loop()` between 1 and 2 creates and opens the file for writing. To write text to the file, we use `dataFile.print()` or `dataFile.println()`.

This code works in the same manner as, for example, `Serial.println()`, so you can write it in the same manner as you would to the Serial Monitor. At 1 we set the name of the created text file, which must be eight characters or less, followed by a dot, and then three characters, such as *DATA.TXT*.

At 3, we use `DEC` as the second parameter. This states that the variable is a decimal number and should be written to the text file as such. If we were writing a `float` variable instead, then we would use a digit for the number of decimal places to write (to a maximum of six).

When we're finished writing data to the file, at 4, we use `dataFile.close()` to close the file for writing. If this step is not followed, the computer will not be able to read the created text file.

Project #23: Creating a Temperature-Logging Device

Now that you know how to record data, let's measure and record the temperature every minute for 8 hours using our memory card setup from Project 22 and the TMP36 temperature sensor circuit introduced in Chapter 4.

The Hardware

The following hardware is required:

One TMP36 temperature sensor

One breadboard

Various connecting wires

Memory card and module

Arduino and USB cable

Insert the microSD card into the shield, and then insert the shield into the Arduino. Connect the left (5 V) pin of the TMP36 to Arduino 5 V, the middle pin to analog, and the right pin to GND.

The Sketch

Enter and upload the following sketch:

```
// Project 23 - Creating a Temperature-Logging Device

#include <SD.h>
float sensor, voltage, celsius;

void setup()
{
  Serial.begin(9600);
  Serial.println("Initializing SD card...");
  pinMode(10, OUTPUT);

  // check that the memory card exists and can be used
  if (!SD.begin(10))
  {
```

```

        Serial.println("Card failed, or not present");
        // stop sketch
        return;
    }
    Serial.println("memory card is ready");
}
void loop()
{
    // create the file for writing
    File dataFile = SD.open("DATA.TXT", FILE_WRITE);
    // if the file is ready, write to it:
    if (dataFile)
    {
        for ( int a = 0 ; a < 481 ; a++ ) // 480 minutes in 8
hours
        {
            sensor = analogRead(0);
            voltage = (sensor * 5000) / 1024; // convert raw sensor
value to
                                                    // millivolts

            voltage = voltage - 500;
            celsius = voltage / 10;
            dataFile.print(" Log: ");
            dataFile.print(a, DEC);
            dataFile.print(" Temperature: ");
            dataFile.print(celsius, 2);
            dataFile.println(" degrees C");
            delay(599900); // wait just under one minute
        }
        dataFile.close(); // mandatory
        Serial.println("Finished!");
        do {} while (1);
    }
}

```

The sketch will take a little more than 8 hours to complete, but you can alter this period by lowering the value in `delay(599900)`.

After the sketch has finished, remove the microSD card from the Arduino, insert it into your computer, and open the log file in a text editor, as shown in [Figure 7-27](#).

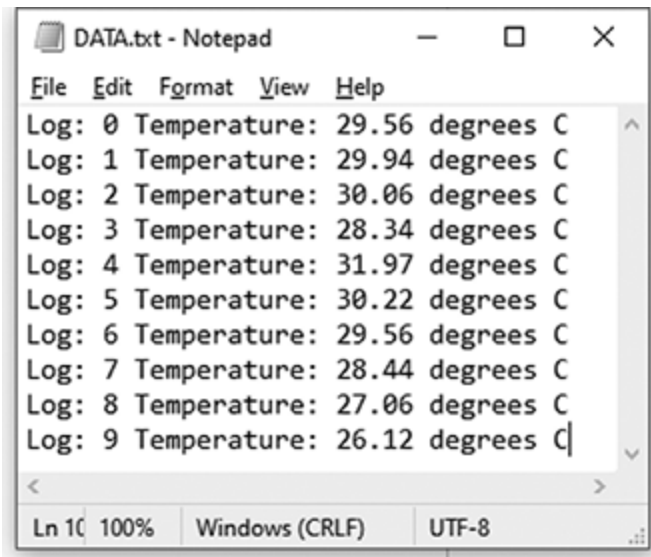


Figure 7-27: Results from Project 23

For more serious analysis of the captured data, delimit the lines of text written to the log file with spaces or colons so that the file can be easily imported into a spreadsheet. For example, you could import the file into OpenOffice Calc or Excel to produce a spreadsheet like the one shown in [Figure 7-28](#).

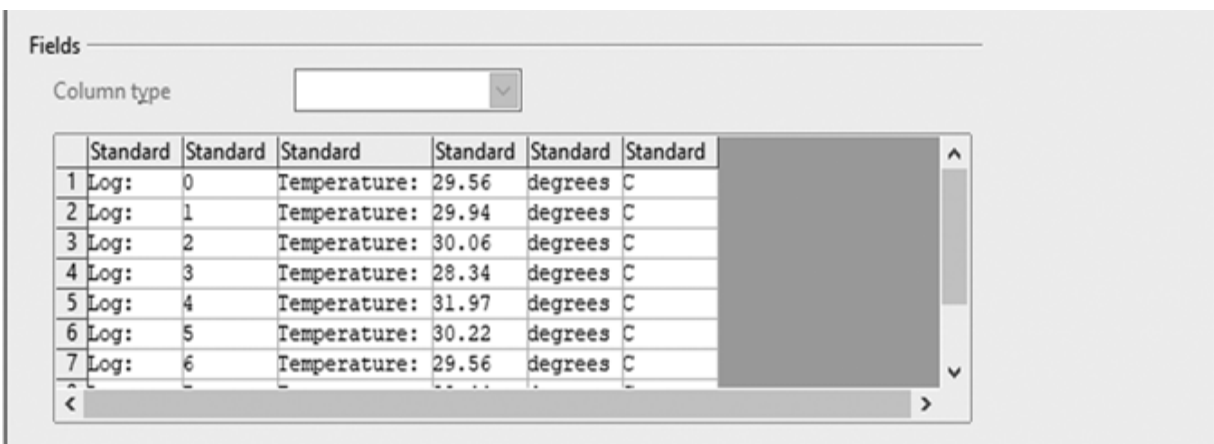
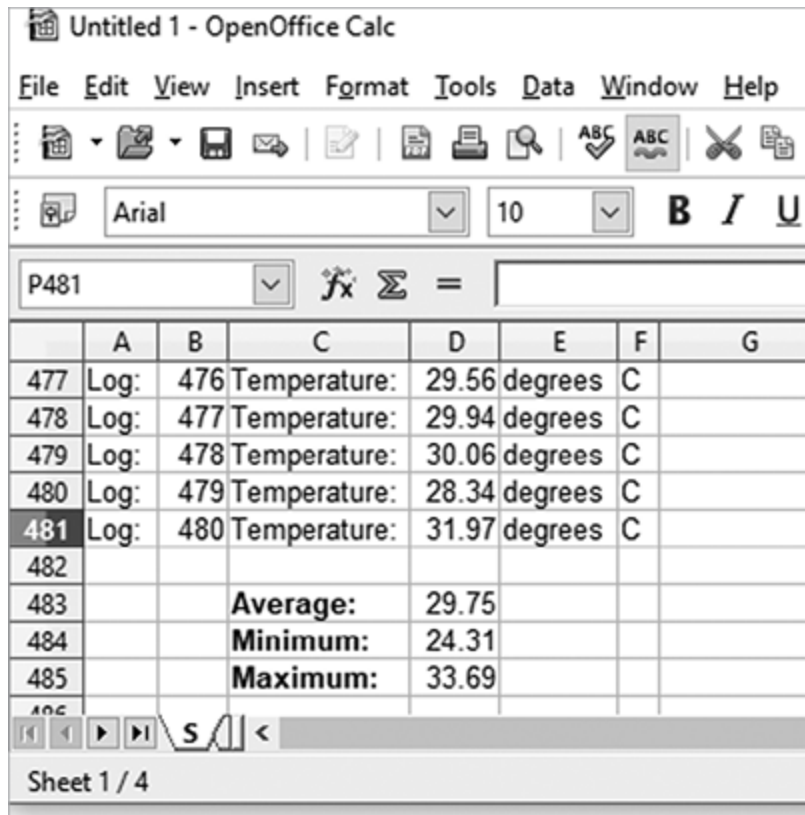


Figure 7-28: Importing the data into a spreadsheet

Then you can easily perform some statistical analysis of the data, as shown in [Figure 7-29](#).

The temperature examples can be hacked to suit your own data analysis projects. You can use these same concepts to record any form of data that

can be generated by an Arduino system.



The screenshot shows the OpenOffice Calc interface with a spreadsheet titled 'Untitled 1 - OpenOffice Calc'. The spreadsheet contains a table of temperature data. The table has columns A through G. The data is as follows:

	A	B	C	D	E	F	G
477	Log:	476	Temperature:	29.56	degrees	C	
478	Log:	477	Temperature:	29.94	degrees	C	
479	Log:	478	Temperature:	30.06	degrees	C	
480	Log:	479	Temperature:	28.34	degrees	C	
481	Log:	480	Temperature:	31.97	degrees	C	
482							
483			Average:	29.75			
484			Minimum:	24.31			
485			Maximum:	33.69			

The status bar at the bottom indicates 'Sheet 1 / 4'.

Figure 7-29: Temperature analysis

Timing Applications with millis() and micros()

Each time the Arduino starts running a sketch, it also records the passage of time using milliseconds and microseconds. A millisecond is one thousandth of a second (0.001), and a microsecond is one millionth of a second (0.000001). You can use these values to measure the passage of time when running sketches.

The following functions will access the time values stored in an unsigned long variable:

```
unsigned long a,b;  
a = micros();  
b = millis();
```

Due to the limitations of the unsigned long variable type (which stores only positive values), the value will reset to 0 after reaching 4,294,967,295, allowing for around 50 days of counting using `millis()` and 70 minutes using `micros()`. Furthermore, due to the limitations of the Arduino's microprocessor, `micros()` values are always a multiple of four.

Let's use these values to see how long it takes for the Arduino to turn a digital pin from low to high and vice versa. To do this, we'll read `micros()` before and after a `digitalWrite()` function, find the difference, and display it in the Serial Monitor. The only required hardware is your Arduino and cable.

Enter and upload the sketch shown in [*Listing 7-2*](#).

```
// Listing 7-2

unsigned long starting, finished, elapsed;

void setup()
{
  Serial.begin(9600);
  pinMode(3, OUTPUT);
  digitalWrite(3, LOW);
}

void loop()
{
1  starting = micros();
   digitalWrite(3, HIGH);
2  finished = micros();
3  elapsed = finished - starting;
   Serial.print("LOW to HIGH: ");
   Serial.print(elapsed);
   Serial.println(" microseconds");
   delay(1000);

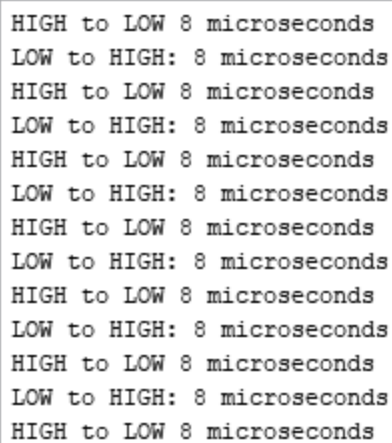
4  starting = micros();
   digitalWrite(3, LOW);
   finished = micros();
   elapsed = finished - starting;
   Serial.print("HIGH to LOW: ");
   Serial.print(elapsed);
   Serial.println(" microseconds");
}
```

```
    delay(1000);  
}
```

Listing 7-2: Timing digital pin state change with `micros()`

The sketch takes readings of `micros()` before and after the `digitalWrite(HIGH)` function call, at 1 and 2, and then it calculates the difference and displays it in the Serial Monitor at 3. This is repeated for the opposite function at 4.

Now open the Serial Monitor to view the results, shown in [Figure 7-30](#).



```
HIGH to LOW 8 microseconds  
LOW to HIGH: 8 microseconds  
HIGH to LOW 8 microseconds  
LOW to HIGH: 8 microseconds  
HIGH to LOW 8 microseconds  
LOW to HIGH: 8 microseconds  
HIGH to LOW 8 microseconds  
LOW to HIGH: 8 microseconds  
HIGH to LOW 8 microseconds  
LOW to HIGH: 8 microseconds  
HIGH to LOW 8 microseconds  
LOW to HIGH: 8 microseconds  
HIGH to LOW 8 microseconds
```

Figure 7-30: Output from [Listing 7-2](#)

Because the resolution is 4 microseconds, if the value is 8 microseconds, we know that the duration is greater than 4 and less than or equal to 8.

Project #24: Creating a Stopwatch

Now that we can measure the elapsed time between two events, we can create a simple stopwatch using an Arduino. Our stopwatch will use two buttons: one to start or reset the count and one to stop counting and show the elapsed time. The sketch will continually check each button's status. When the start button is pressed, a `millis()` value will be stored, and when the stop button is pressed, a new `millis()` value will be stored. The custom function `displayResult()` will convert the elapsed time from milliseconds

into hours, minutes, and seconds. Finally, the time will be displayed in the Serial Monitor.

The Hardware

Use the ProtoShield as described earlier in this chapter and the following additional hardware:

One breadboard

Two push buttons (S1 and S2)

Two 10 k Ω resistors (R1 and R2)

Various connecting wires

Arduino and USB cable

The Schematic

The circuit schematic is shown in [Figure 7-31](#).

NOTE

You will use this circuit for the next project, so don't pull it apart when you're finished!

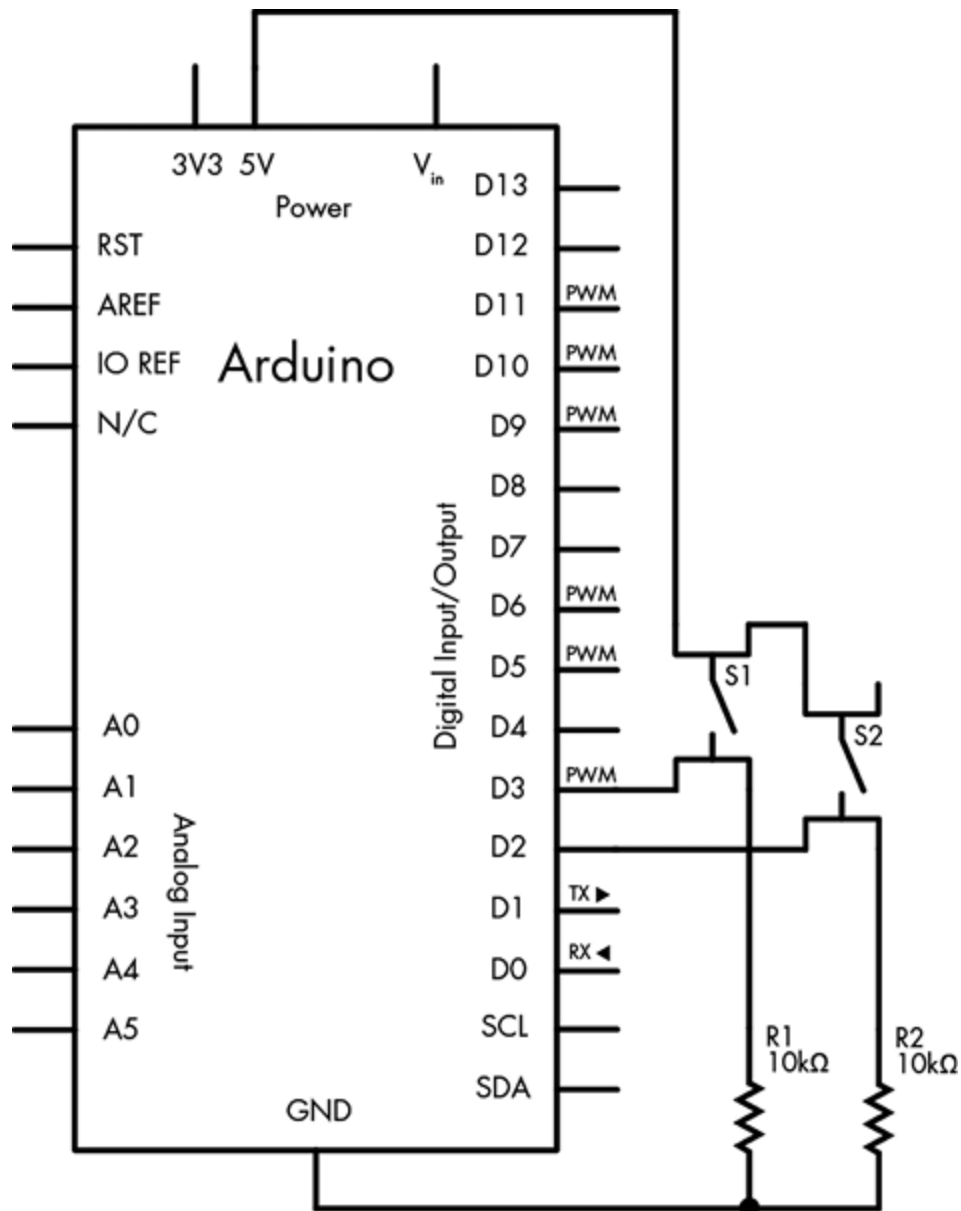


Figure 7-31: Schematic for Project 24

The Sketch

Enter and upload this sketch:

```
// Project 24 - Creating a Stopwatch

unsigned long starting, finished, elapsed;

void setup()
{
```



```

    Serial.begin(9600);
1   pinMode(2, INPUT); // the start button
    pinMode(3, INPUT); // the stop button
    Serial.println("Press 1 for Start/reset, 2 for elapsed
time");
}

void displayResult()
{
    float h, m, s, ms;
    unsigned long over;

2   elapsed = finished - starting;

    h    = int(elapsed / 3600000);
    over = elapsed % 3600000;
    m    = int(over / 60000);
    over = over % 60000;
    s    = int(over / 1000);
    ms   = over % 1000;

    Serial.print("Raw elapsed time: ");
    Serial.println(elapsed);
    Serial.print("Elapsed time: ");
    Serial.print(h, 0);
    Serial.print("h ");
    Serial.print(m, 0);
    Serial.print("m ");
    Serial.print(s, 0);
    Serial.print("s ");
    Serial.print(ms, 0);
    Serial.println("ms");
    Serial.println();
}

void loop()
{
3   if (digitalRead(2) == HIGH)
    {
        starting = millis();
        delay(200); // for debounce
        Serial.println("Started...");
    }
4   if (digitalRead(3) == HIGH)
    {
        finished = millis();

```

```
        delay(200); // for debounce
        displayResult();
    }
}
```

The basis for our stopwatch is simple. At 1, we set up the digital input pins for the start and stop buttons. At 3, if the start button is pressed, then the Arduino notes the value for `millis()` that we use to calculate the elapsed time once the stop button is pressed at 4. After the stop button is pressed, the elapsed time is calculated in the function `displayResult()` at 2 and shown in the Serial Monitor window.

You should see results like those in [Figure 7-32](#) in the Serial Monitor.

```
Started...
Press 1 for Start/reset, 2 for elapsed time
Raw elapsed time: 4368
Elapsed time: 0h 0m 4s 368ms

Raw elapsed time: 13662
Elapsed time: 0h 0m 13s 662ms

Raw elapsed time: 392366
Elapsed time: 0h 6m 32s 366ms

Raw elapsed time: 5052371
Elapsed time: 1h 24m 12s 371ms

Raw elapsed time: 5077826
Elapsed time: 1h 24m 37s 826ms
```

[Figure 7-32](#): Output from Project 24

Interrupts

An *interrupt* in the Arduino world is basically a signal that allows a function to be called at any time within a sketch—for example, when a digital input pin’s state changes or a timer event is triggered. Interrupts are perfect for calling a function to interrupt the normal operation of a sketch, such as when a button is pressed. This type of function is often referred to as an *interrupt handler*.

When an interrupt is triggered, the normal operation and running of your program is halted temporarily as the interrupt function is called and executed. Then, when the interrupt function exits, whatever was happening in the program continues exactly where it left off.

Interrupt functions should be short and simple. They should exit quickly, and keep in mind that if the interrupt function does something that the main loop is already doing, then the interrupt function is going to temporarily override the main loop's activity before the main loop resumes. For example, if the main loop is regularly sending *Hello* out the serial port and the interrupt function sends --- when it is triggered, then you could see any of these come out the serial port: *H----ello*, *He----llo*, *Hel----lo*, *Hell----o*, or *Hello----*.

The Arduino Uno offers two interrupts that are available using digital pins 2 and 3. When properly configured, the Arduino will monitor the voltage applied to the pins. When the voltage changes in a certain defined way (when a button is pressed, for example), an interrupt is triggered, causing a corresponding function to run—maybe something that sends “Stop Pressing Me!”

Interrupt Modes

One of four changes (or *modes*) can trigger an interrupt:

LOW No current is applied to the interrupt pin.

CHANGE The current changes, either between on and off or between off and on.

RISING The current changes from off to on at 5 V.

FALLING The current changes from on at 5 V to off.

For example, to detect when a button attached to an interrupt pin has been pressed, you could use the **RISING** mode. Or, for example, if you had an electric trip wire running around your garden (connected between 5 V and the interrupt pin), then you could use the **FALLING** mode to detect when the wire has been tripped and broken.

NOTE

The `delay()` and `Serial.available()` functions will not work within a function that has been called by an interrupt.

Configuring Interrupts

To configure interrupts, use the following in `void setup()`:

```
attachInterrupt(0, function, mode);  
attachInterrupt(1, function, mode);
```

Here, `0` is for digital pin 2, `1` is for digital pin 3, *function* is the name of the function to call when the interrupt is triggered, and *mode* is one of the four modes that triggers the interrupt.

Activating or Deactivating Interrupts

Sometimes you won't want to use the interrupts within a sketch. You can deactivate a single interrupt using:

```
detachInterrupt(digitalPinToInterrupt(pin))
```

where *pin* is the digital pin number used. Or you can deactivate them all using the following:

```
noInterrupts(); // deactivate interrupts
```

And then reactivate them with this:

```
interrupts(); // reactivate interrupts
```

Interrupts work quickly and they are very sensitive. These qualities make them useful for time-critical applications or for “emergency stop” buttons on projects.

Project #25: Using Interrupts

We'll use the circuit from Project 24 to demonstrate the use of interrupts. Our example will blink the built-in LED every 500 milliseconds, during which time both interrupt pins will be monitored. When the button on interrupt 0 is pressed, the value for `micros()` will be displayed in the Serial Monitor, and when the button on interrupt 1 is pressed, the value for `millis()` will be displayed.

The Sketch

Enter and upload the following sketch:

```
// Project 25 - Using Interrupts

#define LED 13
void setup()
{
  Serial.begin(9600);
  pinMode(13, OUTPUT);
  attachInterrupt(0, displayMicros, RISING);
  attachInterrupt(1, displayMillis, RISING);
}
1 void displayMicros()
{
  Serial.write("micros() = ");
  Serial.println(micros());
}

2 void displayMillis()
{
  Serial.write("millis() = ");
  Serial.println(millis());
}

3 void loop()
{
  digitalWrite(LED, HIGH);
  delay(500);
  digitalWrite(LED, LOW);
  delay(500);
}
```

This sketch will blink the onboard LED as shown in `void loop()` at 3. When interrupt 0 is triggered, the function `displayMicros()` at 1 will be

called; when interrupt 1 is triggered, the function `displayMillis()` at 2 will be called. After either function has finished, the sketch resumes running the code in `void loop()`.

Open the Serial Monitor window and press the two buttons to view the values for `millis()` and `micros()`, as shown in [Figure 7-33](#).

```
millis() = 3769
micros() = 4595052
millis() = 5483
micros() = 6328740
millis() = 7155
micros() = 7673876
millis() = 8339
micros() = 8874420
millis() = 9511
micros() = 10024460
millis() = 10618
micros() = 11339048
millis() = 11942
micros() = 12413384
millis() = 12880
micros() = 13292308
millis() = 13809
micros() = 14309052
```

Figure 7-33: Output from Project 25

Looking Ahead

This chapter has given you more tools and options that you can adapt to create and improve your own projects. In future chapters, we will work with more Arduino shields, use interrupts for timing projects, and use the memory card module in other data-logging applications.

8

LED NUMERIC DISPLAYS AND MATRICES

In this chapter, you will

- Use MAX7219-based numeric LED displays

- Build your own digital stopwatch timer

- Use MAX7219-based LED matrix modules

- Build a scrolling text LED display

Although LED numeric displays (such as those found in contemporary digital alarm clocks) may not be on the bleeding edge of display technology, they are easy to read and—more importantly—easy to use with our Arduino boards.

You learned how to use one- and two-digit LED numeric displays in Chapter 6. However, using more than two digits at a time can become messy—there’s a lot more wiring, more control ICs, and so on to take care of. Fortunately, there’s a popular IC that can control up to 64 LEDs (eight digits of a numeric display) with only three control wires from our Arduino: the MAX7219 LED driver IC from Maxim Integrated.

The MAX7219 is available in both a through-hole package type, which means it has metal legs that can fit into a circuit board or a solderless breadboard ([Figure 8-1](#)), and a surface-mount package type ([Figure 8-2](#)).

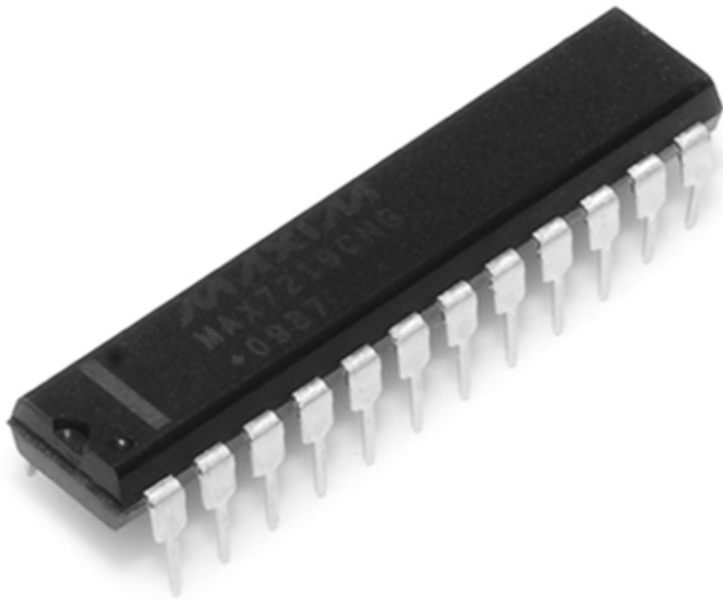


Figure 8-1: The MAX7219 in a through-hole package type

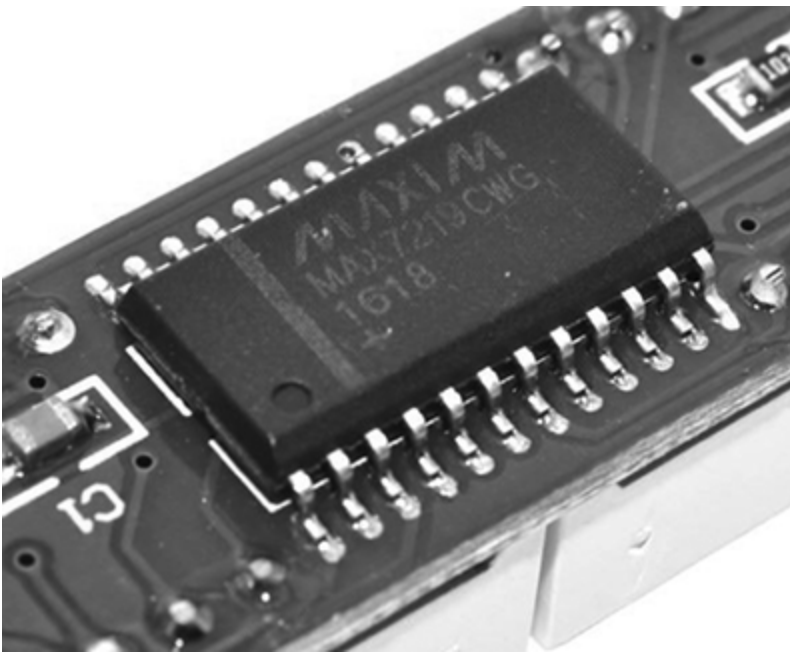


Figure 8-2: The MAX7219 in a surface-mount package type

In this chapter, you'll learn how to use the MAX7219 to control up to eight numeric LED digits. You'll also learn how to use the MAX7219 to control interesting LED matrix modules that allow for scrolling text displays.

LED Numeric Displays

LED numeric displays that use the MAX7219 come in many shapes and sizes, usually with four to eight digits fitted to the module. For our examples, we're using an eight-digit module, which is easily available and great value for the money (see [Figure 8-3](#)).



Figure 8-3: The eight-digit LED module

These modules have the surface-mount version of the MAX7219, shown in [Figure 8-2](#), on the back. The modules usually include some inline header pins to allow for attaching control wires. If you haven't already done so, solder these to your module, as shown in [Figure 8-4](#).



Figure 8-4: Inline header pins connected to an eight-digit LED module

Before you can use the numeric display, you'll need to connect five wires to both the display and the Arduino. This is easily done by connecting male-

to-female jumper wires to the header pins that you soldered to the board. Make the connections as shown in [Table 8-1](#).

Table 8-1: *Connections Between the Display Module and Arduino*

Module pin label	Arduino pin	Module pin function
Vcc	5V	Power (+)
GND	GND	Power (–) or ground
DIN	D12	Data in
CS	D10	Chip select
CLK	D11	Clock signal

Installing the Library

There are several Arduino libraries for the MAX7219. These libraries vary according to the configuration of the display module used. We will use the LedControl library. You will need to download the library ZIP file from <https://github.com/wayoda/LedControl/>. Click **Clone or Download** and then **Download ZIP**, as shown in [Figure 8-5](#).

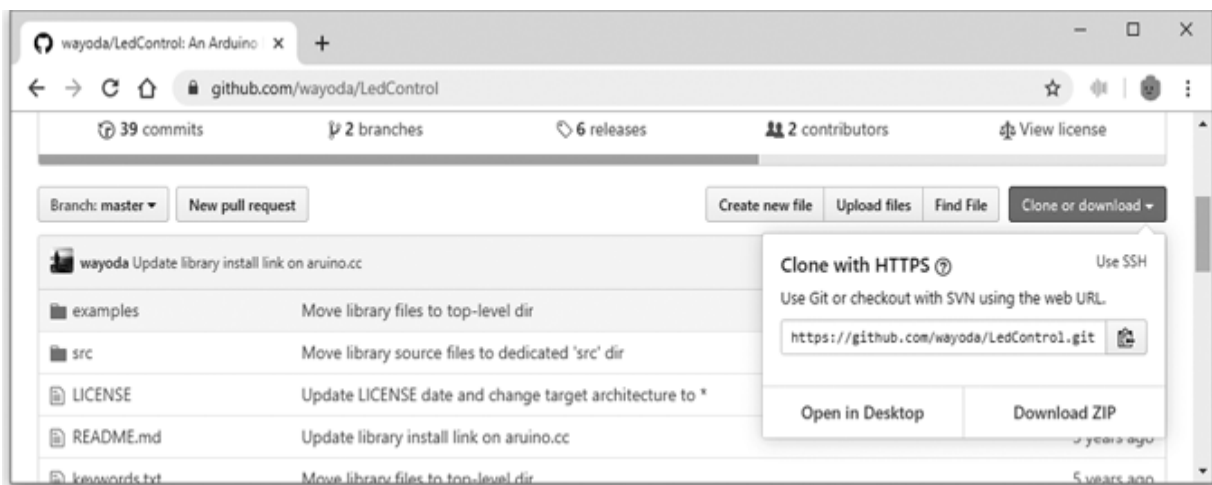


Figure 8-5: *Downloading the LedControl library*

Once you have the ZIP file, install it as described in Chapter 7. Next, to use the display module, we will first examine a demonstration sketch that uses the required functions. Enter and upload the basic sketch shown in [Listing 8-1](#).

```
// Listing 8-1
1 #include "LedControl.h" // need the library
  LedControl lc = LedControl(12, 11, 10, 1);
  void setup()
  {
2    lc.shutdown(0, false); // enable display
      lc.setIntensity(0, 3); // set brightness
      lc.clearDisplay(0); // clear screen
  }

  void loop()
  {
    // numbers with decimal point
    for (int a = 0; a < 8; a++)
    {
3      lc.setDigit(0, a, a, true);
        delay(500);
        lc.clearDisplay(0) ; // clear screen
    }
    // dashes
    for (int a = 0; a < 8; a++)
    {
4      lc.setChar(0, a, '-', false);
        delay(500);
        lc.clearDisplay(0) ; // clear screen
    }
    // numbers without decimal point
    for (int a = 0; a < 8; a++)
    {
      lc.setDigit(0, a, a, false);
      delay(500);
      lc.clearDisplay(0) ; // clear screen
    }
5    // display "abcdef"
      lc.setDigit(0, 5, 10, false);
      lc.setDigit(0, 4, 11, false);
      lc.setDigit(0, 3, 12, false);
      lc.setDigit(0, 2, 13, false);
      lc.setDigit(0, 1, 14, false);
      lc.setDigit(0, 0, 15, false);
      delay(500);
      lc.clearDisplay(0) ; // clear screen
  }
}
```

Listing 8-1: Display module demonstration sketch

Let's take a look at how the sketch in [Listing 8-1](#) works. At 1, we include the necessary code to load the library for the display module. The `LedControl()` function has four parameters:

```
LedControl lc = LedControl(12, 11, 10, 1);
```

The first three say which digital pins are connected (see [Table 8-1](#)), and the fourth parameter is the number of display modules connected to the Arduino—in this case one. (You can daisy-chain more than one module.)

At 2, we have three functions that control aspects of the display. The first one turns the display on or off:

```
lc.shutdown(0, false);
```

The first parameter is the display. We use 0 because only one display is connected. If you have connected multiple displays, the second is display 1, the third is display 2, and so on. The second parameter specifies whether the display is on or off: `false` for on, `true` for off.

The second function is used to set the brightness of the LEDs in the display:

```
lc.setIntensity(0, 3);
```

The first parameter is the display number. The second is the brightness, which can be between 0 and 15 inclusive.

The third function simply turns all the LEDs off:

```
lc.clearDisplay(0);
```

This is great for clearing previously displayed data.

At 3, we display a digit on the screen using `setDigit()`:

```
lc.setDigit(0, a, b, true);
```

The first parameter is the display number. The second is the physical position of the digit on the display; for an eight-digit display, this value ranges from 7 (the leftmost digit) to 0 (the rightmost digit). The third

parameter is the actual number to display (0 to 9). If you use 10 to 16, you can display the letters A to F, as we've done at 5. Finally, the fourth parameter controls the decimal point: `true` for on and `false` for off.

You can also write the characters A to F, H, L, P, dash, period, and underscore using `setChar()`, as at 4:

```
lc.setChar(0, a, '-', false);
```

The parameters are the same, except you enclose the character with single quotes.

Now that we've been through all the commands for showing numbers and characters on the display, let's put them into action.

Project #26: Digital Stopwatch

You learned about timing in Project 24 in Chapter 7, and you've just learned how to use a display module, so now you can combine these concepts to create a digital stopwatch. While not accurate to Olympic timing levels, this is a fun and useful project. Your stopwatch will be able to display milliseconds, seconds, minutes, and up to nine hours.

You will need to connect the ProtoShield (or equivalent circuit) as described in Chapter 7 and to the numeric display used earlier in this chapter. Then just upload the following sketch:

```
// Project 26 - Digital Stopwatch
#include "LedControl.h" // need the library
LedControl lc = LedControl(12, 11, 10, 1);
unsigned long starting, finished, elapsed;
void setup()
{
  pinMode(2, INPUT); // the start button
  pinMode(3, INPUT); // the stop button
  lc.shutdown(0, false); // enable display
  lc.setIntensity(0, 3); // set brightness
  lc.clearDisplay(0); // clear screen
  starting = millis();
}
```

```

1 void displayResultLED()
  {
    float h, m, s, ms;
    int m1, m2, s1, s2, ms1, ms2, ms3;
    unsigned long over;
    finished = millis();
    elapsed = finished - starting;

2   h = int(elapsed / 3600000);
    over = elapsed % 3600000;
    m = int(over / 60000);
    over = over % 60000;
    s = int(over / 1000);
    ms = over % 1000;

3   // display hours
    lc.setDigit(0, 7, h, true);

    // display minutes
    m1 = m / 10;
    m2 = int(m) % 10;
    lc.setDigit(0, 6, m1, false);
    lc.setDigit(0, 5, m2, true);

    // display seconds
    s1 = s / 10;
    s2 = int(s) % 10;
    lc.setDigit(0, 4, s1, false);
    lc.setDigit(0, 3, s2, true);

    // display milliseconds (1/100 s)
    ms1 = int(ms / 100);
    ms2 = (int((ms / 10)) % 10);
    ms3 = int(ms) % 10;
    lc.setDigit(0, 2, ms1, false);
    lc.setDigit(0, 1, ms2, false);
    lc.setDigit(0, 0, ms3, false);
  }

void loop()
{
4   if (digitalRead(2) == HIGH) // reset count
    {
      starting = millis();
      delay(200);                // for debounce
    }
  }

```

```

    }
5   if (digitalRead(3) == HIGH) // display count for five
    seconds then resume
    {
        finished = millis();
        delay(5000);           // for debounce
    }
    displayResultLED();
}

```

A moment after the sketch has been uploaded, the display will start counting up, as shown in [Figure 8-6](#).

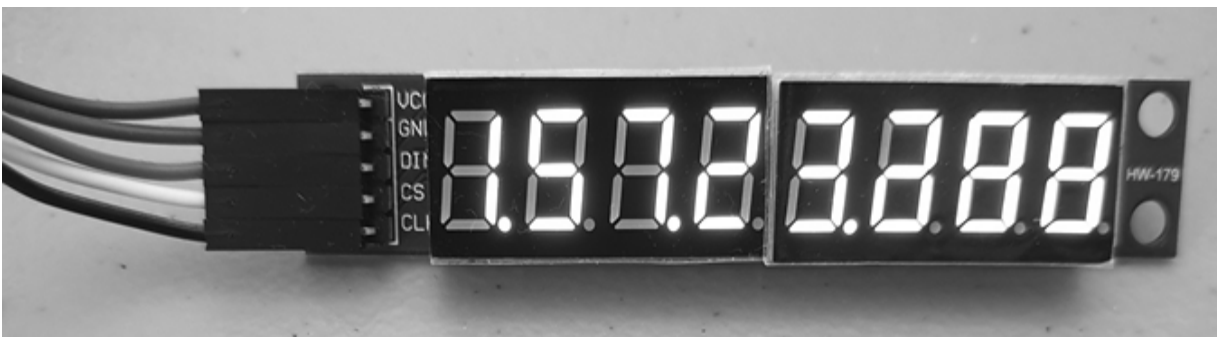


Figure 8-6: The stopwatch at work

Just as we did in Project 24, in this sketch we use `millis()` to track the elapsed time. We have put the time calculation and display in the function `void displayResultLED()` 1.

At 2, you can see how the elapsed time in milliseconds is broken down into hours, minutes, seconds, and milliseconds. Then, each digit of the display from left to right is filled with the corresponding time values, starting with hours 3. The stopwatch controls are simple: when the user presses the button connected to digital input 2, the counter is reset to zero by making the starting time equal to the current value returned by `millis()` 4. When the button connected to digital input 3 is pressed 5, the display is frozen; this functionality is ideal for taking a lap reading. Note, however, that the counting continues and the display resumes after about five seconds.

This project can easily be changed to display data in a simpler format—such as hours, minutes, and seconds—or to be used for longer periods, such

as up to 24 hours. But for now, let's move on to a more complex project involving LED matrix display boards.

Project #27: Using LED Matrix Modules

The MAX7219 can control up to 64 LEDs. These displayed numbers in the last project. Here, we'll use modules that arrange the LEDs in an 8×8 matrix form that is ideal for more interesting applications, such as displaying fixed or scrolling text.

LED matrix modules are generally sold either as individual units or in sets of four; both are shown in [Figure 8-7](#).

You may also see these advertised as kits; however, the cost savings is negligible, so save time with the preassembled versions. The LED displays fit onto the socketed pins on the module, as shown in [Figure 8-8](#), allowing you to change colors easily.

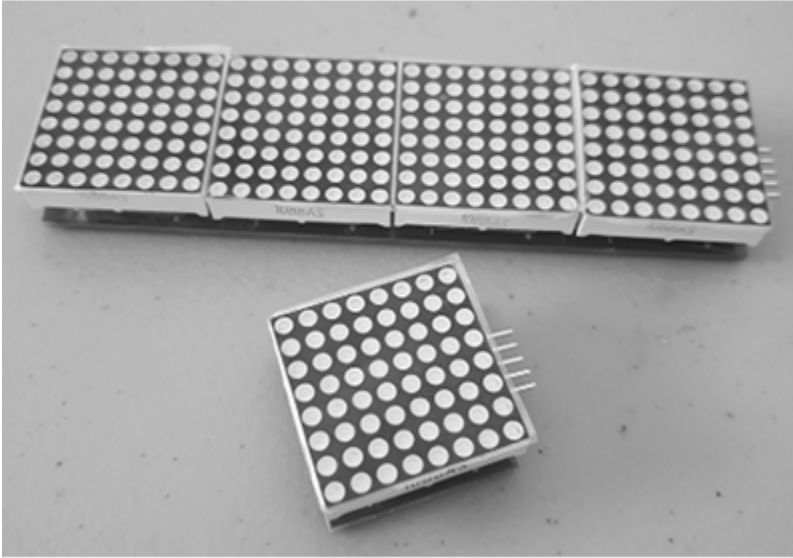


Figure 8-7: LED matrix modules

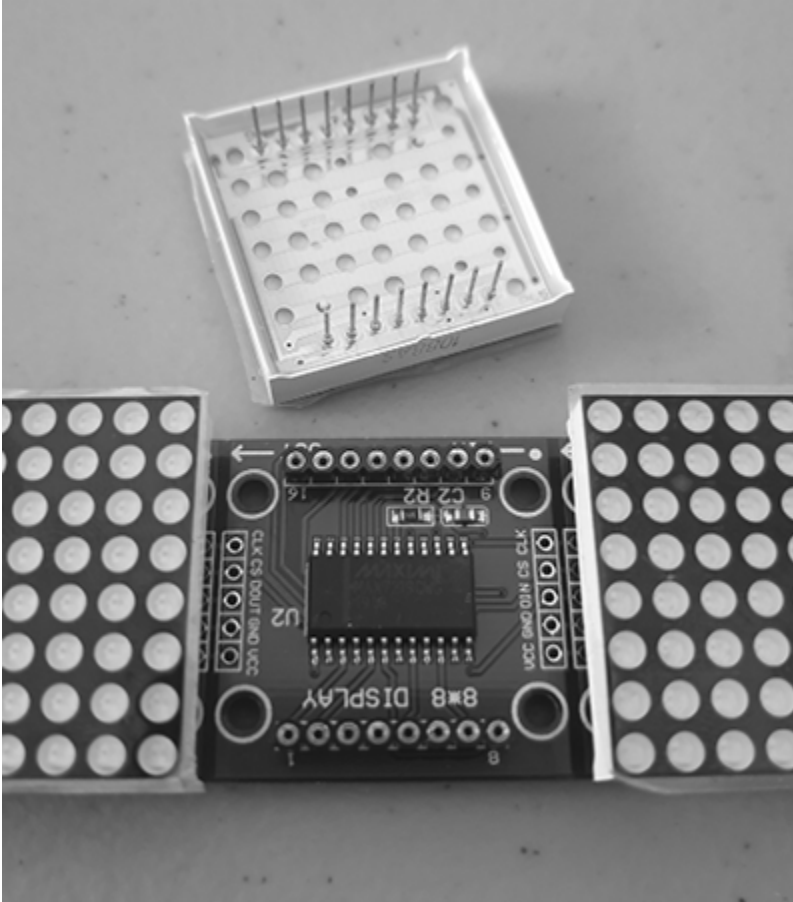


Figure 8-8: A removable LED matrix

Take care when inserting the LED matrix into the module, as some LED matrices have pins that get bent easily. Experience has shown that you still need to solder inline header pins to the matrix modules. However, these pins are generally included with the module and fit neatly, as shown in [Figure 8-9](#).

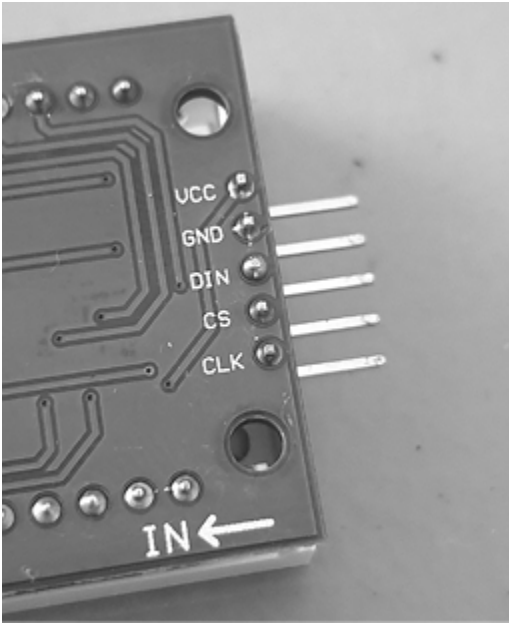


Figure 8-9: Inline header pins connected to a matrix module

Once again, before you can use the matrix modules, you'll need to connect five wires to both the module and the Arduino, just as you did with the numeric display. Make the connections as shown in [Table 8-2](#).

Table 8-2: *Connections Between the Matrix Module and Arduino*

Module pin label	Arduino pin	Module pin function
Vcc	5V	Power (+)
GND	GND	Power (–) or ground
DIN	D11	Data in
CS	D9	Chip select
CLK	D13	Clock signal

Installing the Library

You'll use a different library for these modules than for the MAX7219. To get the library, visit <https://github.com/bartoszbielawski/LEDMatrixDriver/> and click **Clone or Download**, then **Download ZIP**, as shown in [Figure 8-10](#).

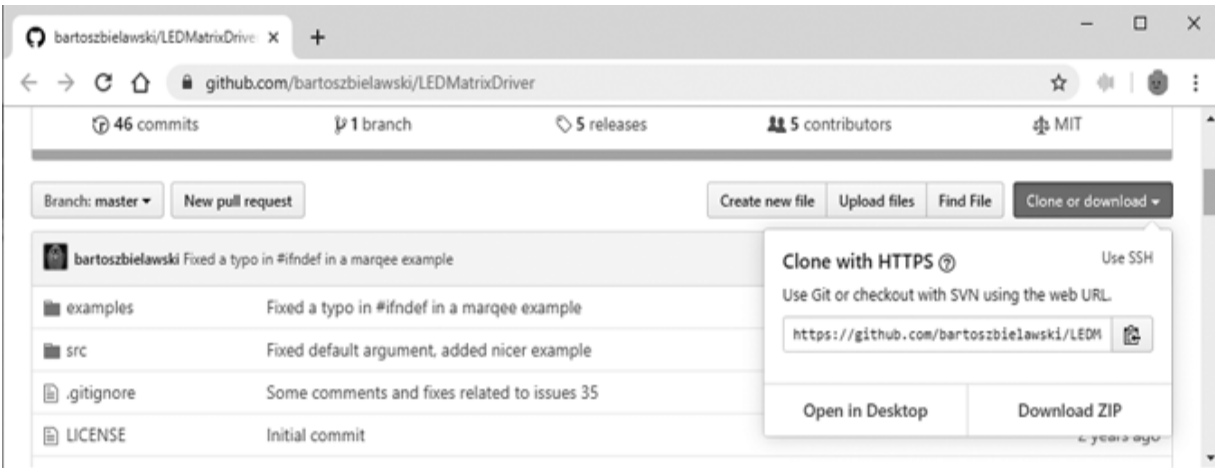


Figure 8-10: Downloading the LEDMatrixDriver library

After you have downloaded the ZIP file, install it as described in Chapter 7. Enter and upload the sketch that follows. (At this point, I'd like to remind you that all the code in this book can be downloaded from <https://nostarch.com/arduino-workshop-2nd-edition/>.)

NOTE

If you're using a single matrix module, change the value of `const int LEDMATRIX_SEGMENTS = 4` on line 4 from 4 to 1.

```
// Project 27 - Using LED Matrix Modules
1 #include <LEDMatrixDriver.hpp>
  const uint8_t LEDMATRIX_CS_PIN = 9;
  // Number of matrix modules you are connecting
  const int LEDMATRIX_SEGMENTS = 4;
  const int LEDMATRIX_WIDTH = LEDMATRIX_SEGMENTS * 8;
  LEDMatrixDriver lmd(L EDMATRIX_SEGMENTS, LEDMATRIX_CS_PIN);
  // Text to display
2 char text[] = "** LED MATRIX DEMO! ** (1234567890) ++
  \"ABCDEFGH IJKLMNOPQRSTUVWXYZ\" ++ <$%/?'.@,> --";
  // scroll speed (smaller = faster)
3 const int ANIM_DELAY = 30;
  void setup() {
4 // init the display
  lmd.setEnabled(true);
  lmd.setIntensity(2); // 0 = low, 10 = high
```

```

}
int x = 0, y = 0; // start top left
// font definition
5 byte font[95][8] = { {0, 0, 0, 0, 0, 0, 0, 0}, // SPACE
    {0x10, 0x18, 0x18, 0x18, 0x18, 0x00, 0x18, 0x18}, // EXCL
    {0x28, 0x28, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00}, // QUOT
    {0x00, 0x0a, 0x7f, 0x14, 0x28, 0xfe, 0x50, 0x00}, // #
    {0x10, 0x38, 0x54, 0x70, 0x1c, 0x54, 0x38, 0x10}, // $
    {0x00, 0x60, 0x66, 0x08, 0x10, 0x66, 0x06, 0x00}, // %
    {0, 0, 0, 0, 0, 0, 0, 0}, // &
    {0x00, 0x10, 0x18, 0x18, 0x08, 0x00, 0x00, 0x00}, // '
    {0x02, 0x04, 0x08, 0x08, 0x08, 0x08, 0x08, 0x04}, // (
    {0x40, 0x20, 0x10, 0x10, 0x10, 0x10, 0x10, 0x20}, // )
    {0x00, 0x10, 0x54, 0x38, 0x10, 0x38, 0x54, 0x10}, // *
    {0x00, 0x08, 0x08, 0x08, 0x7f, 0x08, 0x08, 0x08}, // +
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x18, 0x08}, // COMMA
    {0x00, 0x00, 0x00, 0x00, 0x7e, 0x00, 0x00, 0x00}, // -
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x06}, // DOT
    {0x00, 0x04, 0x04, 0x08, 0x10, 0x20, 0x40, 0x40}, // /
    {0x00, 0x38, 0x44, 0x4c, 0x54, 0x64, 0x44, 0x38}, // 0
    {0x04, 0x0c, 0x14, 0x24, 0x04, 0x04, 0x04, 0x04}, // 1
    {0x00, 0x30, 0x48, 0x04, 0x04, 0x38, 0x40, 0x7c}, // 2
    {0x00, 0x38, 0x04, 0x04, 0x18, 0x04, 0x44, 0x38}, // 3
    {0x00, 0x04, 0x0c, 0x14, 0x24, 0x7e, 0x04, 0x04}, // 4
    {0x00, 0x7c, 0x40, 0x40, 0x78, 0x04, 0x04, 0x38}, // 5
    {0x00, 0x38, 0x40, 0x40, 0x78, 0x44, 0x44, 0x38}, // 6
    {0x00, 0x7c, 0x04, 0x04, 0x08, 0x08, 0x10, 0x10}, // 7
    {0x00, 0x3c, 0x44, 0x44, 0x38, 0x44, 0x44, 0x78}, // 8
    {0x00, 0x38, 0x44, 0x44, 0x3c, 0x04, 0x04, 0x78}, // 9
    {0x00, 0x18, 0x18, 0x00, 0x00, 0x18, 0x18, 0x00}, // :
    {0x00, 0x18, 0x18, 0x00, 0x00, 0x18, 0x18, 0x08}, // ;
    {0x00, 0x10, 0x20, 0x40, 0x80, 0x40, 0x20, 0x10}, // <
    {0x00, 0x00, 0x7e, 0x00, 0x00, 0xfc, 0x00, 0x00}, // =
    {0x00, 0x08, 0x04, 0x02, 0x01, 0x02, 0x04, 0x08}, // >
    {0x00, 0x38, 0x44, 0x04, 0x08, 0x10, 0x00, 0x10}, // ?
    {0x00, 0x30, 0x48, 0xba, 0xba, 0x84, 0x78, 0x00}, // @
    {0x00, 0x1c, 0x22, 0x42, 0x42, 0x7e, 0x42, 0x42}, // A
    {0x00, 0x78, 0x44, 0x44, 0x78, 0x44, 0x44, 0x7c}, // B
    {0x00, 0x3c, 0x44, 0x40, 0x40, 0x40, 0x44, 0x7c}, // C
    {0x00, 0x7c, 0x42, 0x42, 0x42, 0x42, 0x44, 0x78}, // D
    {0x00, 0x78, 0x40, 0x40, 0x70, 0x40, 0x40, 0x7c}, // E
    {0x00, 0x7c, 0x40, 0x40, 0x78, 0x40, 0x40, 0x40}, // F
    {0x00, 0x3c, 0x40, 0x40, 0x5c, 0x44, 0x44, 0x78}, // G
    {0x00, 0x42, 0x42, 0x42, 0x7e, 0x42, 0x42, 0x42}, // H
    {0x00, 0x7c, 0x10, 0x10, 0x10, 0x10, 0x10, 0x7e}, // I
    {0x00, 0x7e, 0x02, 0x02, 0x02, 0x02, 0x04, 0x38}, // J
    {0x00, 0x44, 0x48, 0x50, 0x60, 0x50, 0x48, 0x44}, // K

```

```

    {0x00, 0x40, 0x40, 0x40, 0x40, 0x40, 0x40, 0x7c}, // L
    {0x00, 0x82, 0xc6, 0xaa, 0x92, 0x82, 0x82, 0x82}, // M
    {0x00, 0x42, 0x42, 0x62, 0x52, 0x4a, 0x46, 0x42}, // N
    {0x00, 0x3c, 0x42, 0x42, 0x42, 0x42, 0x44, 0x38}, // O
    {0x00, 0x78, 0x44, 0x44, 0x48, 0x70, 0x40, 0x40}, // P
    {0x00, 0x3c, 0x42, 0x42, 0x52, 0x4a, 0x44, 0x3a}, // Q
    {0x00, 0x78, 0x44, 0x44, 0x78, 0x50, 0x48, 0x44}, // R
    {0x00, 0x38, 0x40, 0x40, 0x38, 0x04, 0x04, 0x78}, // S
    {0x00, 0x7e, 0x90, 0x10, 0x10, 0x10, 0x10, 0x10}, // T
    {0x00, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x3e}, // U
    {0x00, 0x42, 0x42, 0x42, 0x42, 0x44, 0x28, 0x10}, // V
    {0x80, 0x82, 0x82, 0x92, 0x92, 0x92, 0x94, 0x78}, // W
    {0x00, 0x42, 0x42, 0x24, 0x18, 0x24, 0x42, 0x42}, // X
    {0x00, 0x44, 0x44, 0x28, 0x10, 0x10, 0x10, 0x10}, // Y
    {0x00, 0x7c, 0x04, 0x08, 0x7c, 0x20, 0x40, 0xfe}, // Z
};

```

```

6 void drawString(char* text, int len, int x, int y)
{
    for ( int idx = 0; idx < len; idx ++ )
    {
        int c = text[idx] - 32;

        // stop if char is outside visible area
        if ( x + idx * 8 > LEDMATRIX_WIDTH )
            return;

        // only draw if char is visible
        if ( 8 + x + idx * 8 > 0 )
            drawSprite( font[c], x + idx * 8, y, 8, 8 );
    }
}

```

```

7 void scrollText()
{
    int len = strlen(text);
    drawString(text, len, x, 0);
    lmd.display();

    delay(ANIM_DELAY);

    if ( --x < len * -8 ) {
        x = LEDMATRIX_WIDTH;
    }
}

```

```

void loop()

```

```
{  
  scrollText();  
}
```

A moment or two after the sketch has uploaded, you should see text scrolling from right to left across your LED display modules.

Now let's dig in to see how this sketch works. There's a lot of code, but don't let that put you off. Starting at 1, we call the required functions to use the library and set up the displays. At 2, an array of characters contains the text to show on the display modules. You can change this later if you'd like. You can also adjust the speed of scrolling by altering the value at 3: the smaller the number, the faster the scroll speed.

At 4, we have two functions. This function turns the display on or off:

```
lmd.setEnabled(true);
```

And this one sets the brightness of the LEDs in the display module:

```
lmd.setIntensity(x);
```

The `setIntensity()` function takes values between 0 (dim) and 9 (bright).

The font used by the display is defined in the huge array at 5. We'll return to that in the next section. Finally, the functions `drawstring()` 6 and `scrollText()` 7 are required for display operation.

Editing the Display Font

You can easily specify which characters are usable in the display by changing the data in the byte `font` array 5. First, recall that each matrix module is made up of eight rows of eight LEDs. This means you have 64 LEDs available for any character you create.

Each row of LEDs is defined by a hexadecimal number, and eight of these hexadecimal numbers represent a character. For example, the letter N is defined by:

```
{0x00, 0x42, 0x42, 0x62, 0x52, 0x4a, 0x46, 0x42}, // N
```

To visualize the character, we convert the hexadecimal numbers to binary. For example, our letter N converted from hexadecimal to binary is:

```
0 0 0 0 0 0 0 0 = 0x00
0 1 0 0 0 0 1 0 = 0x42
0 1 0 0 0 0 1 0 = 0x42
0 1 1 0 0 0 1 0 = 0x62
0 1 0 1 0 0 1 0 = 0x52
0 1 0 0 1 0 1 0 = 0x4a
0 1 0 0 0 1 1 0 = 0x46
0 1 0 0 0 0 1 0 = 0x42
```

You can see how the 1s represent the character against a field of 0s, with the 1s being LEDs turned on and the 0s being LEDs turned off. So, to create your own characters, just reverse the process. For example, a nice smiley face can be represented as:

```
0 1 1 1 1 1 1 0 = 0x7e
1 0 0 0 0 0 0 1 = 0x81
1 0 1 0 0 1 0 1 = 0xa5
1 0 0 0 0 0 0 1 = 0x81
1 0 1 0 0 1 0 1 = 0xa5
1 0 0 1 1 0 0 1 = 0x99
1 0 0 0 0 0 0 1 = 0x81
0 1 1 1 1 1 1 0 = 0x7e
```

This would be represented in the array as:

```
{0x7e,0x81,0xa5,0x81,0xa5,0x99,0x81,0x7e} // smiley
```

You can either replace an existing line in the font array with your new data or add your data to the end of the array as another element. If you add another line, you need to increase the first parameter in the byte declaration so that the first parameter equals the number of defined characters (in this case, 96):

```
byte font[96][8]
```

You're probably wondering by now how to refer to your custom character in the sketch. The display library uses the character order in the ASCII

chart, which can be found at

<https://www.arduino.cc/en/Reference/ASCIIchart/>.

If you add another character after the last one in the sketch (which is Z by default), the next character in the table is [. Thus, to scroll three smiley faces across the display, you would set the line with text to display to:

```
char text[] = "[ [ [ ";
```

An example of this output can be seen in [Figure 8-11](#).

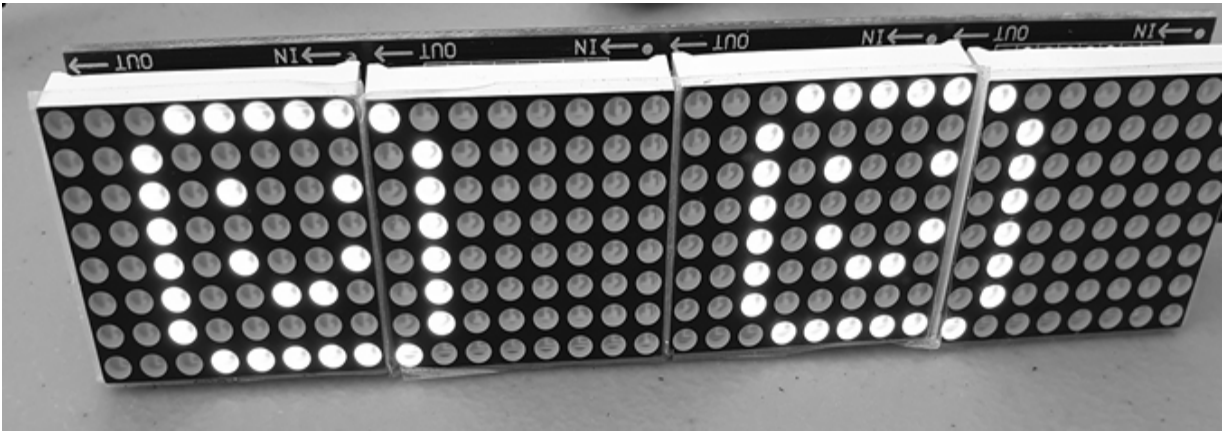


Figure 8-11: Using custom characters to display smiley faces

Looking Ahead

Now that you know how to use them, working with LED numeric and matrix displays will be a cinch. However, there are more types of displays, so turn to the next chapter to learn about another one: liquid crystal displays.

9

LIQUID CRYSTAL DISPLAYS

In this chapter you will

Use character LCD modules to display text and numeric data

Create custom characters to display on character LCD modules

Use color LCD modules to display text and data

Create a temperature history–graphing thermometer display

For some projects, you’ll want to display information to the user somewhere other than on a desktop computer monitor. One of the easiest and most versatile ways to display information is with a liquid crystal display (LCD) module and your Arduino. You can display text, custom characters, and numeric data using a character LCD module and color graphics with a graphic LCD module.

Character LCD Modules

LCD modules that display characters such as text and numbers are the most inexpensive and simplest to use of all LCDs. They can be purchased in various sizes, which are measured by the number of rows and columns of characters they can display. Some include a backlight and allow you to choose the color of the characters and the background color. Any LCD with an HD44780- or KS0066-compatible interface and a 5 V backlight should work with your Arduino. The first LCD we’ll use is a 16-character-by-2-row LCD module with a backlight, as shown in [Figure 9-1](#).

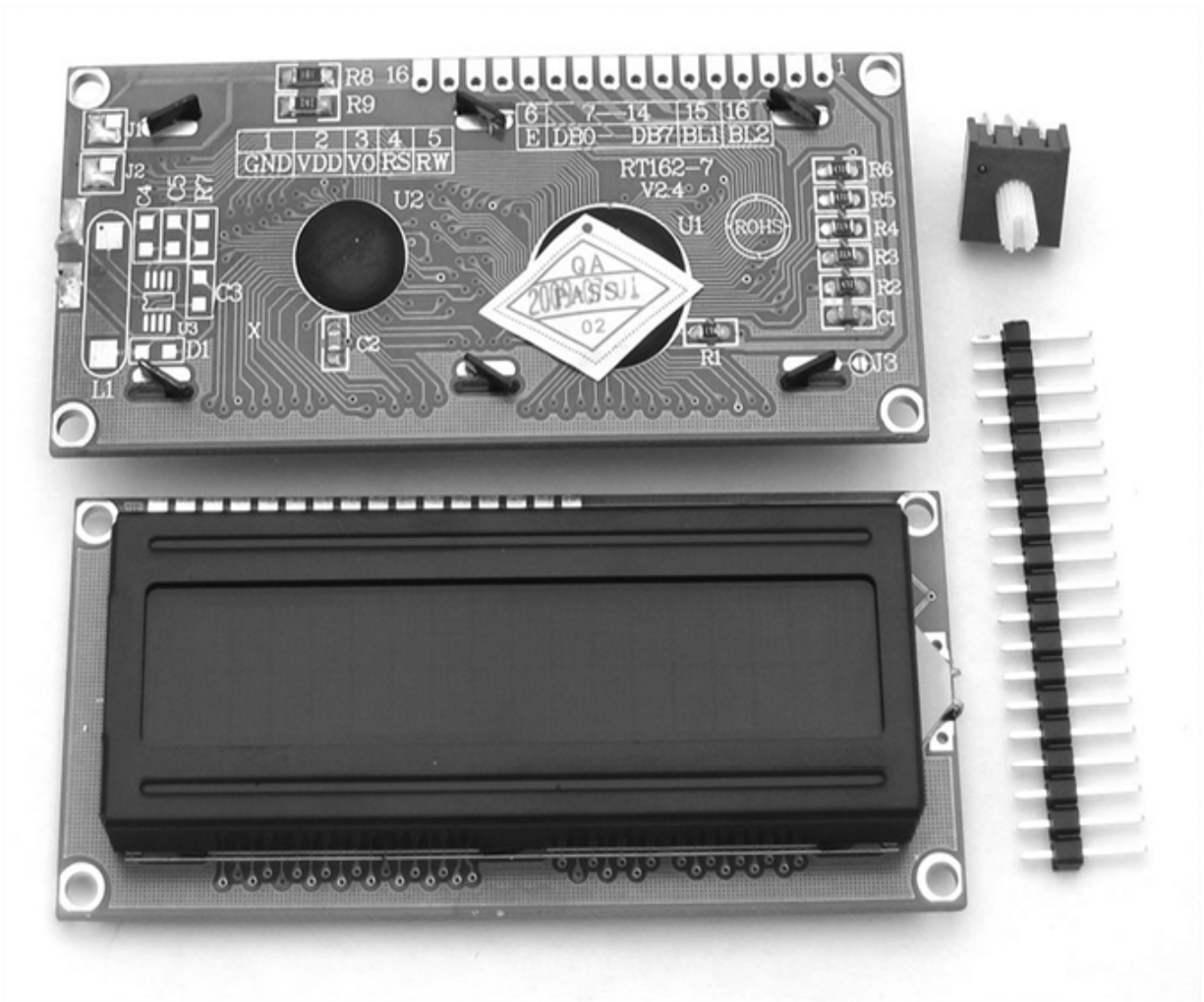


Figure 9-1: Example LCD module with trimpot and header pins

The trimpot (the variable resistor for the LCD) has a value of 10 k Ω and is used to adjust the display contrast. If the header pins have not already been soldered into the row of holes along the top of the LCD, you'll need to do this to make insertion into the breadboard straightforward.

The holes along the top of the LCD are numbered 1 through 16. Number 1 is closest to the corner of the module and marked as VSS (connected to GND) in the schematic shown in [Figure 9-2](#). We'll refer to this schematic for all of the LCD examples in this book. In some rare situations, you could find yourself with an LCD that has a 4.2 V instead of a 5 V backlight. (If you are unsure of this, check with your supplier.) If this is the case, place a 1N4004 diode in series between the Arduino 5 V and the LCD LED+ pin.

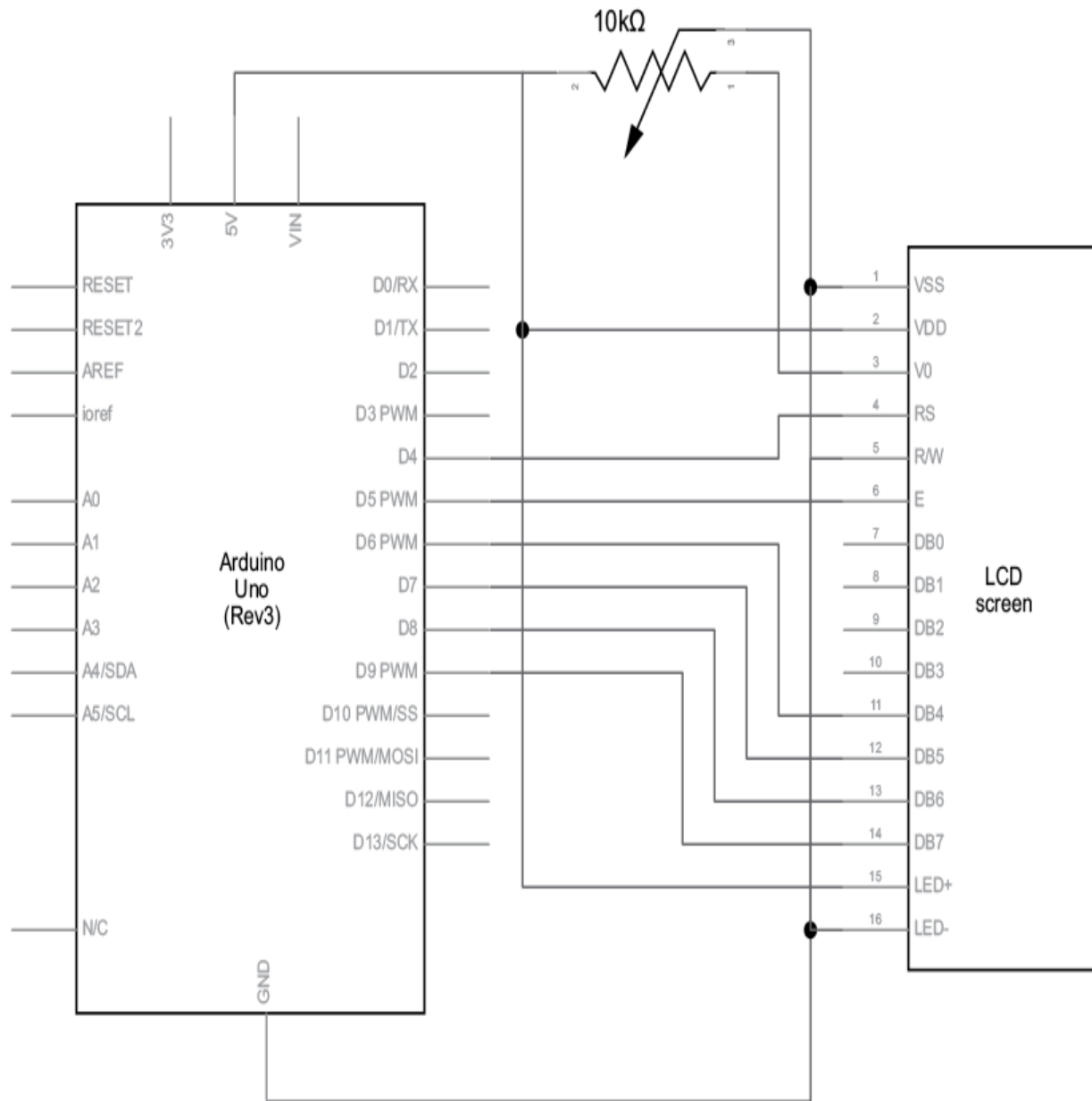


Figure 9-2: Basic LCD schematic

Using a Character LCD in a Sketch

To use the character LCD shown in [Figure 9-1](#), we will first explore the required functions and how they work through some simple demonstrations. Before moving on, you'll need to install the required Arduino library from the Library Manager. Using the method described in Chapter 7, search for and install the "LiquidCrystal by Arduino, Adafruit" library. Then you can enter and upload the basic sketch shown in [Listing 9-1](#).

```
// Listing 9-1
#include <LiquidCrystal.h>
LiquidCrystal lcd(4, 5, 6, 7, 8, 9); // pins for RS, E, DB4,
DB5, DB6, DB7
void setup()
{
    lcd.begin(16, 2);
    lcd.clear();
}
void loop()
{
    lcd.setCursor(0, 5);
    lcd.print("Hello");
    lcd.setCursor(1, 6);
    lcd.print("world!");
    delay(10000);
}
```

Listing 9-1: LCD demonstration sketch

[Figure 9-3](#) shows the result of [Listing 9-1](#).



Figure 9-3: LCD demonstration: “Hello world!”

Now to see how the sketch in [Listing 9-1](#) works. First, we need to add a line whose purpose is to include the library for LCD modules (which is automatically installed with the Arduino IDE). Then we need to tell the library which pins are connected to the Arduino. To do this, we add the following lines *before* the `void setup()` method:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(4, 5, 6, 7, 8, 9); // pins for RS, E, DB4,
DB5, DB6, DB7
```

The numbers entered in the `LiquidCrystal` function match the pins labeled on the LCD. If you’re unsure about your LCD’s pinouts, contact the

supplier.

If you need to use different digital pins on the Arduino, adjust the pin numbers in the second line of this code.

Next, in `void setup()`, we tell the Arduino the size of the LCD in columns and rows. For example, here's how we'd tell the Arduino that the LCD has 2 rows of 16 characters each:

```
lcd.begin(16, 2);
```

Displaying Text

With the LCD setup complete, clear the LCD's display with the following:

```
lcd.clear();
```

Then, to position the cursor, which is the starting point for the text, use this:

```
lcd.setCursor(x, y);
```

Here, *x* is the column (0 to 15) and *y* is the row (0 or 1). Next, to display the word *text*, for example, you would enter the following:

```
lcd.print("text");
```

Now that you can position and locate text, let's move on to displaying variable data.

Displaying Variables or Numbers

To display the contents of variables on the LCD screen, use this line:

```
lcd.print(variable);
```

If you're displaying a `float` variable, you can specify the number of decimal places to use. For example, here `lcd.print(pi, 3)` tells the Arduino to display the value of `pi` to three decimal places, as shown in [Figure 9-4](#):

```
float pi = 3.141592654;  
lcd.print("pi: ");  
lcd.print(pi, 3);
```

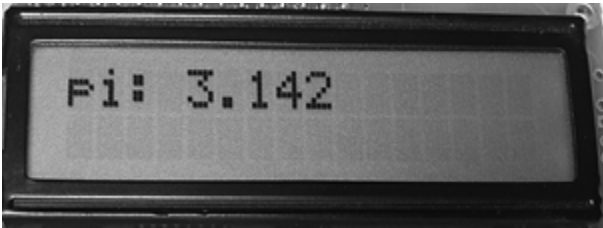


Figure 9-4: LCD displaying a floating-point number

When you want to display an integer on the LCD screen, you can display it in hexadecimal or binary, as shown in [Listing 9-2](#).

```
// Listing 9-2  
int zz = 170;  
lcd.setCursor(0, 0);  
lcd.print("Binary: ");  
lcd.print(zz, BIN);    // display 170 in binary  
lcd.setCursor(0, 1);  
lcd.print("Hexadecimal: ");  
lcd.print(zz, HX);    // display 170 in hexadecimal
```

Listing 9-2: Functions for displaying binary and hexadecimal numbers

The LCD will then display the text shown in [Figure 9-5](#).



Figure 9-5: Results of the code in [Listing 9-2](#)

Project #28: Defining Custom Characters

In addition to using the standard letters, numbers, and symbols available on your keyboard, you can define up to eight of your own characters in each

sketch. Notice in the LCD module that each character is made up of eight rows of five dots, or *pixels*. [Figure 9-6](#) shows a close-up.

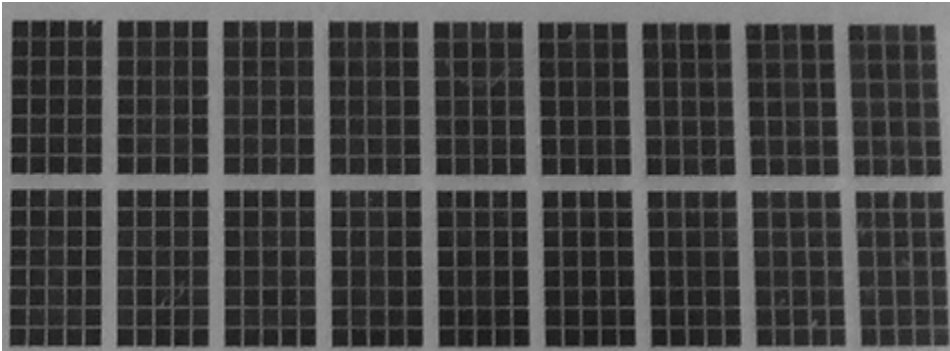


Figure 9-6: Each character is made up of eight rows of five pixels.

To display your own characters, you must first define each one using an *array*. For example, to create a smiley face, you could use the following:

```
byte a[8] = {  B00000,  
                B01010,  
                B01010,  
                B00000,  
                B00100,  
                B10001,  
                B01110,  
                B00000  };
```

Each number in the array addresses an individual pixel in the display. A 0 turns off a pixel, and a 1 turns it on. The elements in the array represent the rows of pixels in the display; the top element is the top row, the next element is the second row down, and so on.

NOTE

When you're planning your custom characters, it can be helpful to plan the character using some graph paper. Each square that is filled in represents a 1, and each empty square represents a 0 in the array.

In this example, since the first element is B00000, all the pixels in the top row are turned off. In the next element, B01010, every other pixel is turned on, and the 1s form the tops of the eyes. The following rows continue to fill out the character.

Next, assign the array (which defines your new character) to the first of the eight custom character slots in void setup() with the following function:

```
    lcd.createChar(0, a); // assign the array a[8] to custom
    character slot 0
```

Finally, to display the character, add the following in void loop():

```
    lcd.write(byte(0));
```

To display our custom character, we'd use the following code:

```
// Project 28 - Defining Custom Characters
#include <LiquidCrystal.h>
LiquidCrystal lcd(4, 5, 6, 7, 8, 9); // pins for RS, E, DB4,
DB5, DB6, DB7
byte a[8] = {    B00000,
                  B01010,
                  B01010,
                  B00000,
                  B00100,
                  B10001,
                  B01110,
                  B00000 };

void setup()
{
    lcd.begin(16, 2);
    lcd.createChar(0, a);
}
void loop()
{
    lcd.write(byte(0)); // write the custom character 0 to
the next cursor          // position
}
}
```

[Figure 9-7](#) shows the smiley faces displayed on the LCD screen.

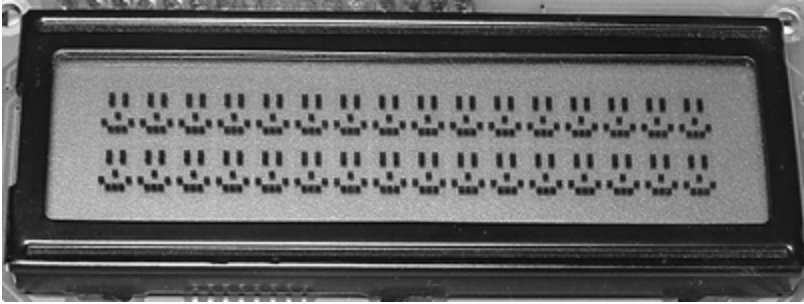


Figure 9-7: The result of Project 28

Character LCD modules are simple to use and somewhat versatile. For example, using what you’ve learned, you could create a detailed digital thermometer by combining this LCD and the temperature measurement part of Project 20, on page 122 in Chapter 6. However, if you need to display a lot of data or graphical items, you will need to use a *graphic LCD module*.

Graphic LCD Modules

Graphic LCD modules are larger and more expensive than character modules, but they’re also more versatile. You can use them not only to display text but also to draw lines, dots, circles, and more to create visual effects. The graphic LCD used in this book is a 128 × 160-pixel color module with an ST7735-compatible interface, as shown in [Figure 9-8](#).



Figure 9-8: A graphic LCD module

Connecting the Graphic LCD

Before you can use the graphic LCD, you’ll need to connect eight wires between the LCD and the Arduino. This is easily done with male-to-female

jumper wires, as the LCD has the connection pins presoldered at the factory. Make the connections as shown in [Table 9-1](#).

Table 9-1: *Connections Between the Graphic LCD Module and Arduino*

LCD pin label	To Arduino pin	LCD pin function
Vcc	5 V	VDD
GND	GND	VSS (GND)
CS	D10	Chip select
RST	D8	Reset
A0 (or DC)	D9	Control
SDA	D11	Data in
SCK	D13	Clock in
LED	3.3 V	Backlight LED

Using the LCD

Before moving on, you’ll need to install the required Arduino library from the Library Manager. Using the method described in Chapter 7, search for and install the “TFT by Arduino, Adafruit” library.

To use the LCD, insert the following three lines before `void setup()`:

```
#include <TFT.h>                // include the graphics LCD
library
#include <SPI.h>                // include the library for the
SPI data bus

TFT TFTscreen = TFT(10, 9, 8); // allocate pins to LCD
#include <SPI.h>                // library for SPI data bus
```

(Don’t panic about the “SPI data bus”; for now, the line above is all you need to know. We’ll examine the SPI bus in more detail in Chapter 19.)

Then add the following lines inside `void setup()` to prepare the display:

```
TFTscreen.begin();                // activate LCD
TFTscreen.background(0, 0, 0);    // clear the LCD screen
```

Controlling the Display

There are five text sizes you can choose from, as shown in Figures 9-9 and 9-10.

The first thing you need to consider is the background color for the display you are generating. This is set with:

```
TFTscreen.background(b, g, r); // set background color
```



Figure 9-9: Four of the five text sizes available on the LCD



Figure 9-10: The largest of the five text sizes available on the LCD

You set the color of the background using RGB (red, green, blue) values between 0 and 255. For example, a white background would be maximum red, maximum green, and maximum blue—so 255, 255, 255. A pure red background would have a value of 255 for red and values of 0 for green and

blue. For a black background, use zero for all three values. (You can find a handy list of RGB color tables at

https://www.rapidtables.com/web/color/RGB_Color.html.)

Next, you need to set the text size if you're writing text to the LCD for the first time or if you need to change the size mid-sketch. To do this, use:

```
TFTscreen.setTextSize(x);
```

where x is a number between 1 and 5 that matches the text sizes shown in Figures 9-9 and 9-10.

Then you set the color of the text with the following function:

```
TFTscreen.stroke(B, G, R);
```

where B , G , and R are the corresponding values for your blue, green, and red color levels, respectively.

Finally, to write text to your screen, use the following function:

```
TFTscreen.text("Hello, world!", x, y);
```

This will display the text “Hello, world!” with the top left of the text positioned on the LCD at x , y .

This works great for static text. However, if you want to display a numeric variable, you need to do a little more work. The variable needs to be converted from a number type to a character array whose size will match the largest possible value. For example, if you're reading the Arduino's analog input 0 and want to display the value, use this:

```
char analogZero[4];
```

Then during the sketch, before sending the analog value to the LCD, convert the value to a string, like so:

```
String sensorVal = String(analogRead(A0));
```

This string gets converted and inserted into the character array:

```
sensorVal.toCharArray(analogZero, 4);
```

Finally, to display the value on the LCD, we can use the `.text()` command as usual:

```
TFTscreen.text(analogZero, x, y);
```

where the value of *analogZero* is displayed with the top left of the text positioned at *x*, *y*.

Now that we've been through all the commands for using text on the LCD, let's put them into action in the next project.

Project #29: Seeing the Text Functions in Action

With this project, you'll make your LCD display text in five sizes as well as the numeric value read from your Arduino's analog input 0.

The Sketch

Wire up your LCD as described in [Table 9-1](#) and then upload the following sketch:

```
// Project 29 - Seeing the Text Functions in Action
#include <TFT.h> // Arduino TFT LCD library
#include <SPI.h> // SPI bus library

TFT TFTscreen = TFT(10, 9, 8); // allocate digital pins
to LCD

char analogZero[4];

void setup()
{
  TFTscreen.begin(); // activate LCD
  TFTscreen.background(0, 0, 0); // set display to black
}

void loop()
{
```

```
TFTscreen.stroke(255, 255, 255); // white text
TFTscreen.setTextSize(1);
TFTscreen.text("Size One", 0, 0);
TFTscreen.setTextSize(2);
TFTscreen.text("Size Two", 0, 10);
TFTscreen.setTextSize(3);
TFTscreen.text("Size 3", 0, 30);
TFTscreen.setTextSize(4);
TFTscreen.text("Size 4", 0, 55);
delay(2000);
TFTscreen.background(0, 0, 0); // set display to black
TFTscreen.setTextSize(5);
TFTscreen.text("Five", 0, 0);
delay(2000);
TFTscreen.background(0, 0, 0); // set display to black
TFTscreen.stroke(255, 255, 255); // white text
TFTscreen.setTextSize(1);
TFTscreen.text("Sensor Value :\n ", 0, 0);
TFTscreen.setTextSize(3);
String sensorVal = String(analogRead(A0));
// convert the reading to a char array
sensorVal.toCharArray(analogZero, 4);
TFTscreen.text(analogZero, 0, 20);
delay(2000);
TFTscreen.background(0, 0, 0); // set display to black
}
```

Running the Sketch

You should see all five sizes of text displayed on the LCD over two screens. Then you should see a third screen with the value from analog input 0, like the example shown in [Figure 9-11](#).



Figure 9-11: Analog input value shown on TFT LCD

Creating More Complex Display Effects with Graphic Functions

Now let's look at the functions we can use to create various display effects. Keep in mind that the graphic LCD screen has a resolution of 160 columns by 128 pixels, but when we refer to these columns and pixels in functions in our sketches, they are counted from 0 to 159 across and 0 to 127 down. Also, as with the text example earlier, we still need to use the five lines of code mentioned in “Using a Character LCD in a Sketch” on page 169 to initialize the display.

There are various functions that allow you to display dots (single pixels), lines, rectangles, and circles on the display. Apply your project requirements and add a dash of imagination to create a colorful and useful display output. We'll run through those functions now, and then you can see them in action through a demonstration sketch.

Before drawing any object, you need to define its color. This is done with

```
TFTscreen.stroke(B, G, R);
```

where *B*, *G*, and *R* are the corresponding values for your blue, green, and red color levels, respectively.

To draw a single dot on the display, we use

```
TFTscreen.point(X, Y);
```

where X and Y are the horizontal and vertical coordinates of the dot. With our LCD, the X range falls between 0 and 159 and the Y range falls between 0 and 127.

To draw a line from one point to another, we use

```
TFTscreen.line(X1, Y1, X2, Y2);
```

where $X1$ and $Y1$ are the coordinates of the starting point and $X2$ and $Y2$ are the coordinates of the end of the line.

To draw a circle, we use

```
TFTscreen.circle(X, Y, R);
```

where X and Y are the coordinates of the center of the circle, and R is the radius of the circle in pixels. If you wish to fill the circle (or a rectangle, described a bit later) with a color, instead of just drawing an outline, precede the `circle()` function with

```
TFTscreen.fill(B, G, R);
```

where B , G , and R are the corresponding values for your blue, green, and red fill levels, respectively. Note that a fill color doesn't change the shape's outline, so you still need to precede the shape function with the `stroke()` function.

If you wish to draw more than one filled item, you only need to use the `fill()` command once. If you then want to turn off the fill and revert to outlines only, use this:

```
TFTscreen.noFill();
```

Finally, you can draw rectangles with the following function:

```
TFTscreen.rect(X1, Y1, X2, Y2);
```

where $x1$, $y1$ are the coordinates for the top left of the rectangle and $x2$, $y2$ are the coordinates for the bottom right of the rectangle.

Project #30: Seeing the Graphic Functions in Action

Now that we've been through all the commands for using the graphic functions on the LCD, let's put them into action in this project.

The Sketch

Wire up your LCD as described in [Table 9-1](#) and then upload the following sketch:

```
// Project 30 - Seeing the Graphic Functions in Action

#include <TFT.h>                // Arduino TFT LCD library
#include <SPI.h>                // SPI bus library
TFT TFTscreen = TFT(10, 9, 8); // allocate digital pins to
LCD

int a;

void setup()
{
  TFTscreen.begin(); // activate LCD
  TFTscreen.background(0, 0, 0); // set display to black
  randomSeed(analogRead(0));    // for random numbers
}

void loop()
{
  // random dots
  for (a = 0; a < 100; a++)
  {
    TFTscreen.stroke(random(256), random(256), random(256));
    TFTscreen.point(random(160), random(120));
    delay(10);
  }
  delay(1000);
  TFTscreen.background(0, 0, 0); // set display to black
```

```

// random lines
for (a = 0; a < 100; a++)
{
    TFTscreen.stroke(random(256), random(256), random(256));
    TFTscreen.line(random(160), random(120), random(160),
random(120));
    delay(10);
}
delay(1000);
TFTscreen.background(0, 0, 0); // set display to black

// random circles
for (a = 0; a < 50; a++)
{
    TFTscreen.stroke(random(256), random(256), random(256));
    TFTscreen.circle(random(160), random(120), random(50));
    delay(10);
}
delay(1000);
TFTscreen.background(0, 0, 0); // set display to black

// random filled circles
for (a = 0; a < 50; a++)
{
    TFTscreen.fill(random(256), random(256), random(256));
    TFTscreen.stroke(random(256), random(256), random(256));
    TFTscreen.circle(random(160), random(120), random(50));
    delay(10);
}
delay(1000);
TFTscreen.background(0, 0, 0); // set display to black

// random rectangles
TFTscreen.noFill();
for (a = 0; a < 50; a++)
{
    TFTscreen.stroke(random(256), random(256), random(256));
    TFTscreen.rect(random(160), random(120), random(160),
random(120));
    delay(10);
}
delay(1000);
TFTscreen.background(0, 0, 0); // set display to black

// random filled rectangles
TFTscreen.noFill();
for (a = 0; a < 50; a++)
{

```

```
TFTscreen.fill(random(256), random(256), random(256));  
TFTscreen.stroke(random(256), random(256), random(256));  
TFTscreen.rect(random(160), random(120), random(160),  
random(120));  
  delay(10);  
}  
delay(1000);  
TFTscreen.background(0, 0, 0); // set display to black  
}
```

After the sketch has uploaded, the display will run through all the graphic functions we have examined in this chapter. For example, you should see the lines shown in [Figure 9-12](#).

With the functions discussed so far and some imagination, you can create a variety of display effects or display data graphically. In the next section, we'll build on our quick-read thermometer project using the LCD screen and some of these functions.

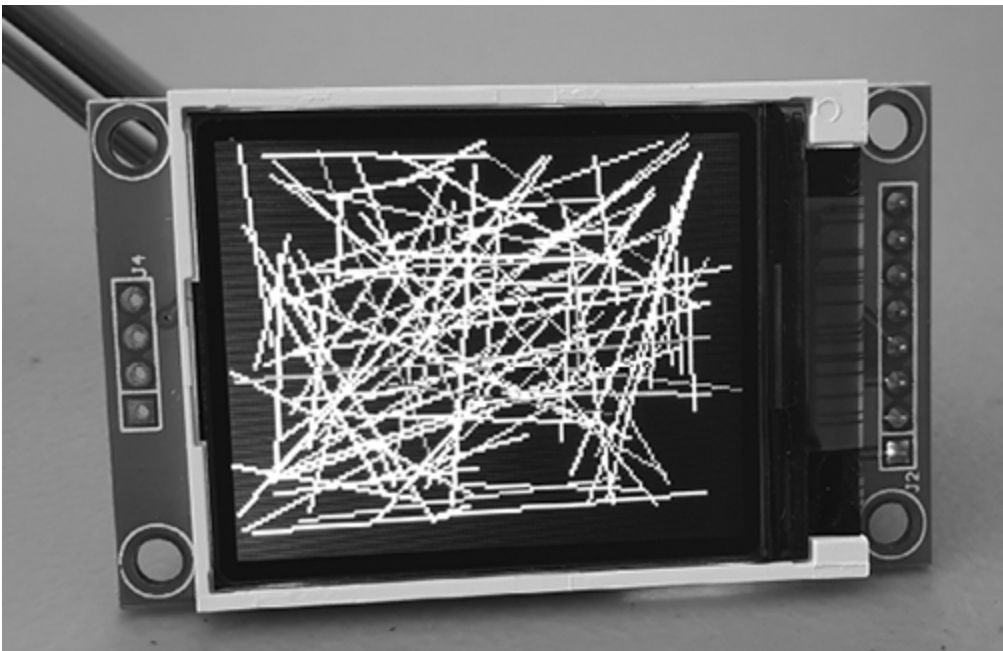


Figure 9-12: Random lines on the LCD

Project #31: Creating a Temperature History Monitor

In this project, our goal is to measure the temperature once every 20 minutes and display the last 120 readings in a dot graph. Each reading will be represented as a pixel, with the temperature on the vertical axis and time on the horizontal axis.

The most current reading will appear on the left, and the display will continually scroll the readings from left to right. The current temperature will also be displayed as a numeral.

The Algorithm

Although it may sound complex, this project is fairly easy, requiring only two functions. The first function takes a temperature reading from the TMP36 temperature sensor and stores it in an array of 120 values. Each time a new reading is taken, the previous 119 values are moved down the array to make way for the new reading, and the oldest reading is erased.

The second function draws on the LCD screen. It displays the current temperature, a scale for the graph, and the positions of each pixel for the display of the temperature data over time.

The Hardware

Here's what you'll need to create this project:

One 160 × 128-pixel ST7735 TFT LCD module, as used in this chapter

One TMP36 temperature sensor

Various connecting wires

One breadboard

Arduino and USB cable

Connect the graphic LCD as described in [Table 9-1](#) and connect the TMP36 sensor to 5 V, analog 5, and GND as you did in Project 20 in Chapter 6.

The Sketch

Our sketch combines the code we used to measure temperature in Chapter 6 and the graphic functions described earlier in this chapter. Enter and upload

the following sketch, which includes relevant comments about the functions used:

```
// Project 31 - Creating a Temperature History Monitor

#include <TFT.h> // Arduino TFT LCD library
#include <SPI.h> // SPI bus library

TFT TFTscreen = TFT(10, 9, 8);
// allocate digital pins to LCD

int tcurrent = 0;
int tempArray[120];

char currentString[3];

void getTemp() // function to read temperature from TMP36
{
    float sum = 0;
    float voltage = 0;
    float sensor = 0;
    float celsius;

    // read the temperature sensor and convert the result to
    degrees C
    sensor = analogRead(5);
    voltage = (sensor * 5000) / 1024;
    voltage = voltage - 500;
    celsius = voltage / 10;
    tcurrent = int(celsius);

    // insert the new temperature at the start of the array of
    past temperatures
    for (int a = 119 ; a >= 0 ; --a )
    {
        tempArray[a] = tempArray[a - 1];
    }
    tempArray[0] = tcurrent;
}

void drawScreen() // generate TFT LCD display effects
{
    int q;
    // display current temperature
    TFTscreen.background(0, 0, 0); // clear screen to black
    TFTscreen.stroke(255, 255, 255); // white text
    TFTscreen.setTextSize(2);
```

```

TFTscreen.text("Current:", 20, 0);
String tempString = String(tcurrent);
tempString.toCharArray(currentString, 3);
TFTscreen.text(currentString, 115, 0);
// draw scale for graph
TFTscreen.setTextSize(1);
TFTscreen.text("50", 0, 20);
TFTscreen.text("45", 0, 30);
TFTscreen.text("40", 0, 40);
TFTscreen.text("35", 0, 50);
TFTscreen.text("30", 0, 60);
TFTscreen.text("25", 0, 70);
TFTscreen.text("20", 0, 80);
TFTscreen.text("15", 0, 90);
TFTscreen.text("10", 0, 100);
TFTscreen.text(" 5", 0, 110);
TFTscreen.text(" 0", 0, 120);
TFTscreen.line(20, 20, 20, 127);
// plot temperature data points
for (int a = 25 ; a < 145 ; a++)
{
    // convert the temperature value to a suitable y-axis
    position on the LCD
    q = (123 - (tempArray[a - 25] * 2));
    TFTscreen.point(a, q);
}
}

void setup()
{
    TFTscreen.begin();           // activate LCD
    TFTscreen.background(0, 0, 0); // set display to black
}

void loop()
{
    getTemp();
    drawScreen();
    for (int a = 0 ; a < 20 ; a++) // wait 20 minutes until the
    next reading
    {
        delay(60000);           // wait 1 minute
    }
}

```

Running the Sketch

The resulting display should look something like [Figure 9-13](#).



Figure 9-13: Results of Project 31

Modifying the Sketch

Different people can interpret data better when they see it presented in different visual formats. For this reason, you may want to create a bar graph instead, with vertical lines indicating the values.

This type of project could also be used to display other kinds of data, such as the voltage from various sensors as measured by analog input pins. Or you could add another temperature sensor and show both values at once. Almost anything that returns a value can be displayed using the graphic LCD module.

Looking Ahead

Now that you have experience with LCDs, you can see that the Arduino is in fact a small computer: it can accept and process incoming data and display it to the outside world. But this is only the beginning. In the next chapter, you'll examine libraries in much more depth, learn to write your own library, and then use your new library with the temperature sensor used in previous projects.

10

CREATING YOUR OWN ARDUINO LIBRARIES

In this chapter you will

- Learn the components of an Arduino library

- Create a simple library for a repetitive task

- Learn how to install your library in the Arduino IDE

- Create a library that accepts values to perform a function

- Create a library that processes data from a sensor and returns values in an easy-to-use form

Recall Project 22, described in Chapter 7, where you installed an Arduino library that included the functions needed to save data to an SD card. Using the library reduced the amount of time needed to write a sketch, as the library provides the functions related to the card module.

In the future, as you write sketches to solve your own problems and perform your own tasks, you may find yourself repeatedly using certain functions that you have created. At that point, it will be sensible to create your own Arduino library, which you can easily install and use in your sketches.

In this chapter, you will learn how to convert functions into an Arduino library. By following the examples presented here, you'll learn what you need to know to make your own custom libraries. Let's do this now.

Creating Your First Arduino Library

For our first example, consider [Listing 10-1](#). It contains two functions, `blinkSlow()` and `blinkFast()`, which are used to blink the Arduino's onboard LED at a slow or fast rate, respectively.

```
// Listing 10-1
void setup()
{
    pinMode(13, OUTPUT); // using onboard LED
}

void blinkSlow()
{
    for (int i = 0; i < 5; i++)
    {
        digitalWrite(13, HIGH);
        delay(1000);
        digitalWrite(13, LOW);
        delay(1000);
    }
}

void blinkFast()
{
    for (int i = 0; i < 5; i++)
    {
        digitalWrite(13, HIGH);
        delay(250);
        digitalWrite(13, LOW);
        delay(250);
    }
}

void loop()
{
    blinkSlow();
    delay(1000);
    blinkFast();
    delay(1000);
}
```

[Listing 10-1](#): Blinking the Arduino's onboard LED

Without a library, every time you wrote a new sketch and wanted to use the `blinkSlow()` and `blinkFast()` functions, you would have to enter them manually. On the other hand, if you put the code for your functions in a library, from then on, you'll be able to call the library using just one line of code at the start of your sketch.

Anatomy of an Arduino Library

An Arduino library consists of three files, as well as some optional example sketches that demonstrate how the library could be used. The three requisite files for every Arduino library are these:

<library>.h The header file

<library>.cpp The source file

KEYWORDS.TXT The keyword definitions

In the first two filenames, you'll replace *<library>* with the actual name of your library. For our first example, we will call our Arduino library *blinko*. Thus, our two files will be *blinko.h* and *blinko.cpp*.

The Header File

The *blinko.h* file is known as a *header file*, because it contains the definitions of functions, variables, and so on used inside the library. The header file for the *blinko* library is shown in [*Listing 10-2*](#).

```
// Listing 10-2
/*
1 blinko.h - Library for flashing an Arduino's onboard LED
  connected to D13
  */
2 #ifndef blinko_h
  #define blinko_h

3 #include "Arduino.h" // gives library access to standard types
                        // and constants
                        // of the Arduino language

4 class blinko // functions and variables used in the library
{
```

```
    public:
        blinko();
        void slow();
        void fast();
};
5 #endif
```

Listing 10-2: The blinko library header file

The header file shares some similarities with a typical Arduino sketch, but there are also some differences. At 1, there's a useful comment about the purpose of the library. While such comments are not necessary, they should be included to make the library easier for others to use.

At 2, the code checks whether the library has been declared in the host sketch. At 3, the standard Arduino library is included to allow our blinko library access to the standard Arduino sketch functions, types, and constants.

Then, at 4, we create a class. You can think of a *class* as a collection in one spot of all the variables and functions required for the library, including the name of the library. Within the class, there can be public variables and functions, which can be accessed by the sketch that needs to use the library; there can also be private variables and functions, which can be used only from inside the class. Finally, each class has a *constructor* with the same name as the class, which is used to create an instance of the class. This may sound complex. However, after reviewing the examples in this chapter and making a few libraries of your own, you'll be confident in these constructions.

Inside our class, you can see we have the constructor for our library, `blinko()`, and two functions that will be in the library: `slow()` and `fast()`. They follow the `public:` statement, which means they can be used by anyone ("any member of the public") who accesses the blinko library.

Finally, at 5, we end the header definition. By wrapping the header definition inside an `if` statement, we ensure that the header isn't loaded twice.

The Source File

Next, let's take a look at the *blinko.cpp* file. The *.cpp* file is known as a *source file*, because it contains the code that will be run when the library is used. The source file for the blinko library is given in [Listing 10-3](#).

```
// Listing 10-3
/*
1 blinko.cpp - Library for flashing an Arduino's onboard LED
  connected to D13
  */

2 #include "Arduino.h" // gives library access to standard types
  and constants          // of the Arduino language
  #include "blinko.h"

3 blinko::blinko()      // things to do when library is activated
{
    pinMode(13, OUTPUT);
}

4 void blinko::slow()
{
    for (int i=0; i<5; i++)
    {
        digitalWrite(13, HIGH);
        delay(1000);
        digitalWrite(13, LOW);
        delay(1000);
    }
}

4 void blinko::fast()
{
    for (int i=0; i<5; i++)
    {
        digitalWrite(13, HIGH);
        delay(250);
        digitalWrite(13, LOW);
        delay(250);
    }
}
```

Listing 10-3: The blinko library source file

The source file contains the functions we've written that we'll want available to reuse. In addition, some new structural elements are required here. At 2, we give our library access to both the standard Arduino functions, types, and constants and our own library header file.

At 3 we have the definition of the constructor function. The constructor contains things that should happen when the library is used. In our example, we have set digital pin 13 as an output, as we are using the Arduino's onboard LED.

Starting at 4, we list the functions we want to include in this library. They are just like the functions that you would create in a stand-alone sketch, with one important difference: their definition starts with the library class name and two colons. For example, instead of typing `void fast()`, you type `void blinko::fast()`.

The KEYWORDS.TXT File

Finally, we need to create the *KEYWORDS.TXT* file. The Arduino IDE uses this file to determine the keywords in the library, then highlights those words in the IDE. [Listing 10-4](#) is the *KEYWORDS.TXT* file for our blinko library.

```
// Listing 10-4
blinko      KEYWORD1
slow       KEYWORD2
fast       KEYWORD2
```

[Listing 10-4](#): The blinko library keywords file

The first line is the name of the library and is referred to as `KEYWORD1`. The library's functions are both called `KEYWORD2`. Note that the space between the keywords and their definitions must be created by pressing `TAB`, not by pressing the spacebar.

At this point, you have the three files needed for a working library. It's a great idea to also include an example sketch so users can understand what the functions do. [Listing 10-5](#) is our example sketch for the blinko library.

```
// Listing 10-5, blinkotest.ino
1 #include <blinko.h>

2 blinko ArduinoLED;
   void setup() {
       }

   void loop()
   {
3   ArduinoLED.slow(); // blink LED slowly, once every second
       delay(1000);
4   ArduinoLED.fast(); // blink LED rapidly, four times per
       second
       delay(1000);
   }
}
```

Listing 10-5: An example sketch for our blinko library

As you can see, the sketch is basic. It just shows the use of both the `slow()` and `fast()` functions in our library. All the end user needs to do after installing the library is to include the library 1, create an instance 2, and then call either function when required as shown at 3 and 4.

Installing Your New Arduino Library

Now that you've created a new Arduino library, an easy way to store and distribute it is to make a ZIP file. Future users who obtain the ZIP file can easily install the library, as demonstrated earlier in Chapter 7.

Creating a ZIP File Using Windows 7 and Later

To create a ZIP file with Windows, follow these instructions.

First, place the three library files and the example sketch (stored in its own folder, as are all sketches) into one location. [Figure 10-1](#) shows an example.

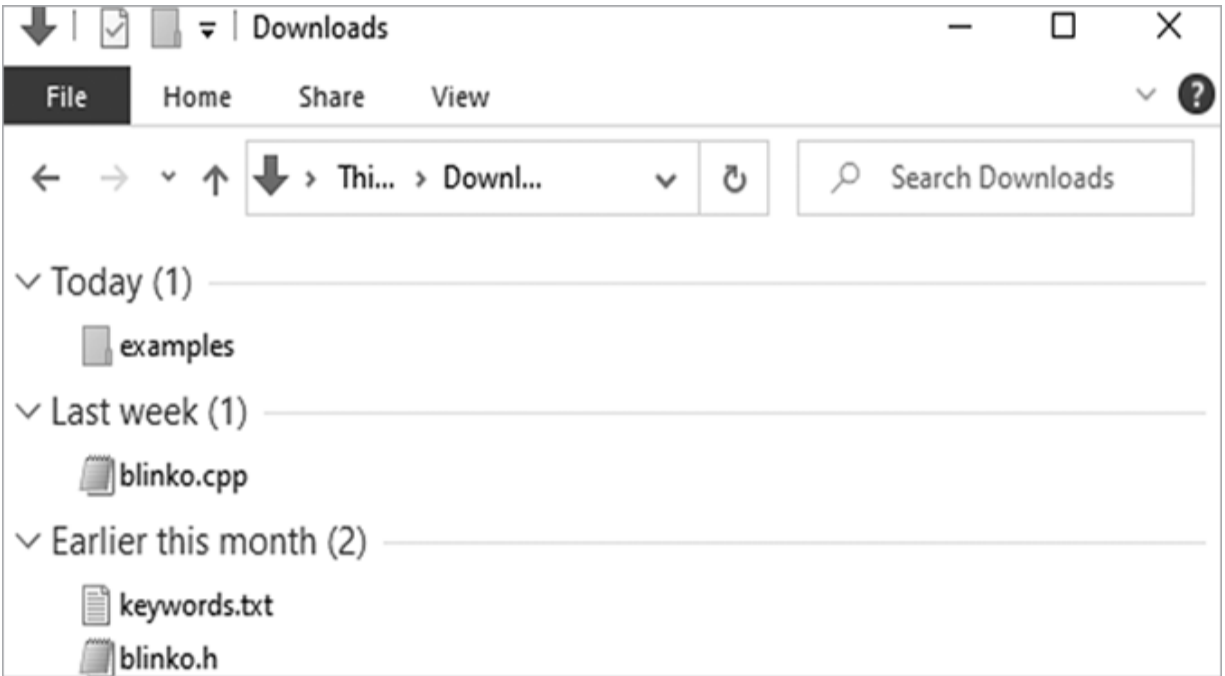


Figure 10-1: Our Arduino library files in one folder

Select all the files, right-click anywhere over the highlighted files, and select **Send To►Compressed (Zipped) Folder**, as shown in [Figure 10-2](#).

A new file will appear in the folder, with a *.zip* extension and name editing enabled. For our library, change the name to *blinko* and then press ENTER, as shown in [Figure 10-3](#).

Now you can move on to “Installing Your New Library” on page 193.

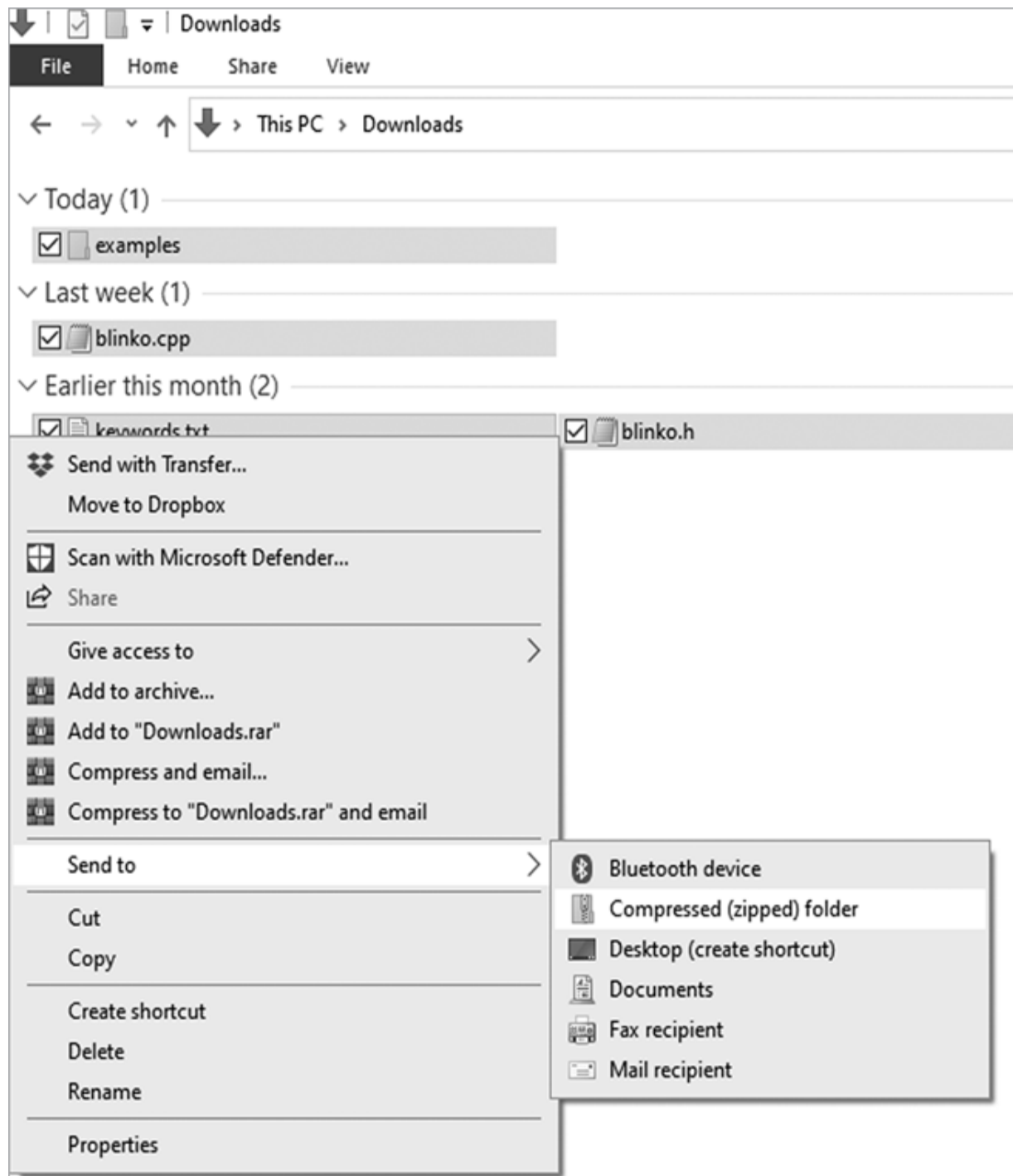


Figure 10-2: Compressing the library files

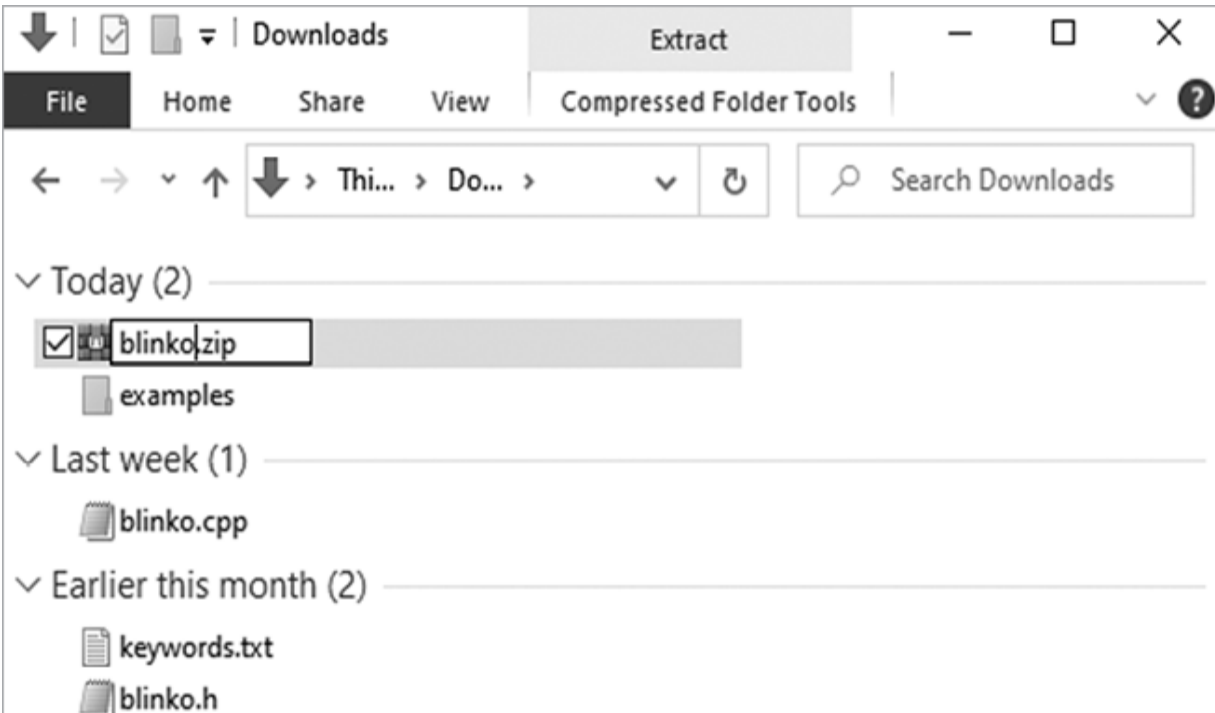


Figure 10-3: Changing the name of the library ZIP file

Creating a ZIP File Using Mac OS X or Later

To create a ZIP file with Mac OS X, gather the three library files and the example sketch (stored in its own folder, as are all sketches) into one location. [Figure 10-4](#) shows an example.

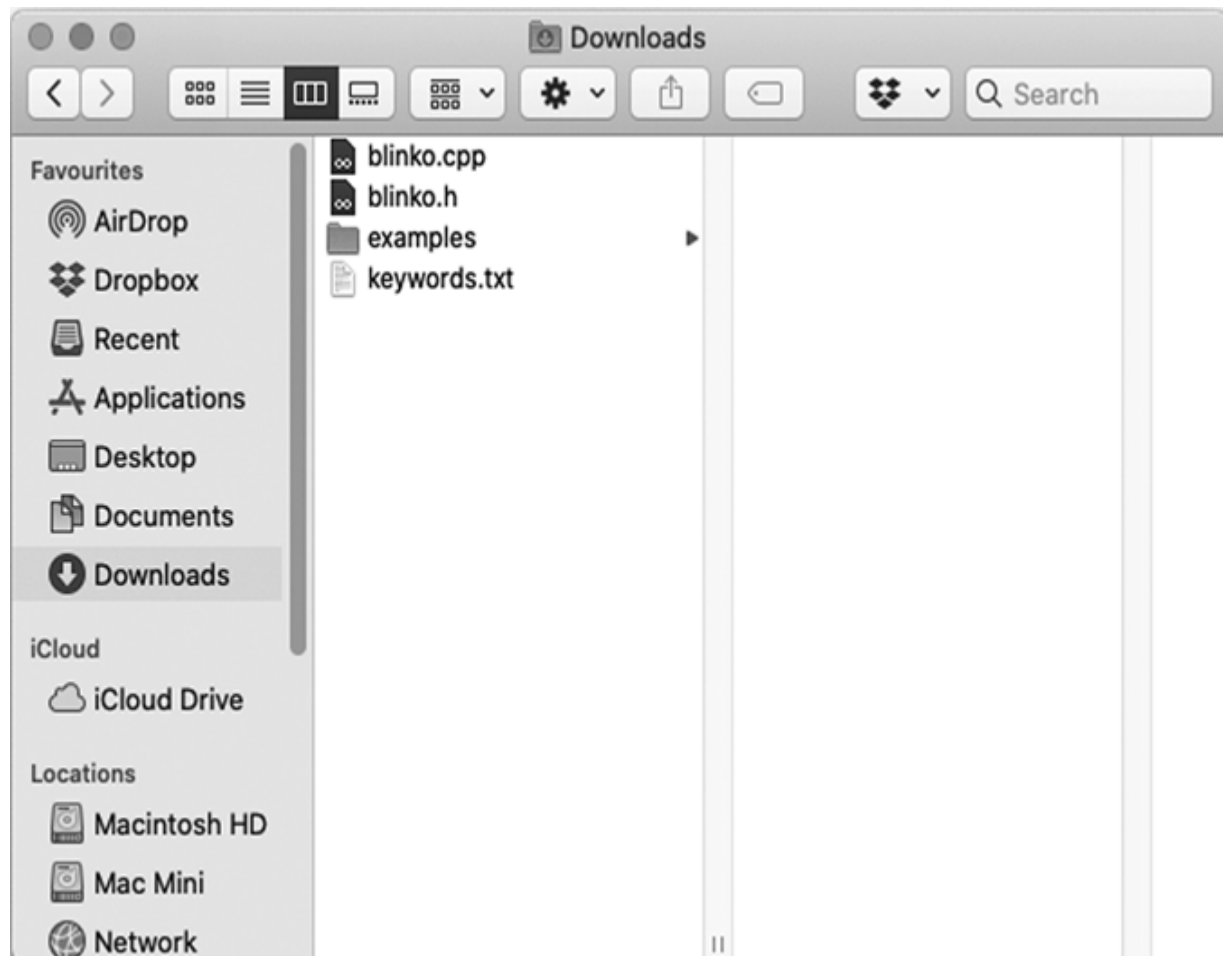


Figure 10-4: Our Arduino library files

Select all the files, right-click anywhere over the files, and select **Compress 4 Items**, as shown in [Figure 10-5](#).

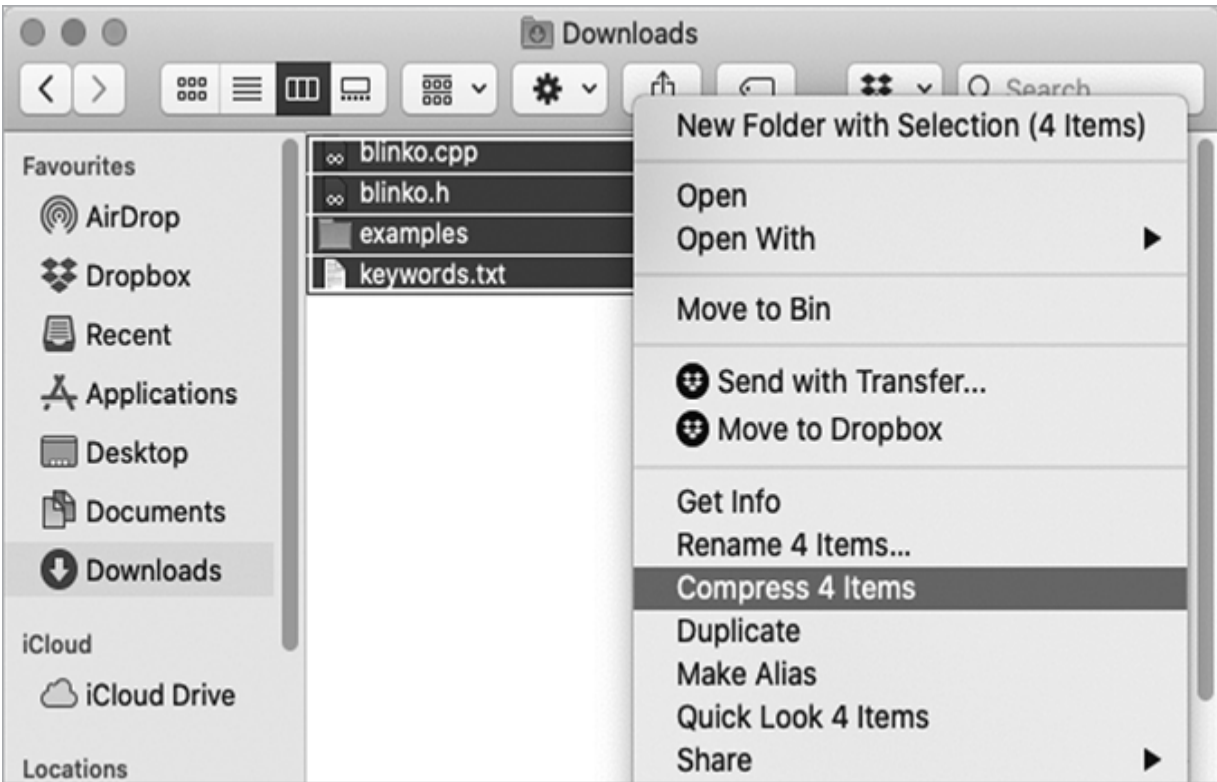


Figure 10-5: Compressing the library files

After a moment, a new file called *Archive.zip* will appear in the folder, as shown in [Figure 10-6](#).

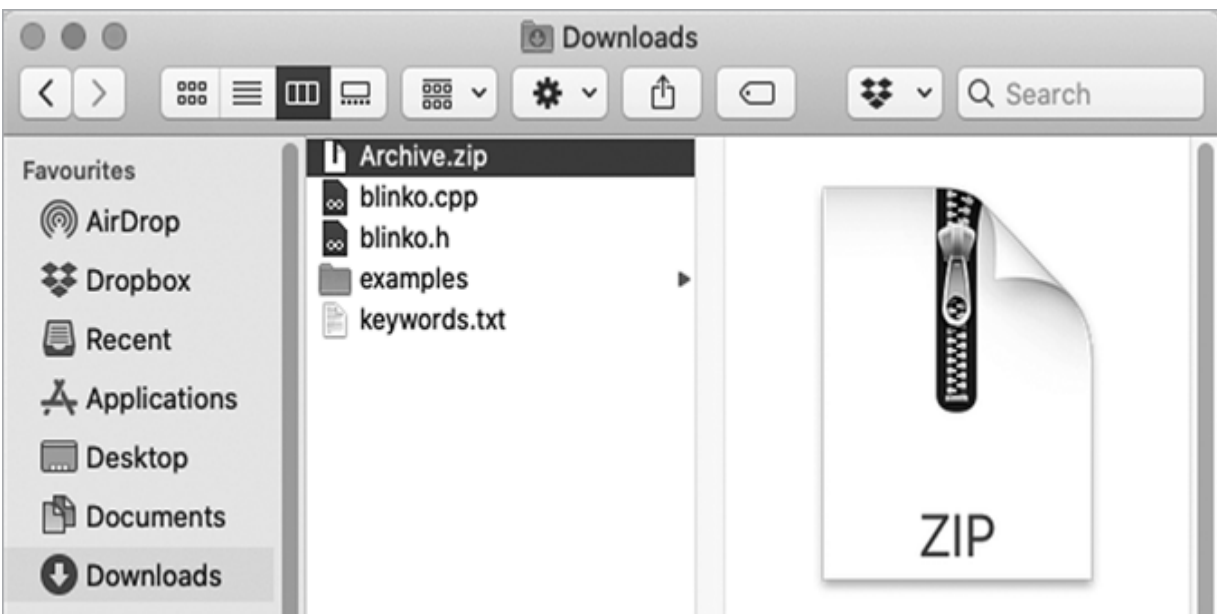


Figure 10-6: The files have been compressed.

Click on the *Archive.zip* folder and change the name to *blinko.zip*, as shown in [Figure 10-7](#).

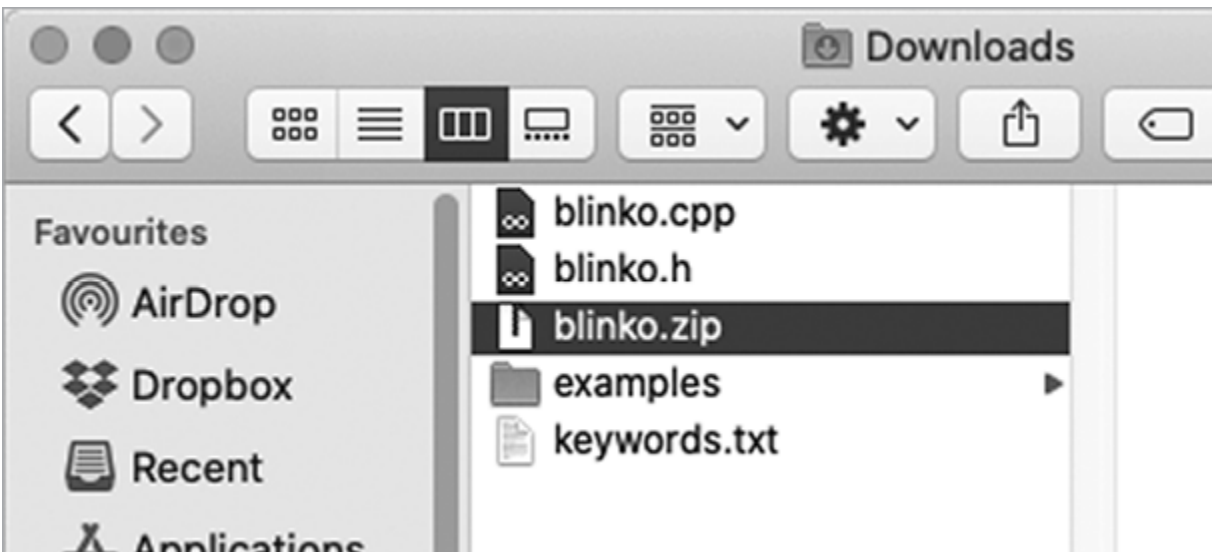


Figure 10-7: Our Arduino library installation ZIP file

You now have a library ZIP file that you can easily distribute to others or install yourself.

Installing Your New Library

At this point you can install your library using the ZIP file method detailed in “Downloading an Arduino Library as a ZIP File” on page 134 in Chapter 7. Once the file has been installed and you have restarted the Arduino IDE, select **Sketch►Include Library** to see your library listed, as shown in [Figure 10-8](#).

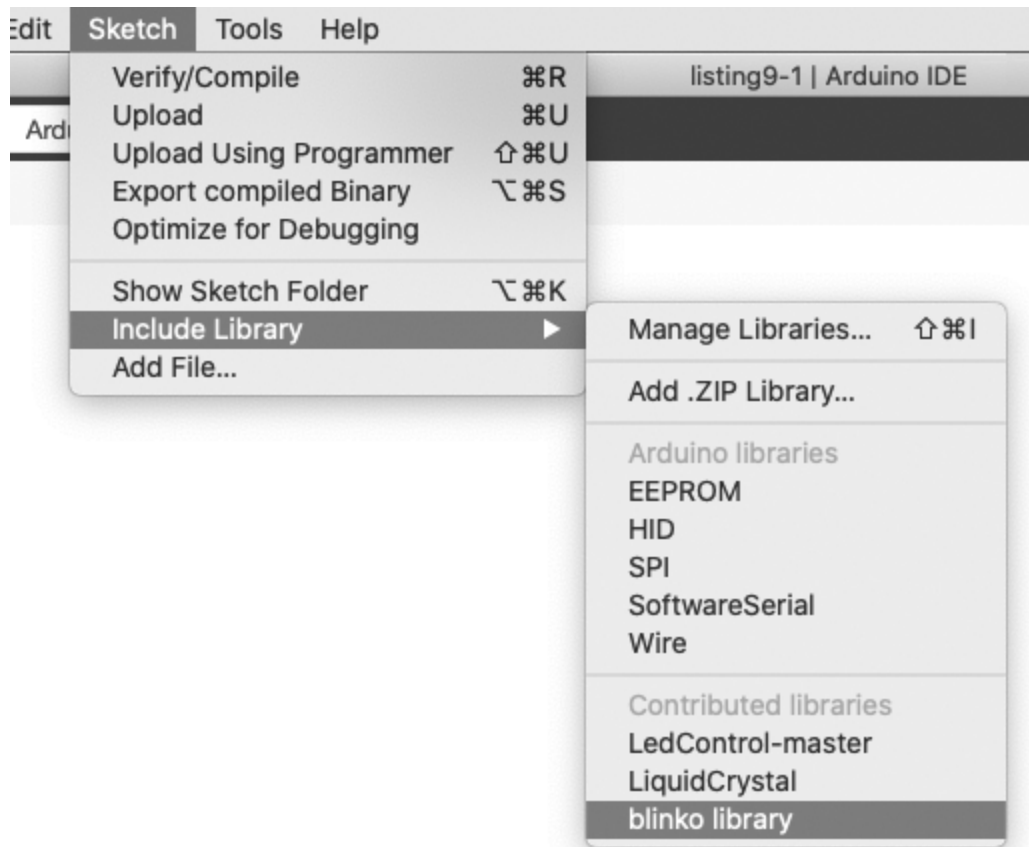


Figure 10-8: Our Arduino library, now available in the IDE

Furthermore, you can now easily access the example sketch; select **File►Examples►blinko**, as shown in [Figure 10-9](#).

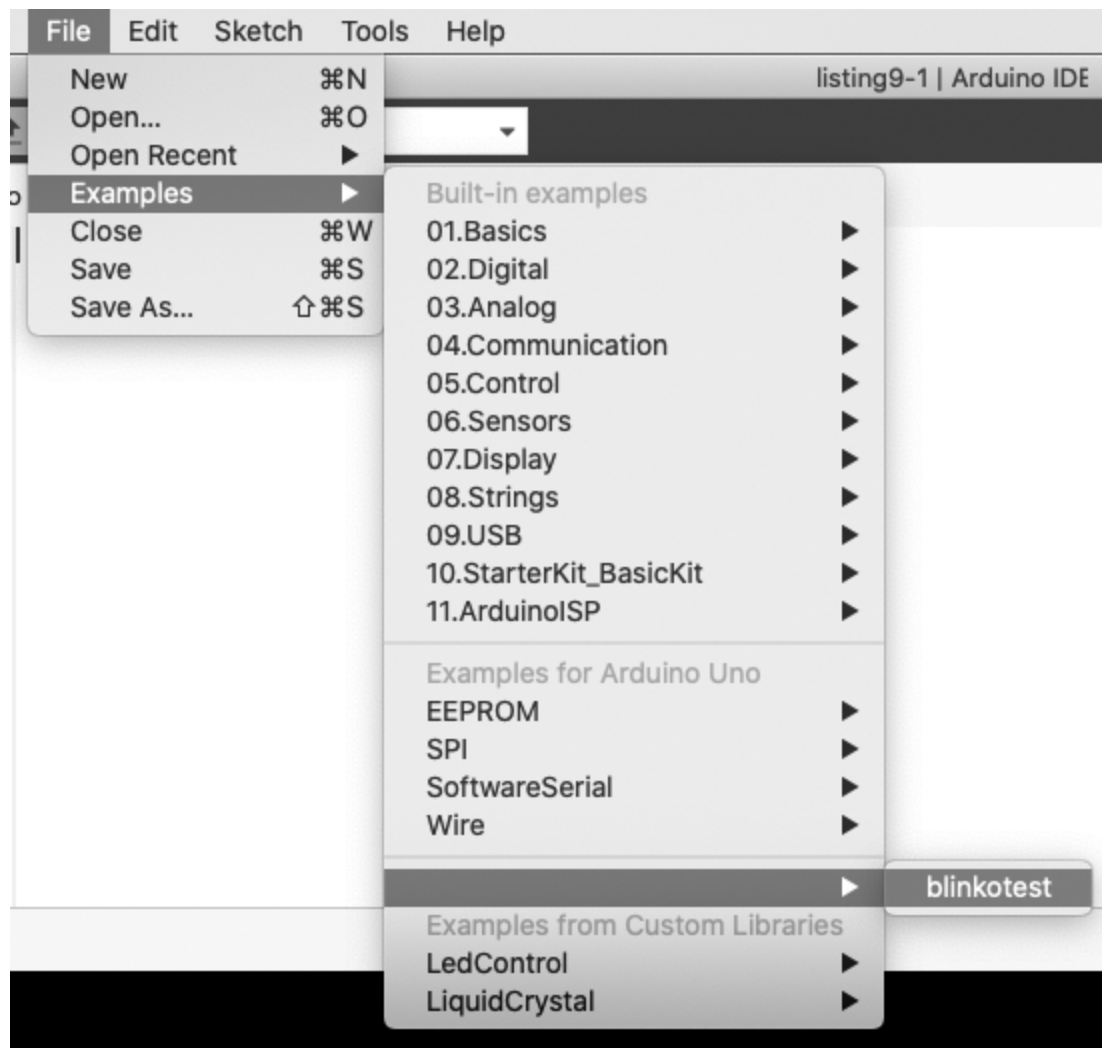


Figure 10-9: Our Arduino library example sketch is installed.

Creating a Library That Accepts Values to Perform a Function

Now that you have the knowledge to create a basic Arduino library, you can move on to the next level: creating a library that can accept values and act on them. Once again, we will look at an example function within a sketch and convert it into a more useful library.

Consider the sketch shown in [Listing 10-6](#). It uses the function `void blinkType()`, which tells the Arduino how many times to blink its onboard LED and the on/off period.

```
// Listing 10-6
void setup() {
    pinMode(13, OUTPUT); // use onboard LED
}

void blinkType(int blinks, int duration)
// blinks - number of times to blink the LED
// duration - blink duration in milliseconds
{
    for (int i = 0; i < blinks; i++)
    {
        digitalWrite(13, HIGH);
        delay(duration);
        digitalWrite(13, LOW);
        delay(duration);
    }
}

void loop()
{
    // blink LED 10 times, with 250 ms duration
    blinkType(10, 250);
    delay(1000);
    // blink LED three times, with 1 second duration
    blinkType(3, 1000);
    delay(1000);
}
```

***Listing 10-6:** Demonstration sketch for the `blinkType()` function*

As you can see, the function `void blinkType()` accepts two values and then acts on them. The first value is the number of times to turn the onboard LED on and off, and the second value is the delay time in milliseconds for each blink.

Let's turn this function into an Arduino library named `blinko2`. [Listing 10-7](#) shows the header file for this library.

```
// Listing 10-7
/*
    blinko2.h - Blinking the Arduino's onboard LED on D13
    Accepts number of blinks and on/off delay
*/

#ifndef blinko2_h
```



```

#define blinko2_h

#include "Arduino.h"

class blinko2
{
    public:
        blinko2();
        void blinkType(int blinks, int duration);
1    private:
        int blinks;
        int duration;
};
#endif

```

Listing 10-7: The blinko2 library header file

The header file maintains the same structure as the header file for the original blinko library. However, there is a new section at 1 called private. The variables declared in the private section are for internal use within the library and cannot be used by the greater Arduino sketch. You can see these variables in use within the library source file shown in [Listing 10-8](#).

```

// Listing 10-8
/*
    blinko2.cpp - Blinking the Arduino's onboard LED on D13
    Accepts number of blinks and on/off delay
*/

#include "Arduino.h"
#include "blinko2.h"

blinko2::blinko2()
{
1    pinMode(13, OUTPUT);
}

2 void blinko2::blinkType(3int blinks, 4int duration)
{
    for (int i=0; i<blinks; i++)
    {
        digitalWrite(13, HIGH);
        delay(duration);
        digitalWrite(13, LOW);
    }
}

```

```
    delay(duration);  
  }  
}
```

Listing 10-8: The blinko2 library source file

The source file for blinko2 maintains the same structure as the source file for the original blinko library.

We set digital pin 13 to an output at 1. At 2, we declare the function `blinkType()`, which accepts the number of times to blink at 3 and the delay time at 4. You can see this in operation via the example sketch for our library in [Listing 10-9](#).

```
// Listing 10-9  
#include <blinko2.h>  
  
blinko2 ArduinoLED;  
  
void setup() {}  
  
void loop()  
{  
  ArduinoLED.blinkType(3,250);  
  // blink LED three times, with a duration of 250 ms  
  delay(1000);  
  ArduinoLED.blinkType(10,1000);  
  // blink LED 10 times, with a duration of 1 second  
  delay(1000);  
}
```

Listing 10-9: An example sketch for our blinko2 library

Next, we need to create the keywords file for our new blinko2 library. Don't forget to use a tab and not spaces between the words. Here is our *KEYWORDS.TXT* file:

blinko2	KEYWORD1
blinkType	KEYWORD2

Now create your ZIP file and install the library using the methods described earlier in this chapter. Then open and run the blinko2 example sketch to

experience how it works.

Creating a Library That Processes and Displays Sensor Values

For our final example of an Arduino library, we'll revisit the Analog Devices TMP36 temperature sensor used in several of our earlier projects. Our ArduinoTMP36 example library will take the raw value from the TMP36 and display the temperature in both Celsius and Fahrenheit via the Serial Monitor.

First, connect your TMP36 to the Arduino by following the schematic shown in [*Figure 10-10*](#).

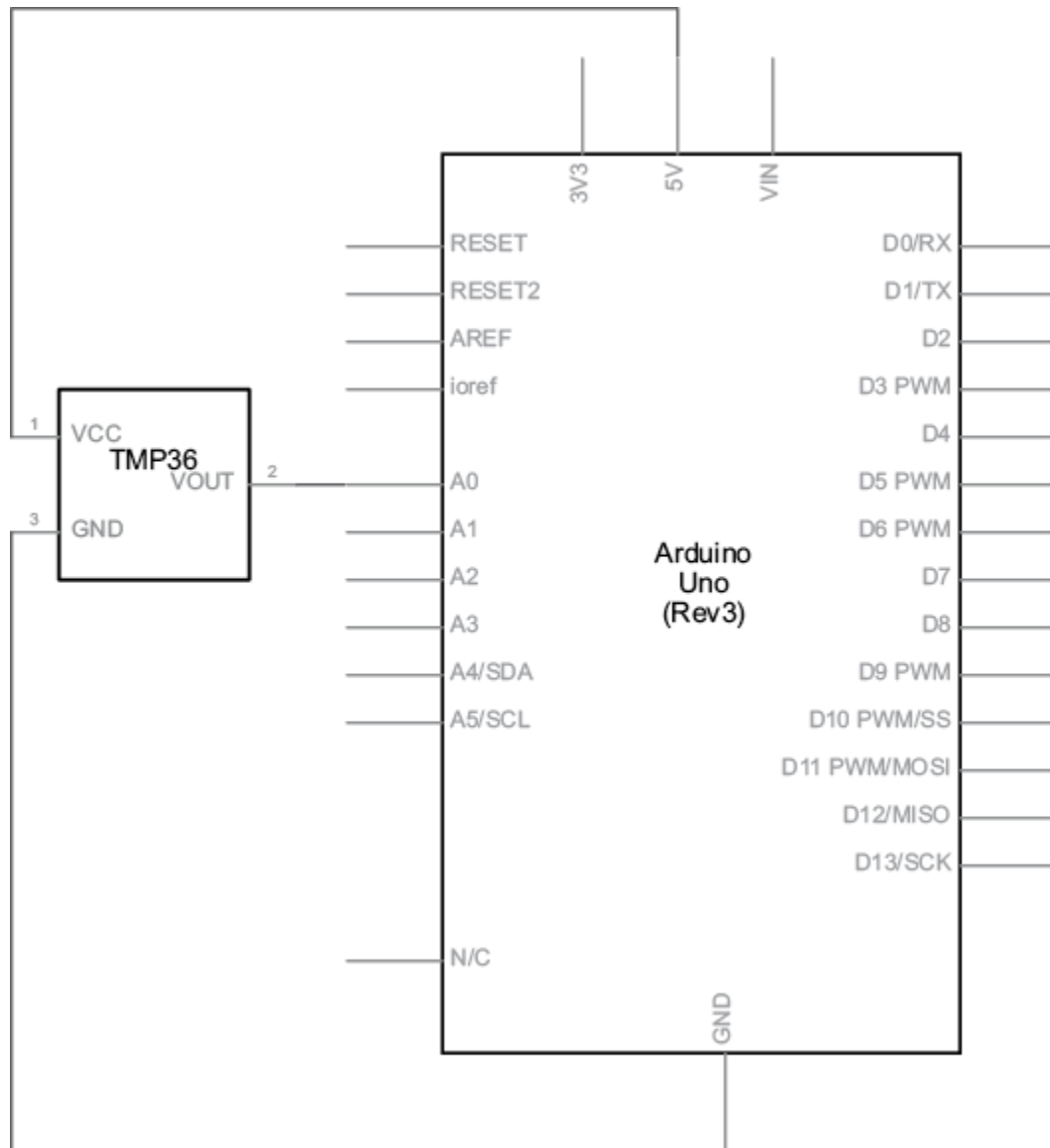


Figure 10-10: Schematic for use with the ArduinoTMP36 library

[Listing 10-10](#) is a sketch that we wish to turn into a library. It uses two functions, `readC()` and `readF()`, to take the raw reading from the TMP36 sensor via analog pin 0, convert it to degrees Celsius and Fahrenheit, and return the results.

```
// Listing 10-10
// display temperature from TMP36 sensor in C and F
float temperature;
float readC()
{
    float tempC;
```

```

    tempC = analogRead(0);
    tempC = tempC = (tempC * 5000) / 1024;
    tempC = tempC - 500;
    tempC = tempC / 10;
    return tempC;
}

float readF()
{
    float tempC;
    float tempF;
    tempC = analogRead(0);
    tempC = tempC = (tempC * 5000) / 1024;
    tempC = tempC - 500;
    tempC = tempC / 10;
    tempF = (tempC * 1.8) + 32;
    return tempF;
}

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

void loop()
{
    Serial.print("Temperature in Celsius is: ");
    temperature = readC();
    Serial.println(temperature);
    Serial.print("Temperature in Fahrenheit is: ");
    temperature = readF();
    Serial.println(temperature);
    delay(1000);
}

```

Listing 10-10: TMP36 demonstration sketch

The functions for temperature conversion are ideal candidates for inclusion in a library, which we will call ArduinoTMP36. The header file is shown in [Listing 10-11](#).

```

// Listing 10-11
1 #ifndef ArduinoTMP36_h
    #define ArduinoTMP36_h

    #include "Arduino.h"

```

```

class ArduinoTMP36
{
2   public:
    ArduinoTMP36();
    float readC();
    float readF();
3   private:
    float tempC;
    float tempF;
};

#endif

```

Listing 10-11: The ArduinoTMP36 library header file

At this point, you probably recognize the structure of the header file. We set up the definitions at 1. Inside the class at 2, we declare the public items, which include the constructor and the readC() and readF() functions. We also declare the private items at 3; these include the two variables used within the library.

Next we have the library source file, shown in [Listing 10-12](#).

```

// Listing 10-12
#include "Arduino.h"
#include "ArduinoTMP36.h"

ArduinoTMP36::ArduinoTMP36()
{
}

float ArduinoTMP36::readC()
{
    float tempC;
    tempC = analogRead(0);
    tempC = tempC*(5000)/1024;
    tempC = tempC-500;
    tempC = tempC/10;
    return tempC;
}

float ArduinoTMP36::readF()
{
    float tempC;
    float tempF;
}

```

```
tempC = analogRead(0);  
tempC = tempC*(tempC*5000)/1024;  
tempC = tempC-500;  
tempC = tempC/10;  
tempF = (tempC*1.8)+32;  
return tempF;  
}
```

Listing 10-12: The ArduinoTMP36 library source file

The source file contains the two functions used to calculate the temperatures. They are defined as `float` because they return a floating-point value. The temperatures are determined using the same formulas as in Project 8 in Chapter 4.

Finally, we need to create the keywords file for our new ArduinoTMP36 library. Don't forget to use a tab and not spaces between the words. Our *KEYWORDS.TXT* file is shown here:

ArduinoTMP36	KEYWORD1
readC	KEYWORD2
readF	KEYWORD2

Now create your ZIP file and install the library using the methods described earlier in this chapter. Then open and run the ArduinoTMP36 example sketch, shown in [Listing 10-13](#).

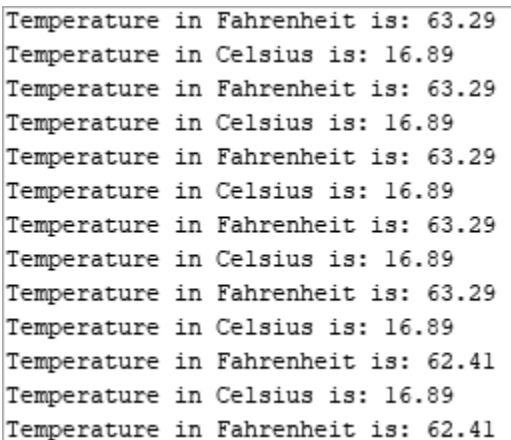
```
// Listing 10-13  
1 #include <ArduinoTMP36.h>  
  ArduinoTMP36 thermometer;  
  
2 float temperature;  
  
  void setup()  
  {  
    Serial.begin(9600);  
  }  
  
  void loop()  
  {  
    Serial.print("Temperature in Celsius is: ");  
3    temperature=thermometer.readC();  
    Serial.println(temperature);
```

```
    Serial.print("Temperature in Fahrenheit is: ");  
4    temperature=thermometer.readF();  
    Serial.println(temperature);  
  
    delay(1000);  
}
```

Listing 10-13: An example sketch for the ArduinoTMP36 library

Simply include the library and create the instance at 1. Then declare a variable to accept the output from the library at 2. After that, the temperature is requested and returned in Celsius and Fahrenheit at points 3 and 4, respectively.

Open the Serial Monitor window and set the data speed to 9,600 baud, and you should be presented with a scrolling updated list of the current temperature in Celsius and Fahrenheit, like that shown in [Figure 10-11](#).



```
Temperature in Fahrenheit is: 63.29  
Temperature in Celsius is: 16.89  
Temperature in Fahrenheit is: 63.29  
Temperature in Celsius is: 16.89  
Temperature in Fahrenheit is: 63.29  
Temperature in Celsius is: 16.89  
Temperature in Fahrenheit is: 63.29  
Temperature in Celsius is: 16.89  
Temperature in Fahrenheit is: 63.29  
Temperature in Celsius is: 16.89  
Temperature in Fahrenheit is: 62.41  
Temperature in Celsius is: 16.89  
Temperature in Fahrenheit is: 62.41
```

Figure 10-11: Example of output from the ArduinoTMP36 library

Now you can appreciate how much time and sketch size is saved by using a library instead of including the functions every time you create a new sketch.

Looking Ahead

Now that you have experience with writing Arduino libraries, you can create your own. This will help you to gain a deeper understanding of the libraries provided by other sources. You can also practice by creating libraries for the projects in this book you've already completed.

In the next chapter, you will learn how to work with user input entered via numeric keypads, so turn the page to get started.

11

NUMERIC KEYPADS

In this chapter you will

Learn how to connect numeric keypads to your Arduino

Read values from the keypad in a sketch

Expand on decision systems with the `switch case` statement

Create a PIN-controlled lock or switch

Using a Numeric Keypad

As your projects become more involved, you might want to accept numeric input from users when your Arduino isn't connected to a device with a keyboard. For example, you might like the ability to turn something on or off by entering a secret number. One option would be to wire up 10 or more push buttons to various digital input pins (for the numbers 0 through 9), but it's much easier to use a numeric keypad like the one shown in [*Figure 11-1*](#).



Figure 11-1: A numeric keypad

One of the benefits of using a keypad is that it uses only 8 pins for 16 active buttons, and with the use of a clever Arduino library, you won't need to add pull-down resistors for debouncing as we did in Chapter 4.

At this point, you will need to download and install the Arduino Keypad library, which is available from <https://github.com/Chris--A/Keypad/archive/master.zip>.

Wiring a Keypad

Physically wiring the keypad to the Arduino is easy. With the keypad facing up, take a look at the end of the ribbon cable. You'll see eight female connectors in a row, as shown in [Figure 11-2](#).



Figure 11-2: The keypad connector

Reading from left to right, the sockets are numbered from 8 to 1. For all the keypad projects in this book, you'll plug the keypad pins into the Arduino pins as shown in [Table 11-1](#).

Table 11-1: Keypad-to-Arduino Connections

Keypad pin number	Arduino pin
8	Digital 9
7	Digital 8
6	Digital 7
5	Digital 6
4	Digital 5
3	Digital 4
2	Digital 3
1	Digital 2

Programming for the Keypad

When you write a sketch for the keypad, you must include certain lines of code to enable the keypad, as identified in [Listing 11-1](#). The required code

starts at 1 and ends at 5.

```
// Listing 11-1
1 // Beginning of keypad configuration code
  #include <Keypad.h>

  const byte ROWS = 4; // set display to four rows
  const byte COLS = 4; // set display to four columns

2 char keys[ROWS][COLS] = {
    {'1','2','3','A'},
    {'4','5','6','B'},
    {'7','8','9','C'},
    {'*','0','#','D'}
  };
3 byte rowPins[ROWS] = {9, 8, 7, 6};
4 byte colPins[COLS] = {5, 4, 3, 2};

  Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins,
    ROWS, COLS );
5 // End of keypad configuration code

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

void loop(){
  char key = keypad.getKey();

  if (key){
    Serial.print(key);
  }
}
```

Listing 11-1: Numeric keypad demonstration sketch

At 2, we introduce `keys`, a char variable array that contains one or more letters, numbers, or symbols that can be generated with a computer keyboard. In this case, it contains the numbers and symbols that your Arduino can expect from the keypad.

The lines of code at 3 and 4 define which digital pins are used on the Arduino. Using these lines and [Table 11-1](#), you can change the digital pins used for input if you want.

Testing the Sketch

After uploading the sketch, open the Serial Monitor and press some keys on the keypad. The characters for the keys you pressed will be displayed in the Serial Monitor, as shown in [Figure 11-3](#).

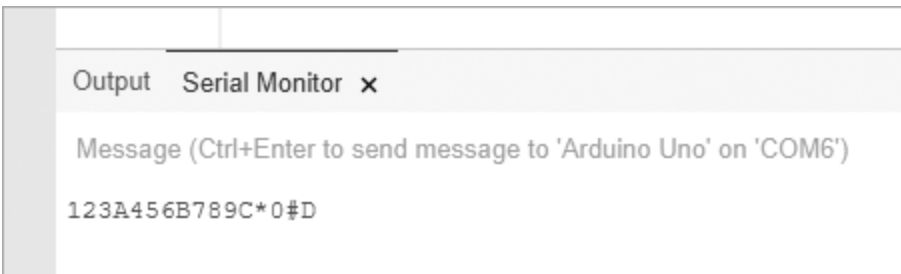


Figure 11-3: The result of pressing keys on the keypad

Making Decisions with switch case

When you need to compare two or more variables against another value, you'll often find it easier and neater to use a switch case statement instead of an if then statement, because switch case statements can make an indefinite number of comparisons and run code when the comparison returns true. For example, if we had the integer variable `xx` with a possible value of 1, 2, or 3 and we wanted to run different code based on whether the value was 1, 2, or 3, we could use code like the following to replace our if then statement:

```
switch(xx)
{
  case 1:
    // do something when the value of xx is 1
    break; // finish and move on with sketch
  case 2:
    // do something when the value of xx is 2
    break;
  case 3:
    // do something when the value of xx is 3
```

```
break;
default:
  // do something if xx is not 1, 2 or 3
  // default is optional
}
```

The optional `default:` section at the end of this code segment lets you choose to run some code when true comparisons no longer exist in the `switch case` statement.

Project #32: Creating a Keypad-Controlled Lock

In this project, we'll start to create a keypad-controlled lock. We'll use the basic setup described in the sketch in [Listing 11-1](#), but we'll also include a six-digit secret code that a user needs to enter on the keypad. The Serial Monitor will tell the user whether the code they've input is correct or not.

The secret code is stored in the sketch but is not displayed to the user. The sketch will call different functions depending on whether the input code (PIN) is correct. To activate and deactivate the lock, the user must press `*` and then the secret number, followed by `#`.

The Sketch

Enter and upload this sketch:

```
// Project 32 - Creating a Keypad-Controlled Lock
// Beginning of necessary code
#include <Keypad.h>

const byte ROWS = 4; // set display to four rows
const byte COLS = 4; // set display to four columns

char keys[ROWS][COLS] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};
byte rowPins[ROWS] = {9, 8, 7, 6};
```

```

byte colPins[COLS] = {5, 4, 3, 2};

Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins,
ROWS, COLS );

// End of necessary code
1 char PIN[6]={'1','2','3','4','5','6'}; // our secret number
  char attempt[6]={0,0,0,0,0,0};
  int z=0;
  void setup()
  {
    Serial.begin(9600);
  }
  void correctPIN() // do this if the correct PIN is entered
  {
    Serial.println("Correct PIN entered...");
  }
  void incorrectPIN() // do this if an incorrect PIN is entered
  {
    Serial.println("Incorrect PIN entered!");
  }
  void checkPIN()
  {
    int correct=0;

2    for (int i = 0;
      i < 6 ;
      i++ )
    {
      // Goes step-by-step through the 6-character array.
      // If each char matches each char in the PIN, increments the
      // counter.
      if (attempt[i]==PIN[i])
      {
        correct++;
      }
      if (correct==6)
      {
3        correctPIN();
      }
      else
      {
4        incorrectPIN();
      }
      for (int i=0; i<6; i++) // removes previously entered code

```



```

    attempt
    {
        attempt[i]=0;
    }
}
void readKeypad()
{
    char key = keypad.getKey();
    if (key != NO_KEY)
    {
5      switch(key)
        {
            case '*':
                z=0;
                break;
            case '#':
                delay(100); // removes the possibility of switch bounce
                checkPIN();
                break;
            default:
                attempt[z]=key;
                z++;
        }
    }
}
void loop()
{
6  readKeypad();
}

```

Understanding the Sketch

After the usual setup routines (as described in [Listing 11-1](#)), the sketch continually “listens” to the keypad by running the function `readKeypad()` at 6. After a key is pressed, the Arduino examines the value of the key using a switch case statement at 5. The Arduino stores the values of the keys pressed on the keypad in the array `attempt`, and when the user presses #, the Arduino calls the function `checkPIN()`.

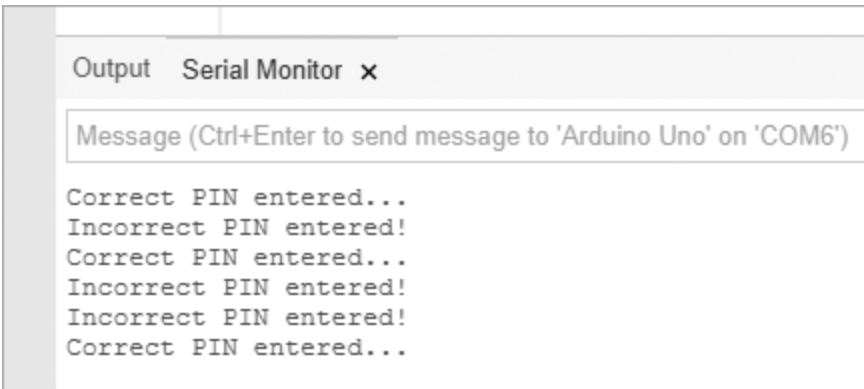
At 2, the Arduino compares the values of the pressed keys against the PIN stored in the array `PIN` at 1. If the correct sequence is entered, the function `correctPIN()` at 3 is called, where you can add your own code to execute. If an incorrect sequence is entered, the function `incorrectPIN()` at 4 is

called. Finally, once the user's entry has been checked, the code deletes the entry from memory so the code is ready for the next test.

Testing the Sketch

After you've uploaded the sketch to your Arduino, open the Serial Monitor window, press star (*) on the numeric keypad, type the secret number, and then enter the pound sign (#). Try entering both correct and incorrect numbers. Your results should be similar to the output shown in [Figure 11-4](#).

This example serves as a perfect foundation for your own PIN-activated devices, such as locks, alarms, or anything else you can imagine. Just be sure to replace the code in `correctPIN()` and `incorrectPIN()` with the code you want to run when a correct or incorrect sequence is entered.



[Figure 11-4](#): Results from entering correct and incorrect PINs

Looking Ahead

You have learned yet another way to gather input for your Arduino. You've also gained the foundational knowledge to create a useful method of controlling a sketch using a numeric keypad, as well as the foundations for a combination lock to access anything that your Arduino can control. Furthermore, you've learned the very useful `switch case` statement. In the next chapter, you'll learn about another form of input: the touchscreen.

12

ACCEPTING USER INPUT WITH TOUCHSCREENS

In this chapter you will

- Learn how to connect a resistive touchscreen to your Arduino

- Discover the values that can be returned from the touchscreen

- Create a simple on/off touch switch

- Learn how to use the `map()` function

- Create an on/off touch switch with a dimmer-style control

We see touchscreens everywhere today: on smartphones, tablets, and even portable video game systems. So why not use a touchscreen to accept input from an Arduino user?

Touchscreens

Touchscreens can be quite expensive, but we'll use an inexpensive model available from Adafruit (part numbers 333 and 3575), originally designed for the Nintendo DS game console.

This touchscreen, which measures about 2.45 by 3 inches, is shown in [*Figure 12-1*](#).

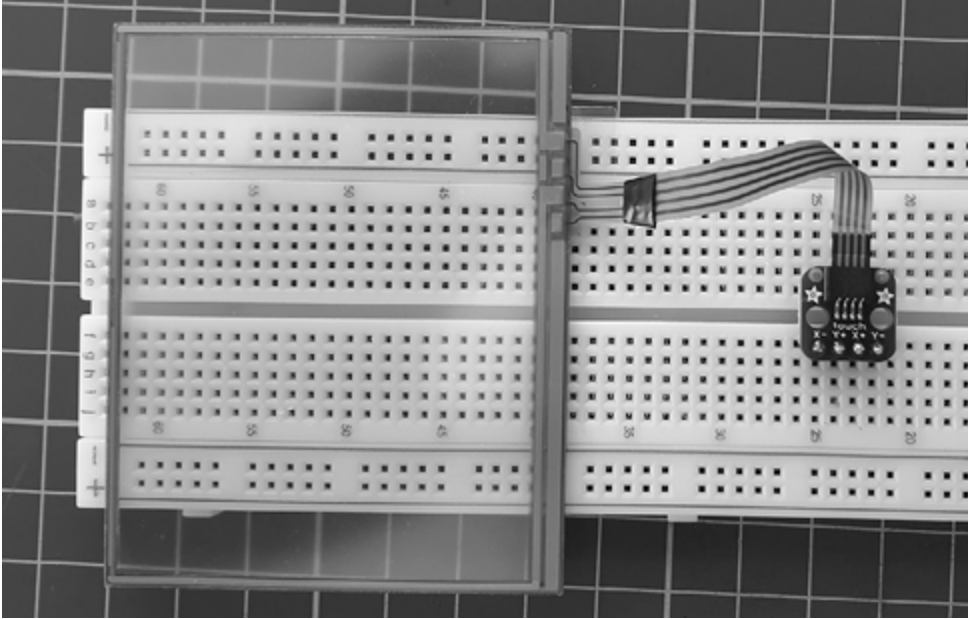


Figure 12-1: A touchscreen mounted on a solderless breadboard

Notice the horizontal ribbon cable connected to the small circuit board on the right. This *breakout board* is used to attach the Arduino and the breadboard to the touchscreen. The header pins included with the breakout board will need to be soldered before use. [Figure 12-2](#) shows a close-up of the breakout board.

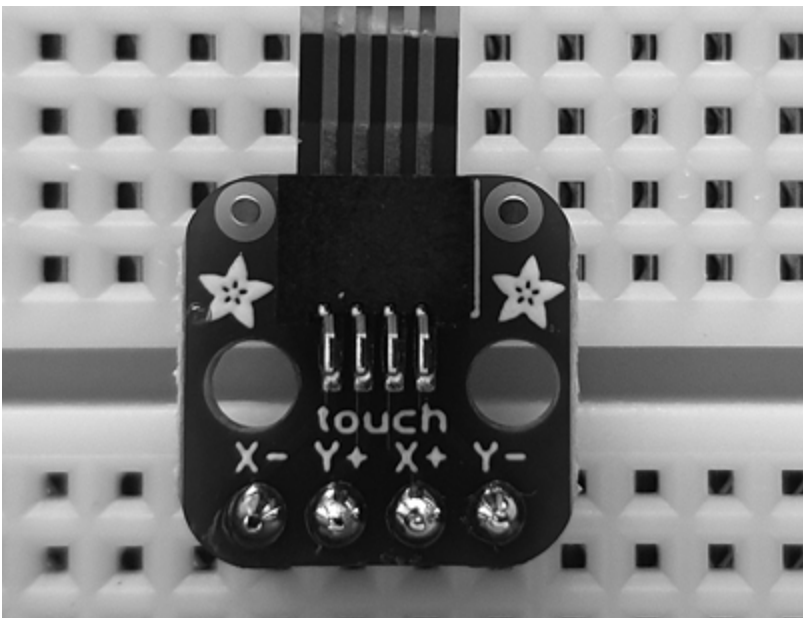


Figure 12-2: The touchscreen breakout board

Connecting the Touchscreen

Connect the touchscreen breakout board to an Arduino as shown in [Table 12-1](#).

Table 12-1: Touchscreen Breakout Board Connections

Breakout board pin	Arduino pin
X-	A3
Y+	A2
X+	A1
Y-	A0

Project #33: Addressing Areas on the Touchscreen

The touchscreen has two layers of resistive coating between the top layer of plastic film and the bottom layer of glass. One coating acts as the x-axis, and the other is the y-axis. As current passes through each coating, the resistance of the coating varies depending on where it has been touched; when the current is measured, the x and y positions of the touched area can be determined.

In this project, we'll use the Arduino to record touched locations on the screen. We'll also have it convert information from the touches into integers that represent areas of the screen.

The Hardware

The following hardware is required:

One Adafruit touchscreen, part 333

One Adafruit breakout board, part 3575

Male-to-male jumper wires

One solderless breadboard

Arduino and USB cable

Connect the touchscreen as described in [Table 12-1](#) and connect the Arduino to the PC via the USB cable.

The Sketch

Enter and upload the following sketch:

```
// Project 33 - Addressing Areas on the Touchscreen
int x,y = 0;
1 int readX() // returns the value of the touchscreen's x-axis
{
    int xr=0;
    pinMode(A0, INPUT);
    pinMode(A1, OUTPUT);
    pinMode(A2, INPUT);
    pinMode(A3, OUTPUT);
    digitalWrite(A1, LOW); // set A1 to GND
    digitalWrite(A3, HIGH); // set A3 as 5V
    delay(5);
    xr=analogRead(0);      // stores the value of the x-axis
    return xr;
}

2 int readY() // returns the value of the touchscreen's y-axis
{
    int yr=0;
    pinMode(A0, OUTPUT);
    pinMode(A1, INPUT);
    pinMode(A2, OUTPUT);
    pinMode(A3, INPUT);
    digitalWrite(14, LOW); // set A0 to GND
    digitalWrite(16, HIGH); // set A2 as 5V
    delay(5);
    yr=analogRead(1);      // stores the value of the y-axis
    return yr;
}

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

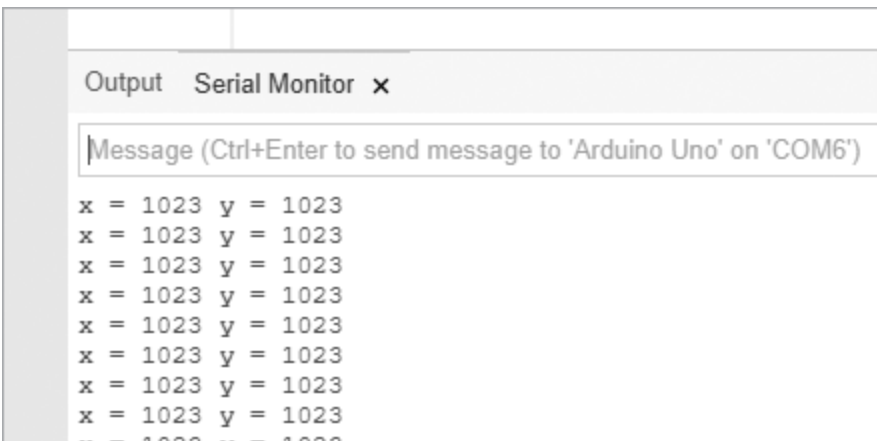
void loop()
{
    Serial.print(" x = ");
```

```
    x=readX();  
3  Serial.print(x);  
    y=readY();  
    Serial.print(" y = ");  
4  Serial.println(y);  
    delay (200);  
}
```

The functions `readX()` and `readY()` at 1 and 2 read the current from the touchscreen's resistive layers, measure it using `analogRead()`, and return the read values. The sketch rapidly runs these two functions to provide the real-time position of the screen area being touched and displays this information in the Serial Monitor at 3 and 4. (The `delay(5)` in each function is required to allow the input/output pins time to change their states.)

Testing the Sketch

To test the sketch, watch the Serial Monitor window while you touch the screen and notice how the x and y values change as you move your finger around the screen. Also take note of the values displayed when the screen is not being touched, as shown in [Figure 12-3](#).



[Figure 12-3](#): Values that appear when the touchscreen is not touched

You can use the values that display when you're not touching the screen in your sketch to detect when the screen is not being touched. Also, displays may vary slightly, so it is important to map out your own unit so you have an understanding of its display boundaries.

Mapping the Touchscreen

You can plot the coordinates for each corner of your touchscreen by touching the corners of the screen and recording the values returned, as shown in [Figure 12-4](#).

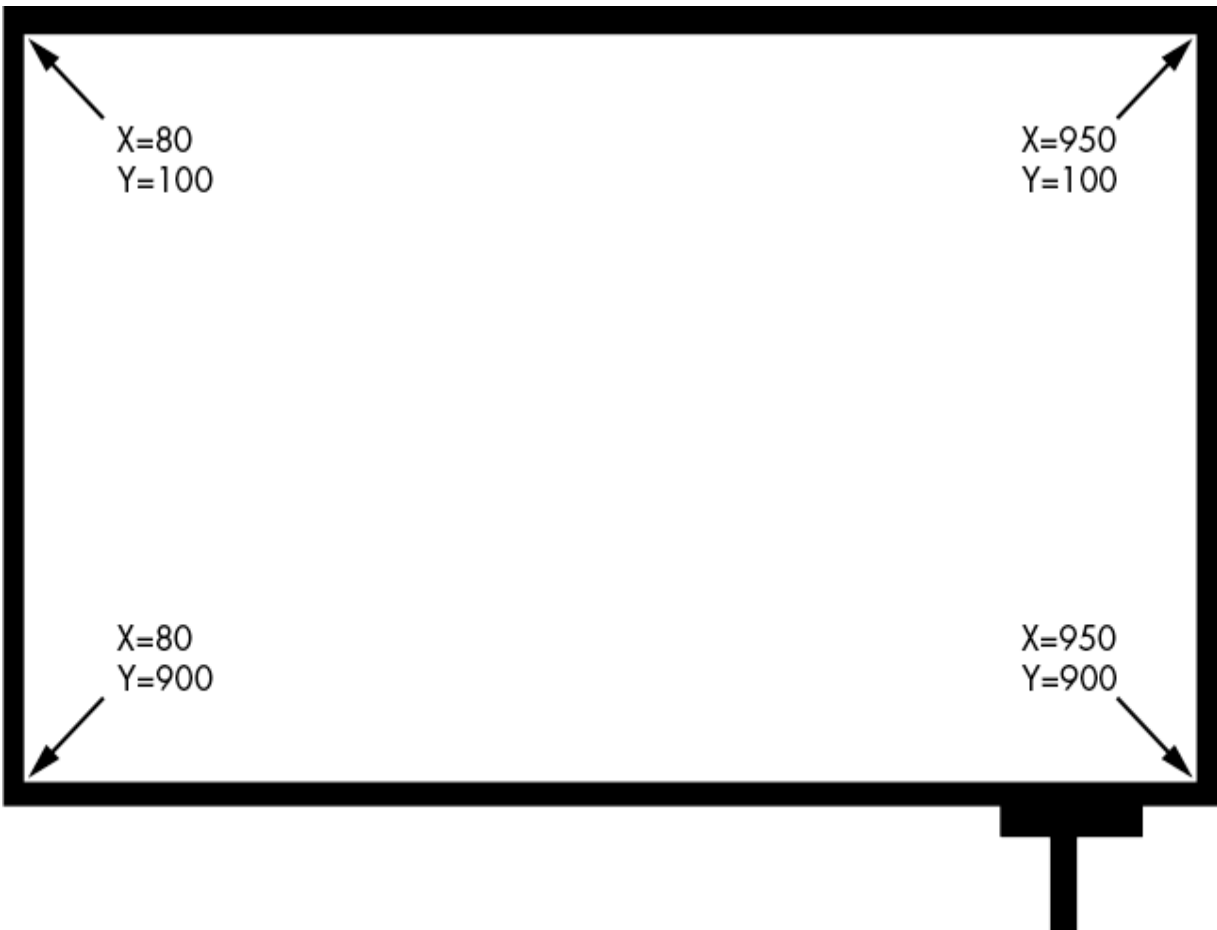


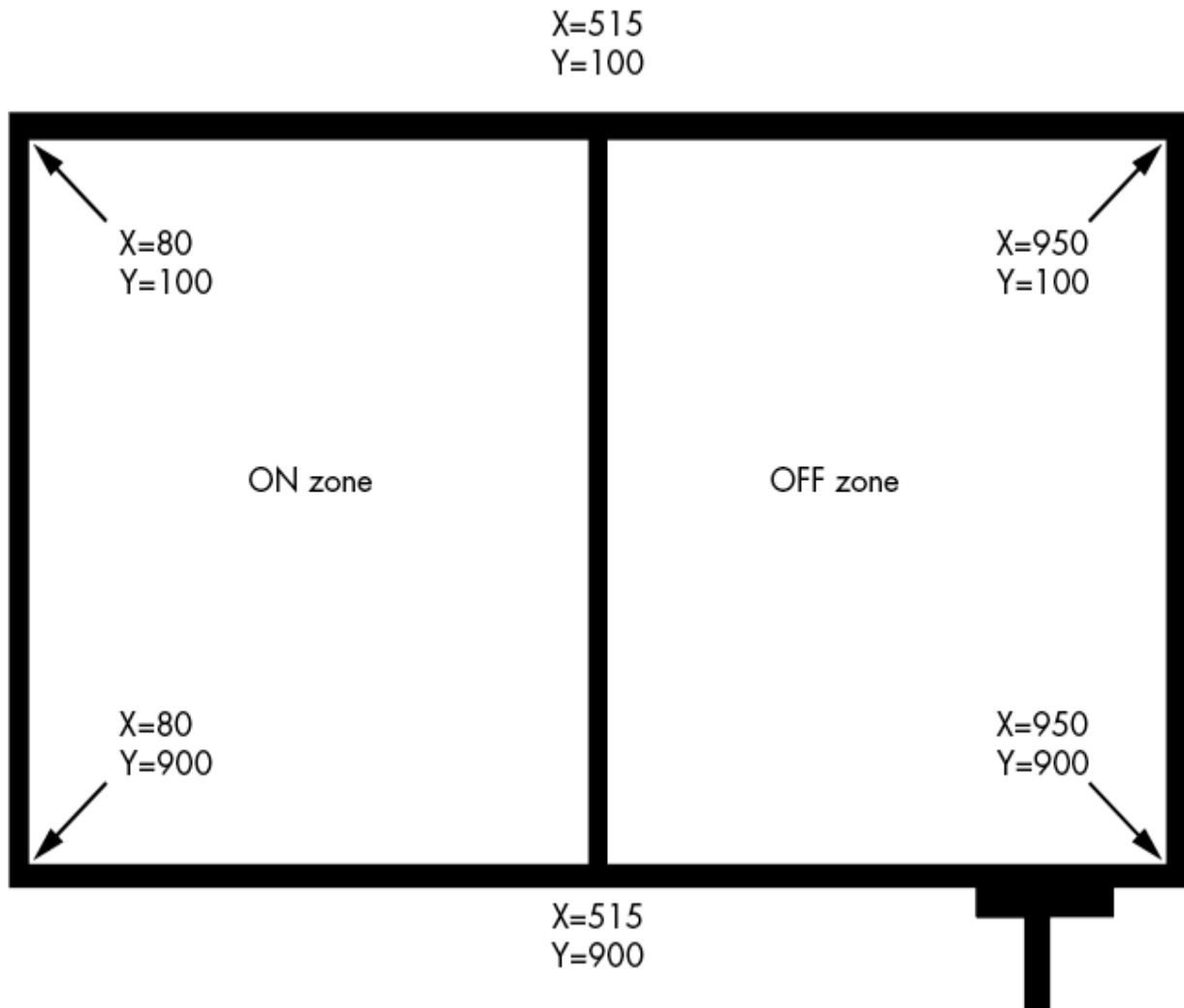
Figure 12-4: A touchscreen map

After you've created your touchscreen map, you can mathematically divide it into smaller regions, which you can then use with `if` statements to cause specific actions to occur depending on where the screen is touched. We'll do that in Project 34.

Project #34: Creating a Two-Zone On/Off Touch Switch

In this project, we'll use our touchscreen map to create an on/off switch. Start by dividing the touchscreen in half vertically, as shown in [Figure 12-5](#).

The Arduino will determine which zone of the screen was touched by comparing the recorded coordinates of the touch to the boundaries of each half of the screen. When the zone has been determined, the code responds by returning on or off (though it could also send an on or off signal to a device).



[Figure 12-5](#): On/off switch map

The Sketch

Enter and upload the following sketch:

```
// Project 34 - Creating a Two-Zone On/Off Touch Switch
int x,y = 0;
void setup()
{
  Serial.begin(9600);
  pinMode(10, OUTPUT);
}

void switchOn()
{
  digitalWrite(10, HIGH);
  Serial.print("Turned ON at X = ");
  Serial.print(x);
  Serial.print(" Y = ");
  Serial.println(y);
  delay(200);
}

void switchOff()
{
  digitalWrite(10, LOW);
  Serial.print("Turned OFF at X = ");
  Serial.print(x);
  Serial.print(" Y = ");
  Serial.println(y);
  delay(200);
}

int readX() // returns the value of the touchscreen's x-axis
{
  int xr=0;
  pinMode(A0, INPUT);
  pinMode(A1, OUTPUT);
  pinMode(A2, INPUT);
  pinMode(A3, OUTPUT);
  digitalWrite(A1, LOW); // set A1 to GND
  digitalWrite(A3, HIGH); // set A3 as 5V
  delay(5);
  xr=analogRead(0);
  return xr;
}

int readY() // returns the value of the touchscreen's y-axis
{
  int yr=0;
  pinMode(A0, OUTPUT);
  pinMode(A1, INPUT);
}
```

```

    pinMode(A2, OUTPUT);
    pinMode(A3, INPUT);
    digitalWrite(A0, LOW); // set A0 to GND
    digitalWrite(A2, HIGH); // set A2 as 5V
    delay(5);
    yr=analogRead(1);
    return yr;
}

void loop()
{
    x=readX();
    y=readY();
1    // test for ON
    if (x<=515 && x>=80)
    {
        switchOn();
    }
2    // test for OFF
    if (x<950 && x>=516)
    {
        switchOff();
    }
}

```

Understanding the Sketch

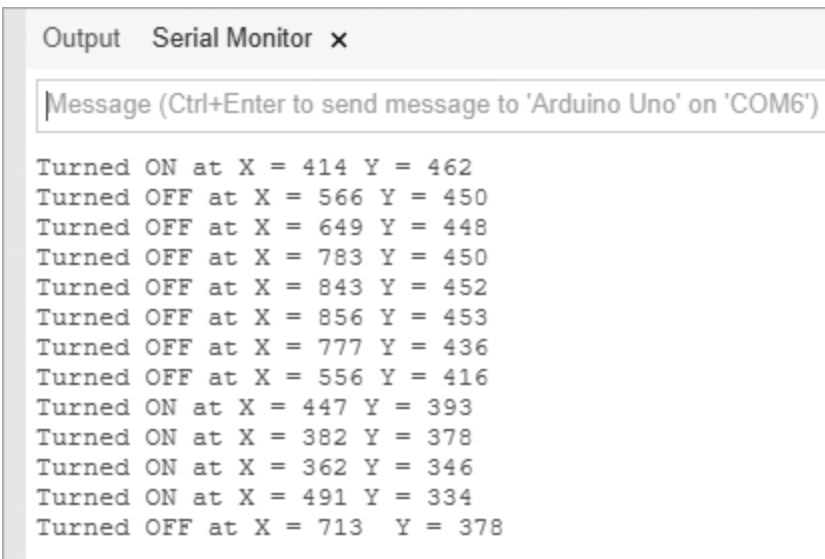
The two `if` statements used in `void loop()` check for a touch on the left or right side of the screen. If the left side is touched, the touch is detected as an “on” press at 1. If the right side is touched (an “off” press), the touch is detected at 2.

NOTE

The y-axis is ignored because the touchscreen is split vertically, so it doesn't matter whether you touch the screen higher up or lower down. If we were to create horizontal boundaries, the y-axis would need to be checked as well, as you'll see in Project 35.

Testing the Sketch

The output of this sketch is shown in [Figure 12-6](#). The status of the switch and the coordinates are shown after each screen touch.



```
Output Serial Monitor x
Message (Ctrl+Enter to send message to 'Arduino Uno' on 'COM6')
Turned ON at X = 414 Y = 462
Turned OFF at X = 566 Y = 450
Turned OFF at X = 649 Y = 448
Turned OFF at X = 783 Y = 450
Turned OFF at X = 843 Y = 452
Turned OFF at X = 856 Y = 453
Turned OFF at X = 777 Y = 436
Turned OFF at X = 556 Y = 416
Turned ON at X = 447 Y = 393
Turned ON at X = 382 Y = 378
Turned ON at X = 362 Y = 346
Turned ON at X = 491 Y = 334
Turned OFF at X = 713 Y = 378
```

Figure 12-6: Output from Project 34

Using the map() Function

There may come a time when you need to convert an integer that falls within one range into a value that falls into another range. For example, the x values of your touchscreen might run from 100 to 900, but you might have to translate that to a range of 0 to 255 to control an 8-bit output.

To do this we use the `map()` function, which is laid out as:

```
map(value, fromLow, fromHigh, toLow, toHigh);
```

For example, to translate 450 on the touchscreen to the range 0–255, you would use this code:

```
x = map(450, 100, 900, 0, 255);
```

This would give x a value of 95. You'll use the `map()` function in Project 35.

Project #35: Creating a Three-Zone Touch Switch

In this project, we'll create a three-zone touch switch for an LED on digital pin 3 that turns the LED on or off and adjusts the brightness from 0 to 255 using PWM (as explained in Chapter 3).

The Touchscreen Map

Our touchscreen map is shown in [Figure 12-7](#).

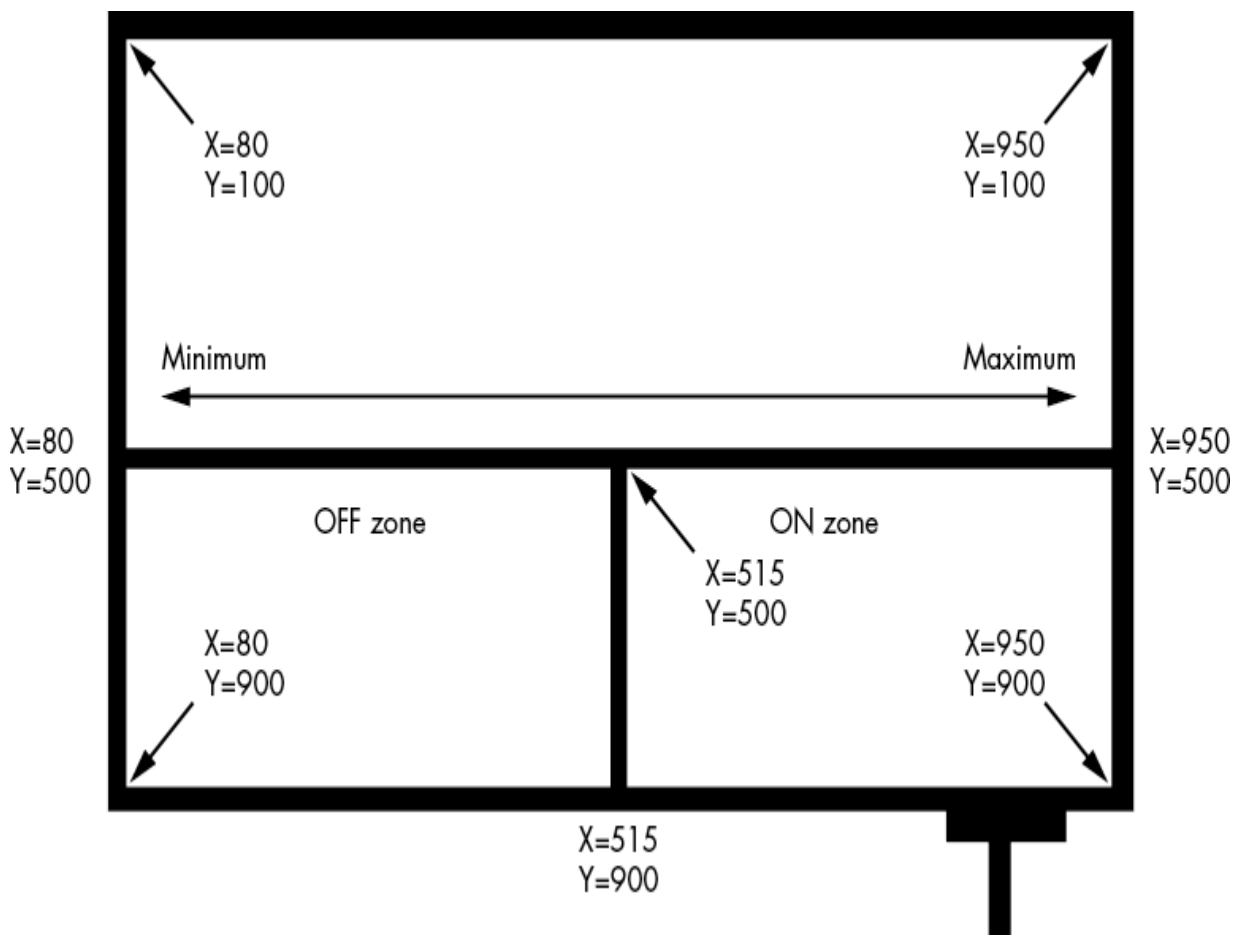


Figure 12-7: Touchscreen map for a three-zone touch switch

The touchscreen map is divided into off and on zones and a brightness control zone. We measure the values returned by the touchscreen to determine which part has been touched, then react accordingly.

The Sketch

Enter and upload the following sketch:

```
// Project 35 - Creating a Three-Zone Touch Switch
int x,y = 0;
void setup()
{
  pinMode(3, OUTPUT);
  Serial.begin(9600);
}
void switchOn()
{
  digitalWrite(3, HIGH);
  delay(200);
}
void switchOff()
{
  digitalWrite(3, LOW);
  delay(200);
}
void setBrightness()
{
  int PWMvalue;
1  PWMvalue=map(x, 80, 950, 0, 255);
  analogWrite(3, PWMvalue);
}
int readX() // returns the value of x-axis
{
  int xr=0;
  pinMode(A0, INPUT);
  pinMode(A1, OUTPUT);
  pinMode(A2, INPUT);
  pinMode(A3, OUTPUT);
  digitalWrite(A1, LOW); // set A1 to GND
  digitalWrite(A3, HIGH); // set A3 as 5V
  delay(5);
  xr=analogRead(0);
  return xr;
}
int readY() // returns the value of y-axis
{
  int yr=0;
  pinMode(A0, OUTPUT);
  pinMode(A1, INPUT);
  pinMode(A2, OUTPUT);
  pinMode(A3, INPUT);
```

```

    digitalWrite(A0, LOW); // set A0 to GND
    digitalWrite(A2, HIGH); // set A2 as 5V
    delay(5);
    yr=analogRead(1);
    return yr;
}
void loop()
{
    x=readX();
    y=readY();
2  // test for ON
    if (x<=950 && x>=515 && y>= 500 && y>900)
    {
        switchOn();
    }
3  // test for OFF
    if (x>80 && x<515 && y>= 500 && y>900)
    {
        switchOff();
    }
    // test for brightness
4  if (y>=100 && y<=500)
    {
        setBrightness();
    }
    Serial.println(x);
}

```

Understanding the Sketch

Like the sketch for the two-zone map, this sketch will check for touches in the on and off zones (which are now smaller, because half the screen is reserved for the brightness zone) at 2 and 3 and for any touches above the horizontal divider, which we'll use to determine brightness, at 4. If the screen is touched in the brightness area, the position on the x-axis is converted to a relative value for PWM using the `map()` function at 1, and the LED is adjusted accordingly using the function `setBrightness()`.

You can use these same functions to create any number of switches or sliders with this simple and inexpensive touchscreen. Furthermore, you could create your own library to easily return X and Y values and control the brightness in any sketch you write in the future.

Looking Ahead

This chapter introduced you to the touchscreen, another way of accepting user data and controlling your Arduino. In the next chapter, we'll focus on the Arduino board itself, learn about some of the different versions available, and create our own version on a solderless breadboard.

13

MEET THE ARDUINO FAMILY

In this chapter you will

Learn how to build your own Arduino circuit on a solderless breadboard

Explore the features and benefits of a wide range of Arduino-compatible boards

Learn about open source hardware

We'll break down the Arduino design into a group of parts, and then you'll build your own Arduino circuit on a solderless breadboard. Building your own circuit can save you money, especially when you're working with changing projects and prototypes. You'll also learn about some new components and circuitry. Then we'll explore ways to upload sketches to your homemade Arduino that don't require extra hardware. Finally, we'll examine the more common alternatives to the Arduino Uno and explore their differences.

Project #36: Creating Your Own Breadboard Arduino

As your projects and experiments increase in complexity or number, the cost of purchasing Arduino boards for each task can easily get out of hand, especially if you like to work on more than one project at a time. At this point, it's cheaper and easier to integrate the circuitry of an Arduino board into your project by building an Arduino circuit on a solderless breadboard that you can then expand for your specific project. It should cost less than

\$10 in parts to reproduce the basic Arduino circuitry on a breadboard (which itself is usually reusable if you're not too hard on it). It's easier to make your own if your project has a lot of external circuitry, because it saves you running lots of wires from an Arduino back to the breadboard.

The Hardware

To build a minimalist Arduino, you'll need the following hardware:

One breadboard

Various connecting wires

One 7805 linear voltage regulator

One 16 MHz crystal oscillator

One ATmega328P-PU microcontroller with Arduino bootloader

One 1 μ F, 25 V electrolytic capacitor (C1)

One 100 μ F, 25 V electrolytic capacitor (C2)

Two 22 pF, 50 V ceramic capacitors (C3 and C4)

One 100 nF, 50 V ceramic capacitor (C5)

Two 560 Ω resistors (R1 and R2)

One 10 k Ω resistor (R3)

Two LEDs of your choice (LED1 and LED2)

One push button (S1)

One six-way header pin

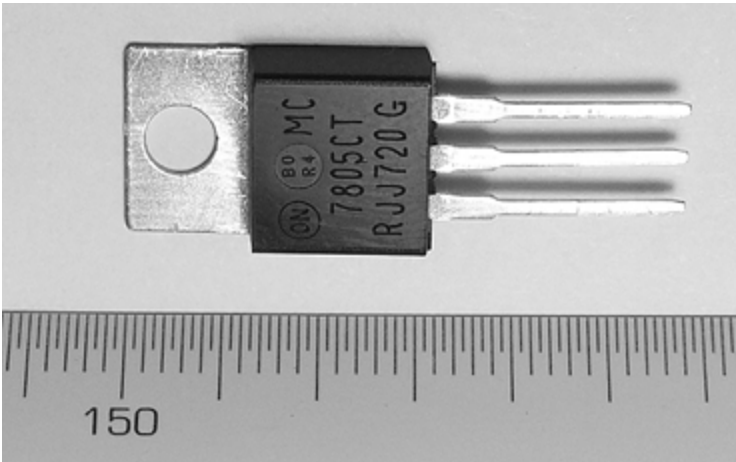
One PP3-type battery snap

One 9 V PP3-type battery

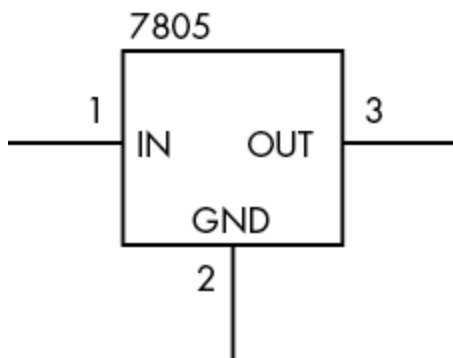
Some of these parts might be new to you. In the following sections, I'll explain each part and show you an example and a schematic of each.

7805 Linear Voltage Regulator

A *linear voltage regulator* contains a simple circuit that converts one voltage to another. The regulator included in the parts list is the 7805 type, which can convert a voltage between 7 and 30 V to a fixed 5 V, with a current up to 1 A—perfect for running our breadboard Arduino. [Figure 13-1](#) shows an example of a 7805 in a TO-220 package next to a ruler for scale.



[Figure 13-1](#): A 7805 linear voltage regulator with a ruler marked in millimeters

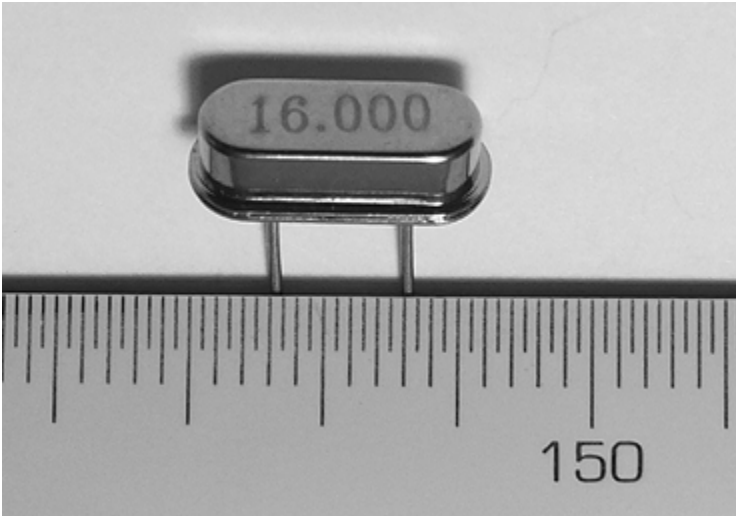


[Figure 13-2](#): 7805 schematic symbol

[Figure 13-2](#) shows the schematic symbol for the 7805. When you're looking at the labeled side of the 7805, the pin on the left (underneath the letter J) is for input voltage, the center pin connects to GND, and the right-hand pin (underneath the letter G) is the 5 V output connection. The metal tab at the top is drilled to allow it to connect to a larger piece of metal known as a *heat sink*. We use a heat sink when the circuit draws up to the maximum of 1 A of current, because the 7805 will become quite warm, like a hot coffee, at that level of use. The metal tab is also connected to GND. We will need one 7805 regulator for our example.

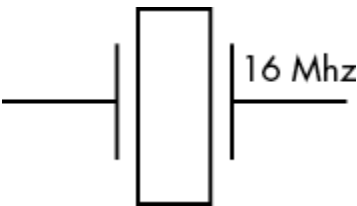
16 MHz Crystal Oscillator

More commonly known as simply a *crystal*, the crystal oscillator creates an electrical signal with a very accurate frequency. In this case, the frequency is 16 MHz. The crystal we'll use is shown in [Figure 13-3](#).



[Figure 13-3](#): A crystal oscillator with a ruler marked in millimeters

Compare this image to the crystal on your Arduino board. They should be identical in shape and size.



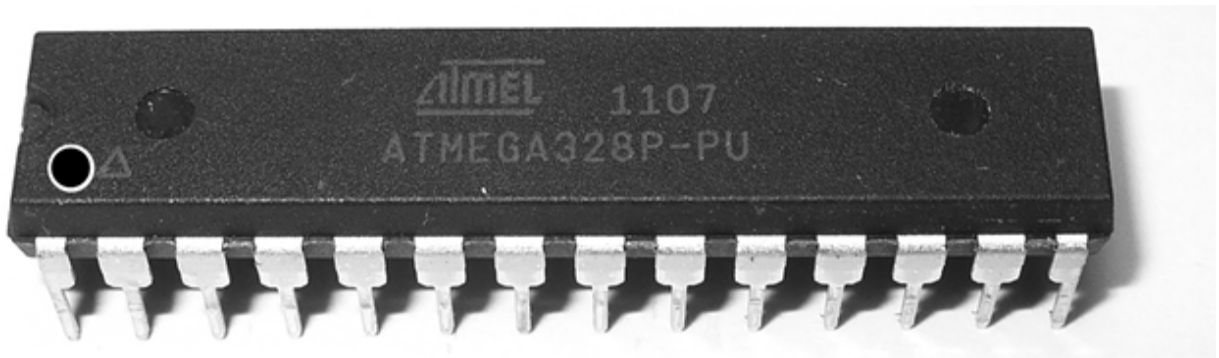
[Figure 13-4](#): Crystal oscillator schematic symbol

Crystals are not polarized. Their schematic symbol is shown in [Figure 13-4](#).

The crystal determines the microcontroller's speed of operation. For example, the microcontroller circuit we'll be assembling runs at 16 MHz, which means it can execute 16 million processor instructions per second. That doesn't mean it can execute a line of a sketch or a function that rapidly, however, since it takes many processor instructions to interpret a single line of code.

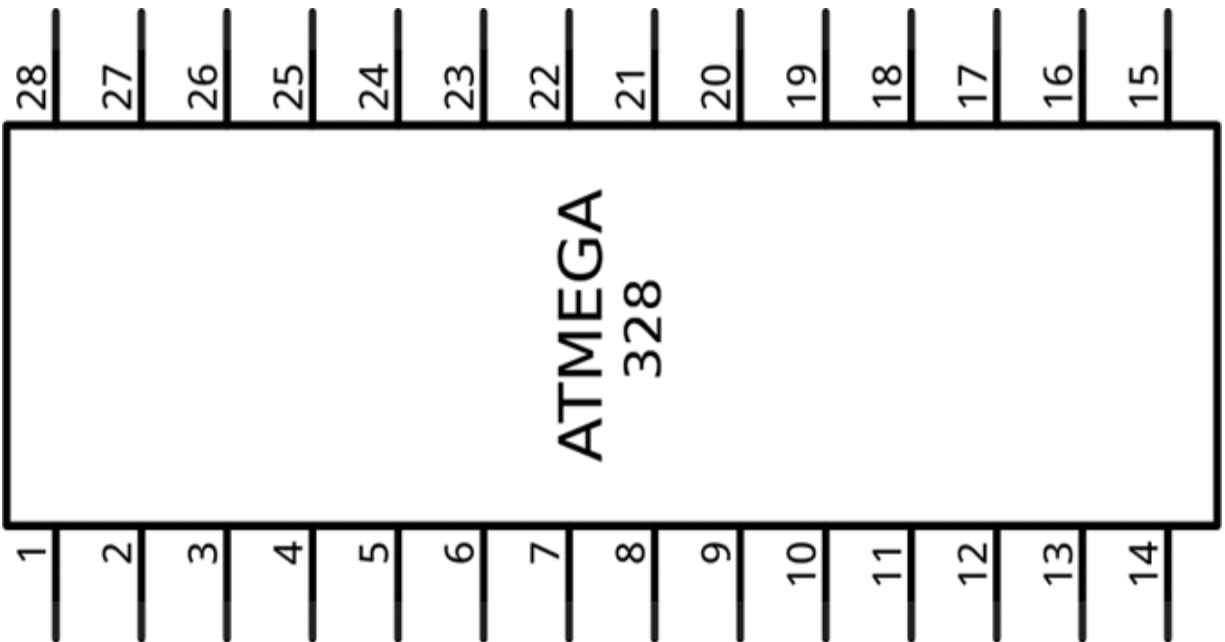
Atmel ATmega328P-PU Microcontroller IC

As noted in Chapter 2, a microcontroller is a tiny computer that is the brains of our breadboard Arduino. It contains a processor that executes instructions, various types of memory to hold data and instructions from our sketch, and various ways to send and receive data. An example of the ATmega328P-PU is shown in [Figure 13-5](#). When looking at the IC in the photo, notice that pin number 1 is at the bottom left of the IC and is marked by a small dot.



[Figure 13-5](#): An ATmega328P-PU

The schematic symbol for the microcontroller is shown in [Figure 13-6](#).

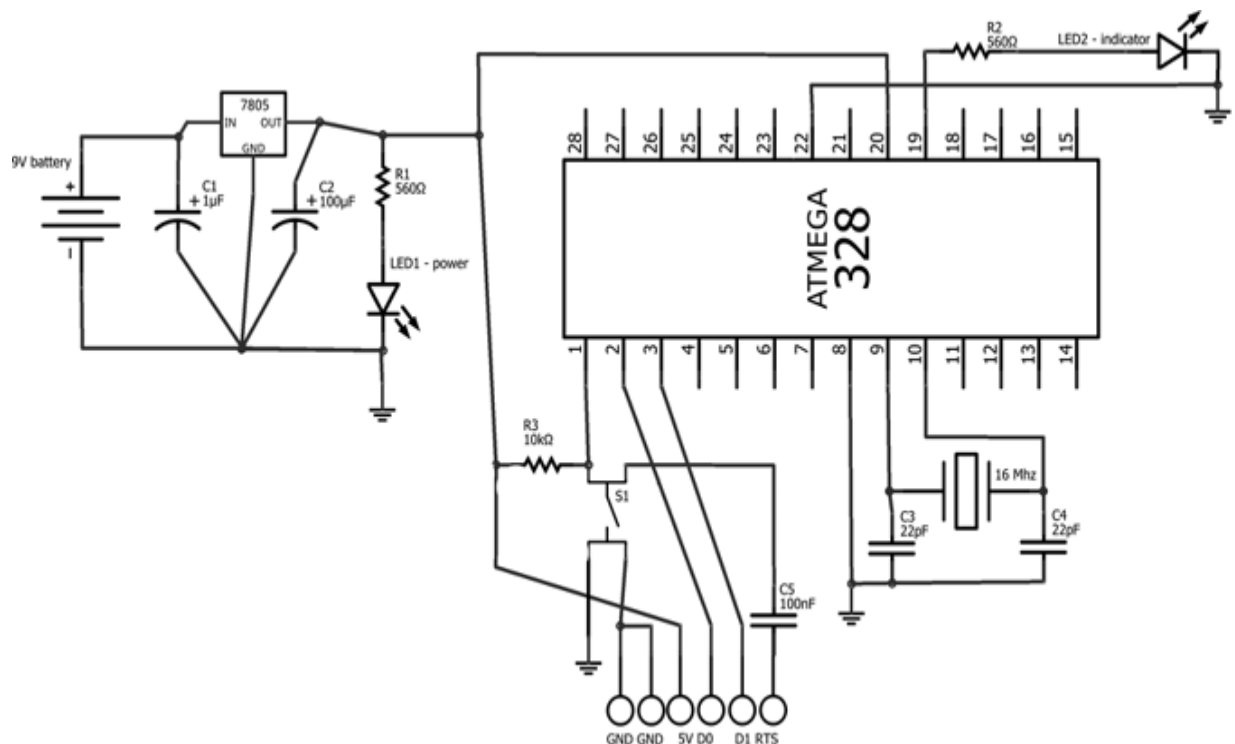


[Figure 13-6](#): Microcontroller schematic symbol

Not all microcontrollers contain the Arduino *bootloader*, the software that allows it to wait for the Arduino IDE to send it a new sketch to run. When choosing a microcontroller to include in a homemade Arduino, be sure to select one that already includes the bootloader. These are generally available from the same retailers that sell Arduino boards, such as Adafruit, PMD Way, and SparkFun.

The Schematic

[Figure 13-7](#) shows the circuit schematic.



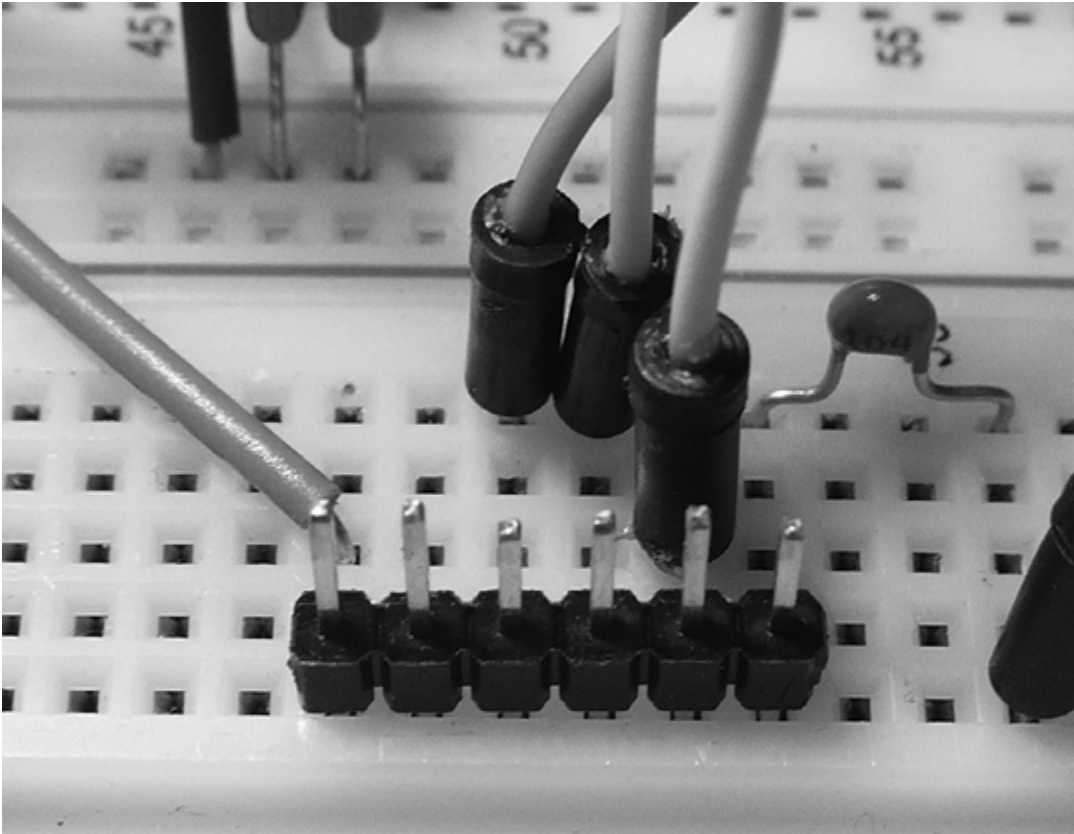
[Figure 13-7](#): Schematic for Project 36

The schematic contains two sections. The first, on the left, is the power supply, which reduces the voltage to a smooth 5 V. You'll see an LED that is lit when the power is on. The second section, on the right, consists of the microcontroller, the reset button, the programming pins, and another LED. This LED is wired to the ATmega328P-PU pin that is used as Arduino pin 13.

Use the schematic to wire up your Arduino. Don't forget to run the wires to the six-way pin header (shown in [Figure 13-8](#)), represented by the six

circles at the bottom of the schematic. We'll use this connection later in the chapter to upload a sketch to our homemade Arduino.

The circuit will be powered using a 9 V battery and matching snap connector, as shown in [Figure 13-9](#). Connect the red lead of the battery snap connector to the positive (+) point and the black lead to the negative (–) point on the left side of the circuit.



[Figure 13-8](#): The six-way pin header



Figure 13-9: A 9 V battery and snap connector

Identifying the Arduino Pins

Where are all the Arduino pins on our homemade Arduino? All the analog, digital, and other pins available on the normal Arduino board are also available in our breadboard version; you simply connect directly to the microcontroller.

In our breadboard Arduino, the R2 and LED2 are on digital pin 13. [Table 13-1](#) lists the Arduino pins on the left and the matching ATmega328P-PU pins on the right.

Table 13-1: Pins for ATmega328P-PU

Arduino pin name	ATmega328P-PU pin
RST	1
RX/D0	2
TX/D1	3
D2	4
D3	5
D4	6
(5 V only)	7
GND	8
D5	11
D6	12
D7	13
D8	14
D9	15
D10	16
D11	17
D12	18
D13	19
(5 V only)	20
AREF	21
GND	22
A0	23
A1	24
A2	25
A3	26
A4	27
A5	28

To avoid confusion, retailers such as Adafruit and Freetronics sell adhesive labels to place over the microcontroller, like those shown in [Figure 13-10](#) (order at <https://www.freetronics.com.au/collections/arduino/products/microcontroller-labels-arduino-pinout/>).

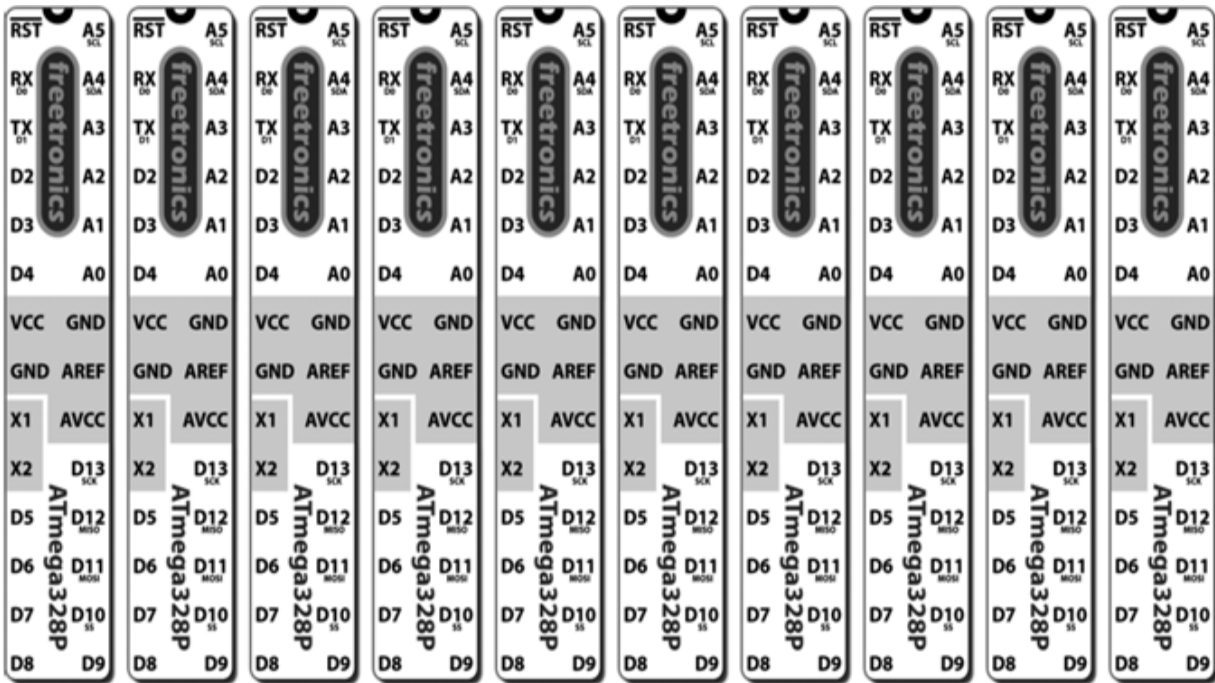


Figure 13-10: Pin labels

Running the Sketch

Now it's time to upload a sketch. We'll start by uploading a simple sketch to blink the LED:

```
// Project 36 - Creating Your Own Breadboard Arduino

void setup()
{
  pinMode(13, OUTPUT);
}

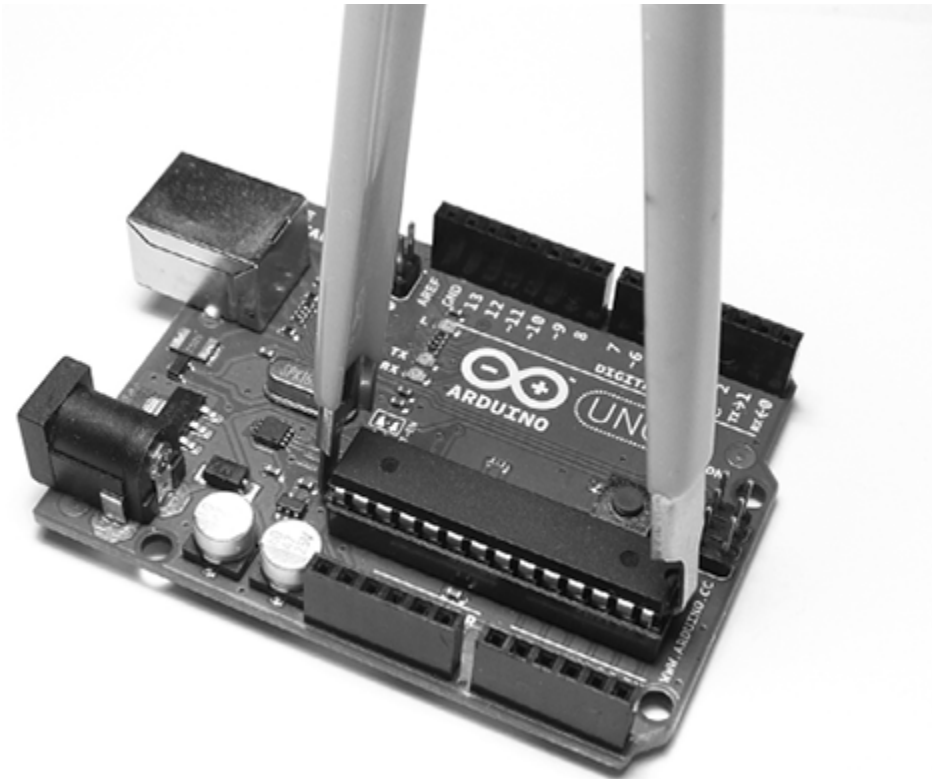
void loop()
{
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

You can upload the sketch in one of three ways.

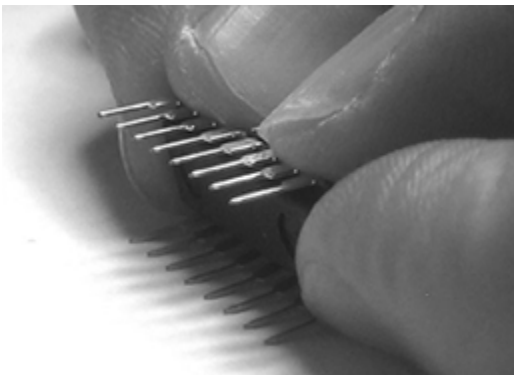
Using the Microcontroller Swap Method

The most inexpensive way to upload a sketch is to remove the microcontroller from an existing Arduino, insert the microcontroller from your homemade Arduino, upload the sketch, and then swap the microcontrollers again.

To remove a microcontroller from the Arduino safely, use an IC extractor, as shown in [Figure 13-11](#).



[Figure 13-11](#): Using an IC extractor to remove a microcontroller



[Figure 13-12](#): Bending the microcontroller pins

When removing the microcontroller, be sure to pull both ends out evenly and slowly *at the same time*—and take your time! Removing the component might be difficult, but eventually the microcontroller will come out.

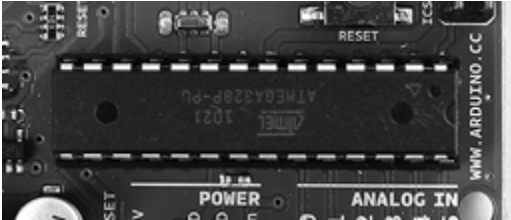


Figure 13-13: Correct orientation of the microcontroller in an Arduino

When inserting a microcontroller into the breadboard or your Arduino, you may have to bend the pins a little to make them perpendicular to the body of the microcontroller so that they can slide in easily. To do this, place one side of the component against a flat surface and gently push down; then repeat on the other side, as shown in [Figure 13-12](#).

Finally, when you return the original microcontroller to your Arduino board, remember that the end with the notch should be on the right side, as shown in [Figure 13-13](#).

Connecting to an Existing Arduino Board

You can also use the USB interface of an Arduino Uno to upload sketches to the microcontroller in your breadboard Arduino. Using this method reduces wear on the Arduino board's socket and saves you money, because you won't need to buy a separate USB programming cable.

Here's how to upload a sketch to the microcontroller using the USB interface:

- Remove the microcontroller from your Arduino Uno and unplug the USB cable.
- Remove the power (if connected) from the breadboard Arduino circuit.
- Connect a wire from Arduino digital pin 0 to pin 2 of the breadboard's ATmega328P-PU; connect another wire from Arduino digital pin 1 to pin 3 of the ATmega328P-PU.

- . Connect the 5 V and GND from the Uno to the matching areas on the breadboard.
- . Connect a wire from Arduino RST to pin 1 of the ATmega328P-PU.
- . Plug the USB cable into the Arduino Uno board.

At this point, the computer should behave as if it were an ordinary Arduino Uno, so you should be able to upload sketches to the breadboard circuit's microcontroller normally and use the Serial Monitor if necessary.

Using an FTDI Programming Cable

The final method is the easiest, but it requires the purchase of a USB programming cable, known as an *FTDI cable* (simply because the USB interface circuitry inside is made by a company called FTDI). When purchasing an FTDI cable, make sure it's the 5 V model, because the 3.3 V model will not work properly. This cable (shown in [Figure 13-14](#)) has a USB plug on one end and a socket with six wires on the other. The USB end of this cable contains circuitry equivalent to the USB interface on an Arduino Uno board. The six-wire socket connects to the header pins shown in Figures 13-7 and 13-8.

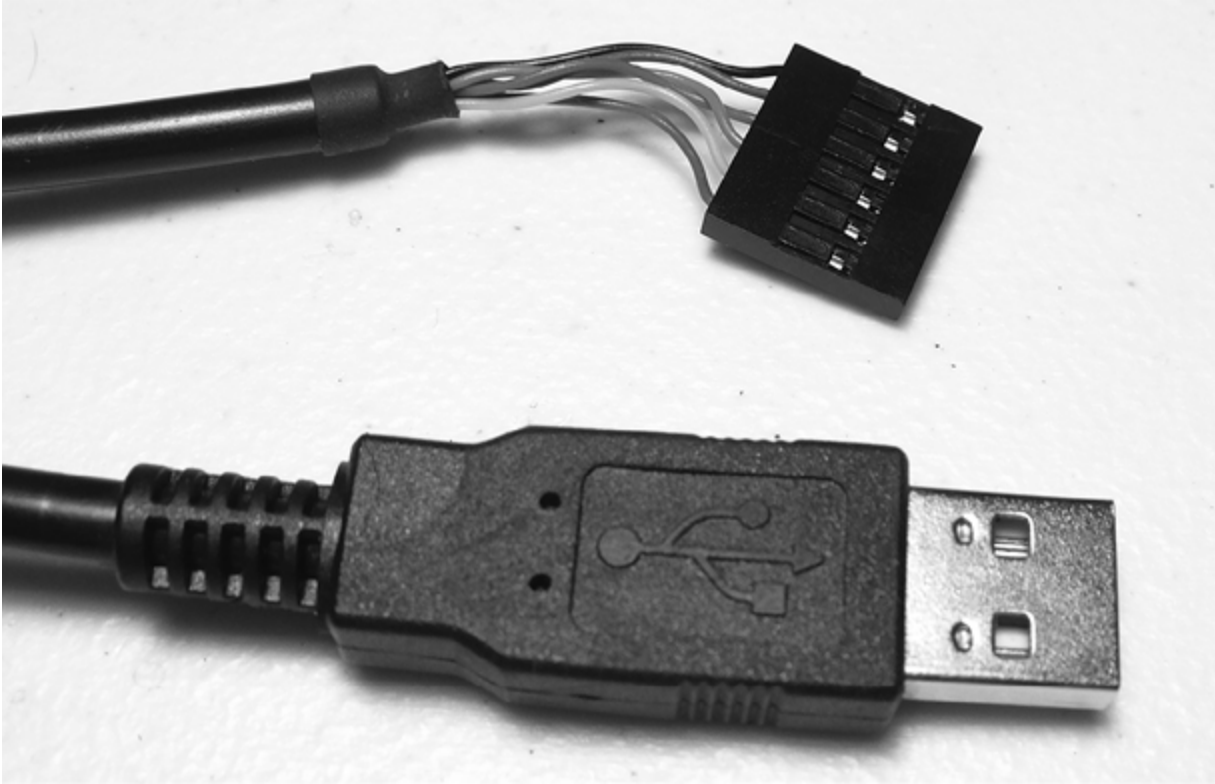


Figure 13-14: An FTDI cable

When you're connecting the cable, be sure that the side of the socket with the black wire connects to the GND pin on the breadboard's header pins. Once the cable is connected, it also supplies power to the circuit, just as a normal Arduino board would do.

Before uploading your sketch or using the serial monitor, change the board type to Arduino Duemilanove or Diecimila by choosing **Tools►Board** and then selecting the correct microcontroller ([Figure 13-15](#)).

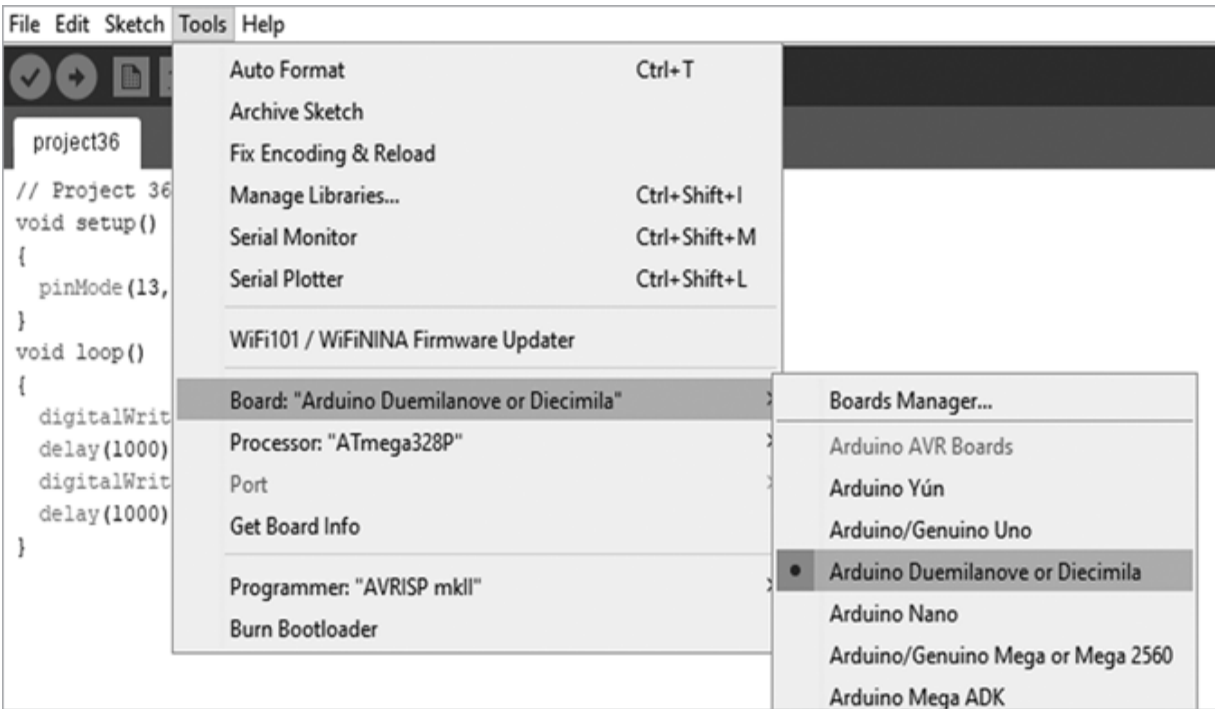


Figure 13-15: Changing the board type in the IDE


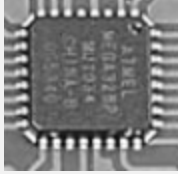

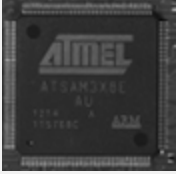
Once you have selected a method of uploading, test it by uploading the Project 36 sketch. Now you should be able to design more complex circuits using only a breadboard, which will let you create more projects for less money. You can even build more permanent projects from scratch if you learn to make your own printed circuit boards.

The Many Arduino and Alternative Boards

Although we have been working exclusively with the Arduino Uno board throughout the book, you can choose from many alternative boards. These will vary in physical size, the number of input and output pins, memory space for sketches, and price.

One of the crucial differences between boards is the microcontroller used. Current boards generally use the ATmega328 or the ATmega2560 microcontroller, and the Due uses another, more powerful version called the SAM3X8E. The main differences among these (including both versions of the ATmega328) are summarized in [Table 13-2](#).

Table 13-2: *Microcontroller Comparison Chart*

	ATmega328P-PU	ATmega328P SMD	ATmega2560	SAM3X8E
				
User-replaceable?	Yes	No	No	No
Processing speed	16 MHz	16 MHz	16 MHz	84 MHz
Operating voltage	5 V	5 V	5 V	3.3 V
Number of digital pins	14 (6 PWM capable)	14 (6 PWM capable)	54 (14 PWM capable)	54 (12 PWM capable)
Number of analog input pins	6	8	16	12
DC current per I/O pin	40 mA	40 mA	40 mA	3–15 mA
Available flash memory	31.5KB	31.5KB	248KB	512KB
EEPROM size	1KB	1KB	4KB	No EEPROM
SRAM size	2KB	2KB	8KB	96KB

The main parameters used to compare various Arduino-compatible boards are the types of memory they contain and the amount of each type.

Following are the three types of memory:

Flash memory is the space available to store a sketch after it has been compiled and uploaded by the IDE.

EEPROM (*electrically erasable programmable read-only memory*) is a small space that can store byte variables, as you'll learn in Chapter 19.

SRAM is the space available to store variables from your programs.

NOTE

Many Arduino boards are available in addition to the Uno, and the few described here are only the tip of the iceberg. When you're planning large or complex projects, don't be afraid to scale up to the larger Mega boards. By the same token, if you need only a few I/O pins for a more permanent project, consider the Nano or even a LilyPad.

Let's explore the range of available boards.

Arduino Uno

The Uno is currently considered the standard Arduino board. All Arduino shields ever made should be compatible with the Uno. The Uno is considered the easiest-to-use Arduino board due to its built-in USB interface and removable microcontroller.

Freeronics Eleven

Many boards on the market emulate the function of the Arduino Uno, and some have even improved on the standard design. One of these is the Freeronics Eleven, shown in [Figure 13-16](#).

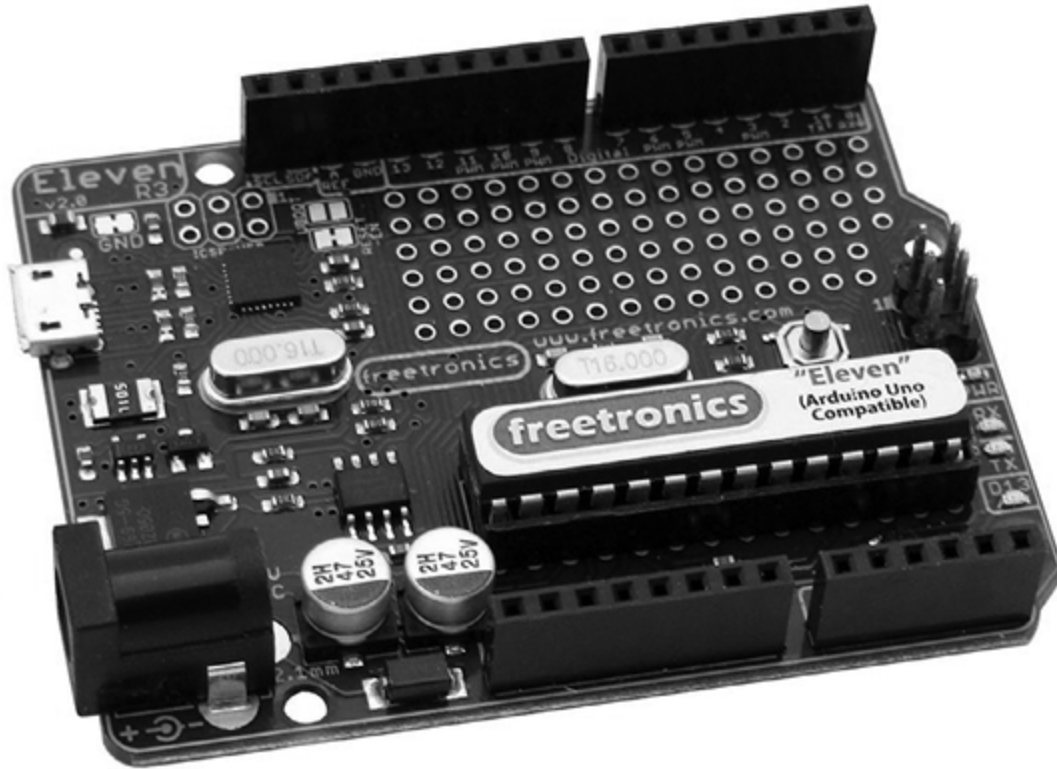


Figure 13-16: A Freetronics Eleven

Although the Eleven is completely compatible with the Arduino Uno, it offers several improvements that make it a worthwhile product. The first is the large prototyping area just below the digital I/O pins. This area allows you to construct your own circuit directly on the main board, which can save you space and money since you won't need to purchase a separate prototyping shield.

Second, the transmitter/receiver (TX/RX), power, and D13 LEDs are positioned on the far right of the board; this placement allows them to be visible even when a shield is attached. Finally, the Eleven uses a micro-USB socket, which is much smaller than the standard USB socket used on the Uno. This makes designing your own shield simpler, since you don't have to worry about your connections bumping into the USB socket. The Eleven is available from <http://www.freetronics.com.au/products/eleven/>.

The Adafruit Pro Trinket

The Adafruit Pro Trinket ([Figure 13-17](#)) is a miniaturized version of the Arduino Uno designed for working with solderless breadboards, wearable

electronics, or any situation in which you need a much smaller board.

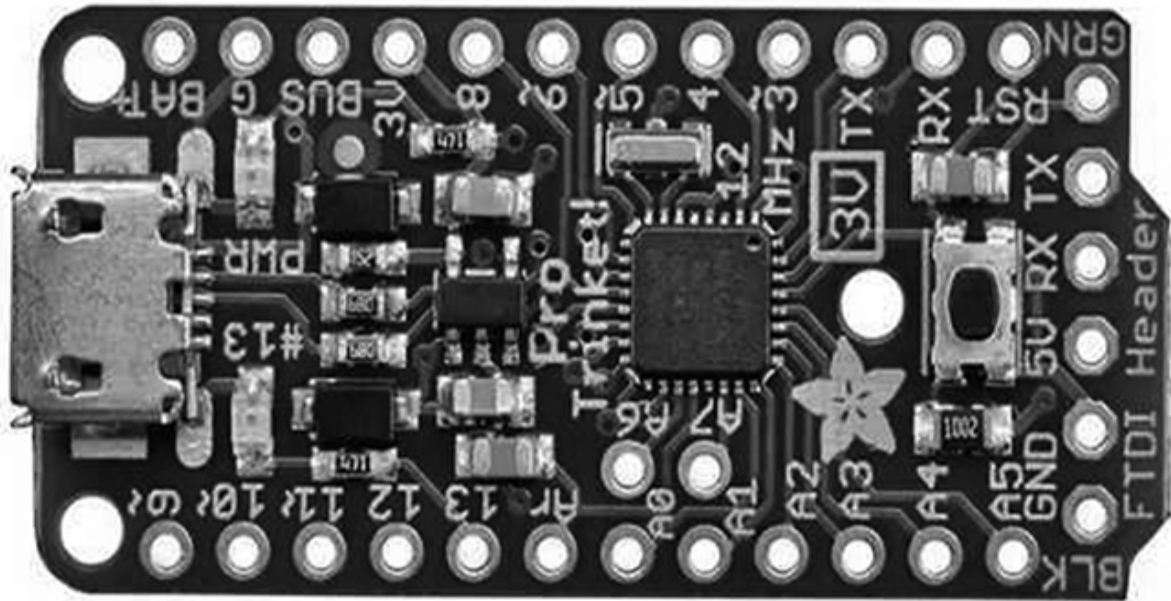


Figure 13-17: An Adafruit Pro Trinket

There are some slight differences from the Arduino Uno (for example, no serial output unless you use an external FTDI cable); however, for the price this board is a great value. The Pro Trinket is available from <http://www.adafruit.com/trinket/>.

The Arduino Nano

When you need a compact, assembled Arduino-compatible board, the Nano should fit the bill. Also designed to work in a solderless breadboard, the Nano ([Figure 13-18](#)) is a tiny but powerful Arduino.

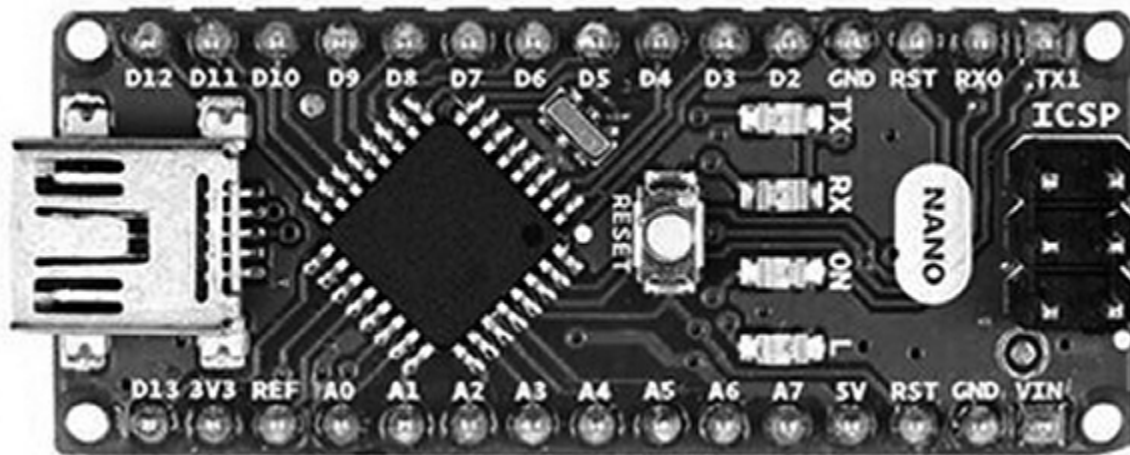


Figure 13-18: An Arduino Nano

The Nano measures only 0.7 inches by 1.77 inches (17.8 mm by 44.9 mm), yet it offers all the functionality of the classic Arduino Duemilanove. Furthermore, it uses the SMD version of the ATmega328P, so it has two extra analog input pins (A6 and A7). The Nano is available from <https://store.arduino.cc/usa/arduino-nano/>.

The LilyPad

The LilyPad is designed to be integrated into creative projects, such as wearable electronics. In fact, you can actually wash a LilyPad with water and a mild detergent, so it's ideal to use for lighting up a sweatshirt, for example. The board design is unique, as shown in [Figure 13-19](#).

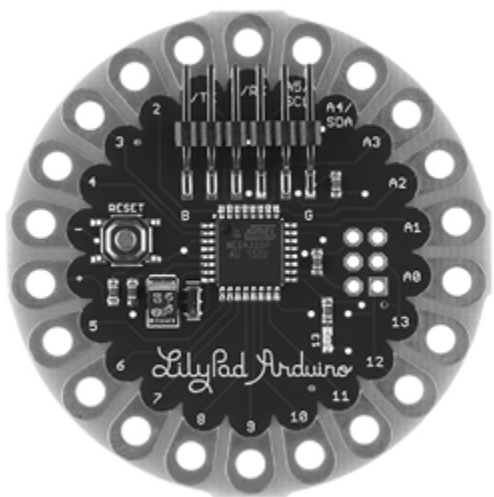


Figure 13-19: An Arduino LilyPad

The I/O pins on the LilyPad require that wires be soldered to the board, so the LilyPad is more suited for use with permanent projects. As part of its minimalist design, it has no voltage regulation circuitry, so it's up to the user to provide their own supply between 2.7 and 5.5 V. The LilyPad also lacks a USB interface, so a 5 V FTDI cable is required to upload sketches. You can get Arduino LilyPad or compatible boards from almost any Arduino retailer.

The Arduino Mega 2560

When you run out of I/O pins on your Arduino Uno or you need space for much larger sketches, consider a Mega 2560, shown in [Figure 13-20](#). It is physically a much larger board than the Uno, measuring 4.3 inches by 2.1 inches (109.2 mm by 53.4 mm).

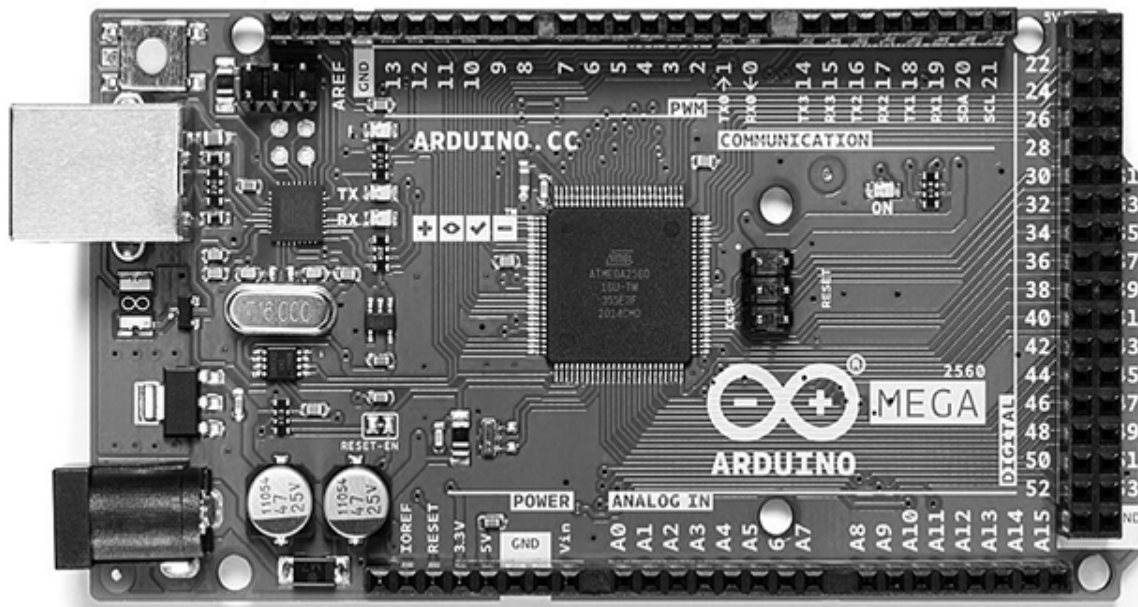


Figure 13-20: An Arduino Mega 2560

Although the Mega 2560 board is much larger than the Uno, you can still use most Arduino shields with it, and Mega-sized prototyping shields are available for larger projects that the Uno can't accommodate. Since the Mega uses the ATmega2560 microcontroller, its memory space and I/O capabilities (as described in [Table 13-2](#)) are much greater than those of the Uno. Additionally, four separate serial communication lines increase its data transmission capabilities. You can get Mega 2560 boards from almost any Arduino retailer.

The Freetronics EtherMega

When you need an Arduino Mega 2560, a microSD card shield, and an Ethernet shield to connect to the internet, your best option is an EtherMega ([Figure 13-21](#)), because it has all these functions on a single board and is less expensive than purchasing each component separately. The EtherMega is available from <http://www.freetronics.com/em/>.

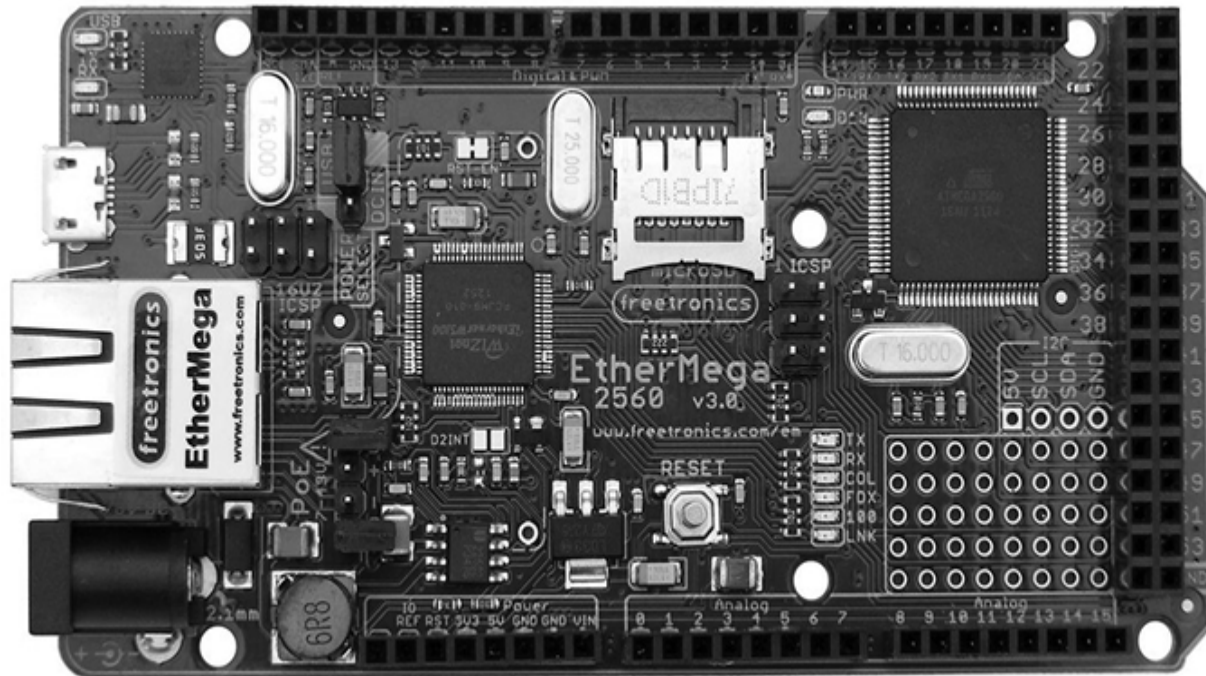


Figure 13-21: A Freetronics EtherMega

The Arduino Due

With an 84 MHz processor that can run your sketches much faster, this is the most powerful Arduino board ever released. As you can see in [Figure 13-22](#), the board is quite like the Arduino Mega 2560, but there is an extra USB port for external devices and different pin labels.

Furthermore, the Due has just over 16 times the memory of an Uno board, so you can create really complex and detailed sketches. However, the Due operates only on 3.3 V—so any circuits, shields, or other devices connected to the analog or digital pins cannot have a voltage greater than 3.3 V. While you need to be aware of these limitations, generally the benefits of using the Due outweigh the changes in the hardware.

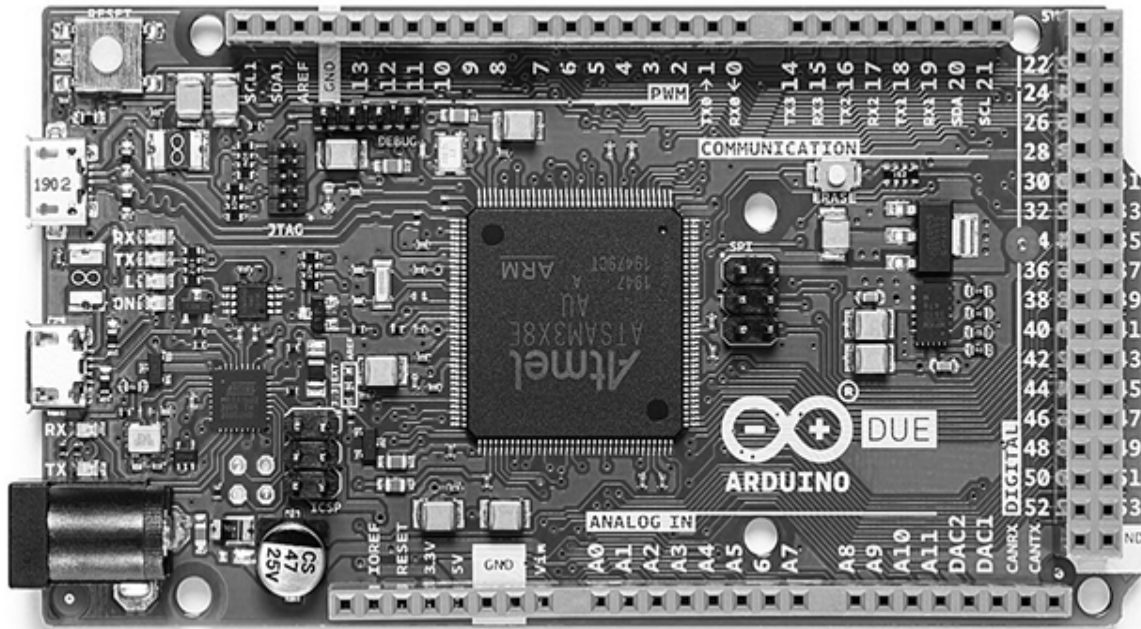


Figure 13-22: An Arduino Due

NOTE

When shopping for your next Arduino board or accessory, be sure to buy from a reputable retailer that offers support and a guarantee. The internet is flooded with cheap alternatives, but manufacturers often cut corners to make these abnormally low-priced products, and you might have no way of seeking recompense if you're sold a faulty or incorrectly specified product.

OPEN SOURCE HARDWARE

The Arduino hardware design is released to the public so that anyone can manufacture, modify, distribute, and use it as they see fit. This type of distribution falls under the umbrella of *open source hardware*—a recent movement that is in opposition to the concept of copyrights and other legal protection of intellectual property. The Arduino team decided to allow its designs to be free for the benefit of the larger hardware community and for the greater good.

In the spirit of open source hardware, many organizations that produce accessories for or modifications of the original Arduino boards publish their designs under the same license. This allows for a much faster process of product improvement than would be possible for a single organization developing the product alone.

Looking Ahead

This chapter has given you a broader picture of the types of hardware available and introduces the idea of a breadboard Arduino that you build yourself. You've seen the parts that make up the Arduino design, and you've seen how to build your own Arduino using a solderless breadboard. You now know how to make more than one Arduino-based prototype without having to purchase more boards. You also know about the variety of Arduino boards on the market, and you should be able to select the board that best meets your needs. Finally, you've gained an understanding of the open source movement and Arduino's participation in it.

In the next chapter, you'll learn to use a variety of motors and begin working on your own Arduino-controlled motorized robot!

14

MOTORS AND MOVEMENT

In this chapter you will

Use a servo to create an analog thermometer

Learn how to control the speed and direction of DC electric motors

Learn how to control small stepper motors

Use an Arduino motor shield

Begin work on a motorized robot vehicle

Use simple microswitches for collision avoidance

Use infrared and ultrasonic distance sensors for collision avoidance

Making Small Motions with Servos

A *servo* (short for *servomechanism*) is an electric motor with a built-in sensor. It can be commanded to rotate to a specific angular position. By attaching the shaft of the servo to other machines, like wheels, gears, and levers, you can precisely control items in the external world. For example, you might use a servo to control the steering of a remote control car by connecting the servo to a *horn*, a small arm or bar that the servo rotates. An example of a horn is one of the hands on an analog clock. [*Figure 14-1*](#) shows a servo and three types of horns.



Figure 14-1: A servo and various horns

Selecting a Servo

When you're selecting a servo, consider several parameters:

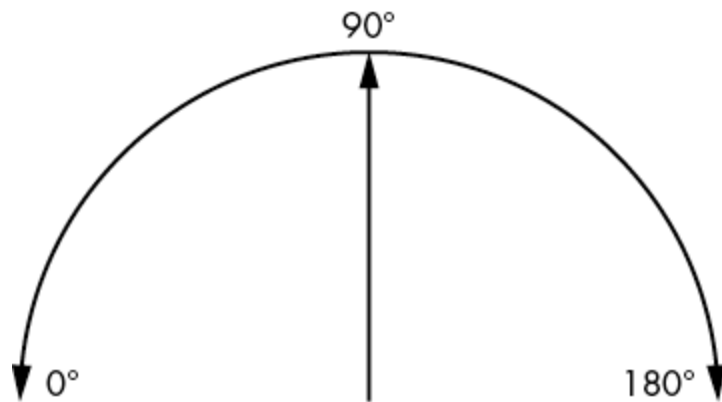
Speed The time it takes for the servo to rotate, usually measured in seconds per angular degree, revolutions per minute (RPM), or seconds per 60 degrees.

Rotational range The angular range through which the servo can rotate—for example, 180 degrees (half of a full rotation) or 360 degrees (one complete rotation).

Current How much current the servo draws. When using a servo with an Arduino, you may need to use an external power supply for the servo.

Torque The amount of force the servo can exert when rotating. The greater the torque, the heavier the item the servo can control. The torque produced is generally proportional to the amount of current used.

The servo shown in [Figure 14-1](#) is a generic SG90-type servo. It is inexpensive and can rotate up to 180 degrees, as shown in [Figure 14-2](#).



[Figure 14-2](#): Example servo rotation range

Connecting a Servo

It's easy to connect a servo to an Arduino because it needs only three wires. If you're using the SG90, the darkest wire connects to GND, the center wire connects to 5 V, and the lightest wire (the *pulse* or *data* wire) connects to a digital pin. If you're using a different servo, check its data sheet for the correct wiring.

Putting a Servo to Work

Now let's put our servo to work. In this sketch, the servo will turn through its rotational range. Connect the servo to your Arduino as described, with the pulse wire connected to digital pin 4, and then enter and upload the sketch in [Listing 14-1](#).

```
// Listing 14-1

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

```
void setup()
{
  myservo.attach(4);
}

void loop()
{
  myservo.write(180);
  delay(1000);
  myservo.write(90);
  delay(1000);
  myservo.write(0);
  delay(1000);
}
```

Listing 14-1: Servo demonstration sketch

In this sketch, we use the Servo library, which needs to be installed. Follow the instructions outlined in Chapter 7. In the Library Manager, find and then install the “Servo by Michael Margolis, Arduino” library. Create an instance of the servo with the following:

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

Then, in `void setup()`, we tell the Arduino which digital pin the servo control is using:

```
myservo.attach(4); // control pin on digital 4
```

Now we simply move the servo with the following:

```
myservo.write(x);
```

Here, x is an integer between 0 and 180 representing the angular position to which the servo will be moved. When running the sketch in [Listing 14-1](#), the servo will rotate across its maximum range, stopping at the extremes (0 degrees and 180 degrees) and at the midpoint (90 degrees). When looking at your servo, note that the 180-degree position is on the left and 0 degrees is on the right.

In addition to pushing or pulling objects, servos can be used to communicate data in a similar way as an analog gauge. For example, you could use a servo as an analog thermometer, as you'll see in Project 37.

Project #37: Building an Analog Thermometer

Using our servo and the TMP36 temperature sensor from earlier chapters, we'll build an analog thermometer. We'll measure the temperature and then convert this measurement to an angle between 0 and 180 degrees to indicate a temperature between 0 and 30 degrees Celsius. The servo will rotate to the angle that matches the current temperature.

The Hardware

The required hardware is minimal:

One TMP36 temperature sensor

One breadboard

One small servo

Various connecting wires

Arduino and USB cable

The Schematic

The circuit is also very simple, as shown in [Figure 14-3](#).

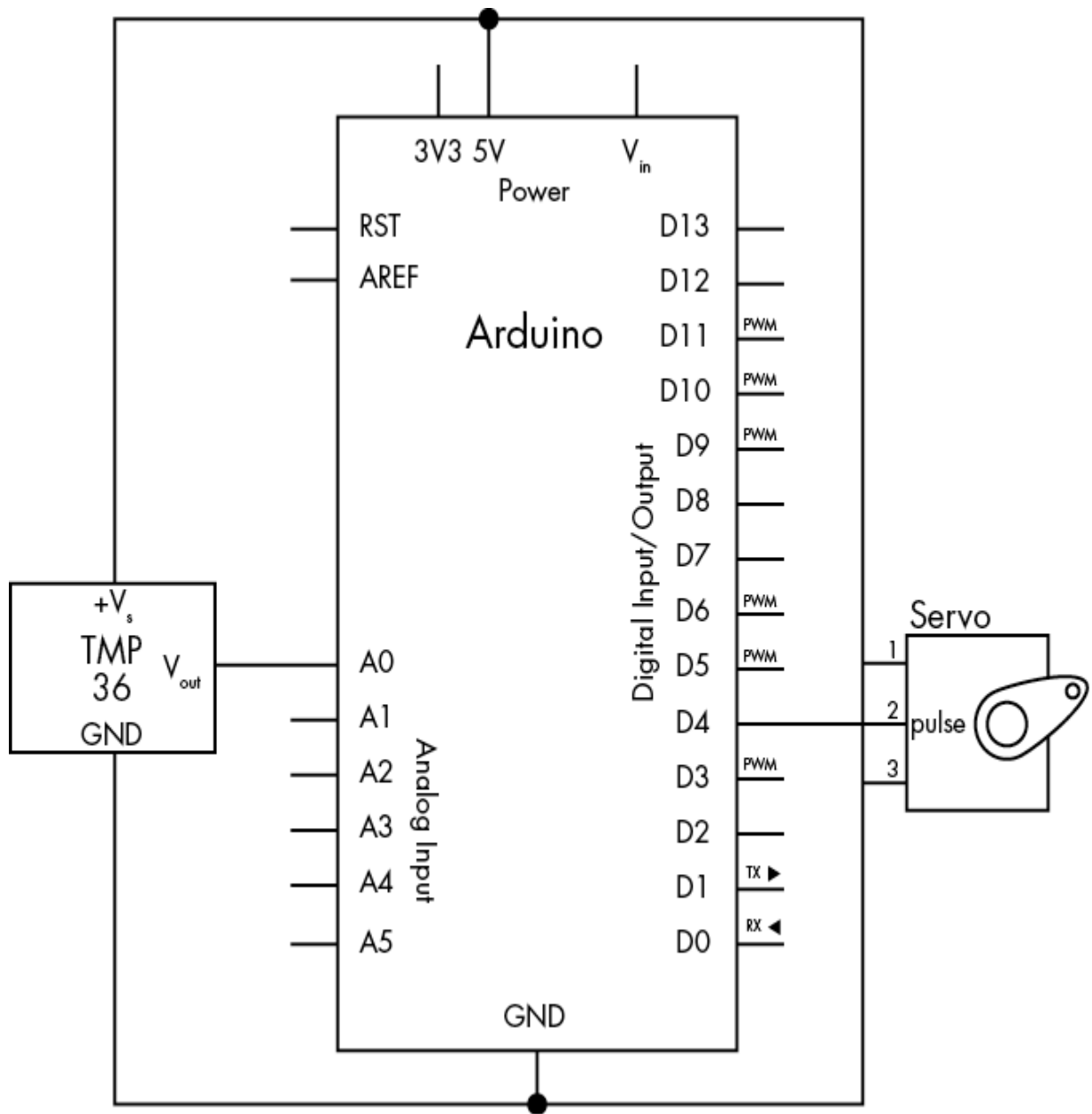


Figure 14-3: Schematic for Project 37

The Sketch

The sketch will determine the temperature using the same method used in Project 8 in Chapter 4. Then it will convert the temperature into an angular rotation value for the servo.

Enter and upload the following sketch:

```
// Project 37 - Building an Analog Thermometer

float voltage = 0;
float sensor = 0;
float currentC = 0;
int    angle = 0;

#include <Servo.h>
Servo myservo;

void setup()
{
    myservo.attach(4);
}
1 int calculateservo(float temperature)
{
    float resulta;
    int resultb;
    resulta = -6 * temperature;
    resulta = resulta + 180;
    resultb = int(resulta);
    return resultb;
}

void loop()
{
    // read current temperature
    sensor = analogRead(0);
    voltage = (sensor*5000)/1024;
    voltage = voltage-500;
    currentC = voltage/10;

    // display current temperature on servo
1  angle = calculateservo(currentC);
    // convert temperature to a servo position
    if (angle>=0 && angle <=30)
    {
        myservo.write(angle); // set servo to temperature
        delay(1000);
    }
}
```

Most of this sketch should be clear to you at this point, but the function `calculateservo()` at 1 is new. This function converts the temperature into the matching angle for the servo to use according to the following formula:

$$\text{angle} = (-6 \times \text{temperature}) + 180$$

You might find it useful to make a *backing sheet* to show the range of temperatures that the servo will display, with a small arrow to create a realistic effect. An example is shown in [Figure 14-4](#). You can download a printable version of the backing sheet from the book's website: <https://nostarch.com/arduino-workshop-2nd-edition/>.

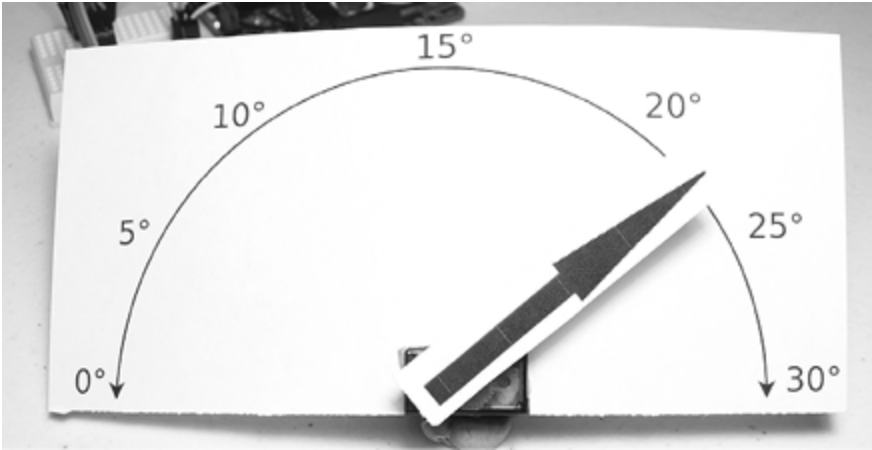


Figure 14-4: A backing sheet indicates the temperature on our thermometer.

Using Electric Motors

The next step in our motor-controlling journey is to work with small electric motors. Small motors are used for many applications, from small fans to toy cars to model railroads.

Selecting a Motor

As with servos, you need to consider several parameters when you're choosing an electric motor:

The operating voltage The voltage at which the motor is designed to operate. This can vary, from 3 V to more than 12 V.

The current without a load The amount of current the motor uses at its operating voltage while spinning freely, without anything connected to the motor's shaft.

The stall current The amount of current used by the motor when it is trying to turn but cannot because of the load on the motor.

The speed at the operating voltage The motor's speed in RPM.

Our example will use a small, inexpensive electric motor with a speed of 8,540 RPM when running on 3 V, similar to the one shown in [Figure 14-5](#).

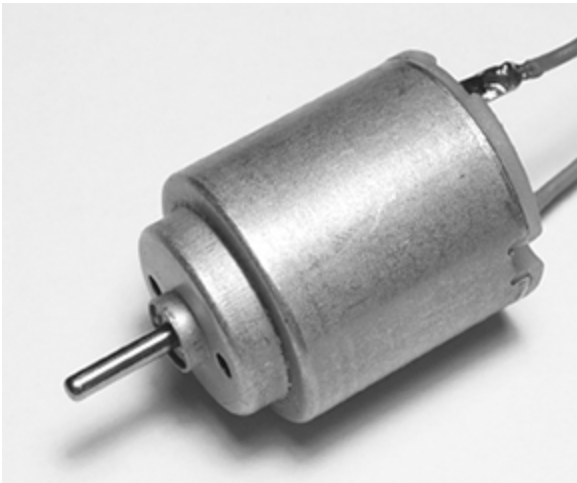


Figure 14-5: Our small electric motor

To control our motor we'll use a transistor, introduced in Chapter 3. Because our motor uses up to 0.7 A of current (more than can be passed by the BC548 transistor), we'll use a transistor called a Darlington for this project.

The TIP120 Darlington Transistor

A *Darlington transistor* is nothing more than two transistors connected together. It can handle high currents and voltages. The TIP120 Darlington can pass up to 5 A of current at 60 V, which is more than enough to control our small motor. The TIP120 uses a similar schematic symbol as the BC548, as shown in [Figure 14-6](#), but the TIP120 transistor is physically larger than the BC548.

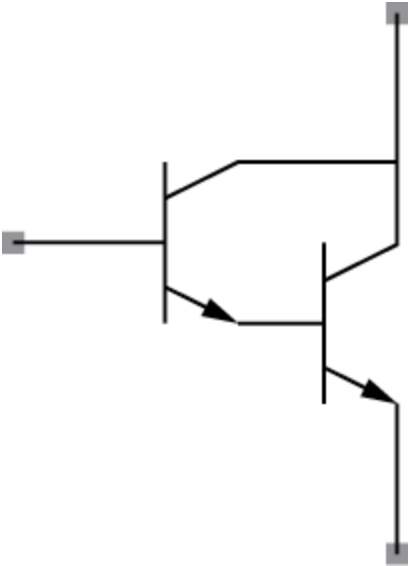


Figure 14-6: TIP120 schematic symbol

The TIP120 uses the TO-220 housing style, as shown in [Figure 14-7](#).

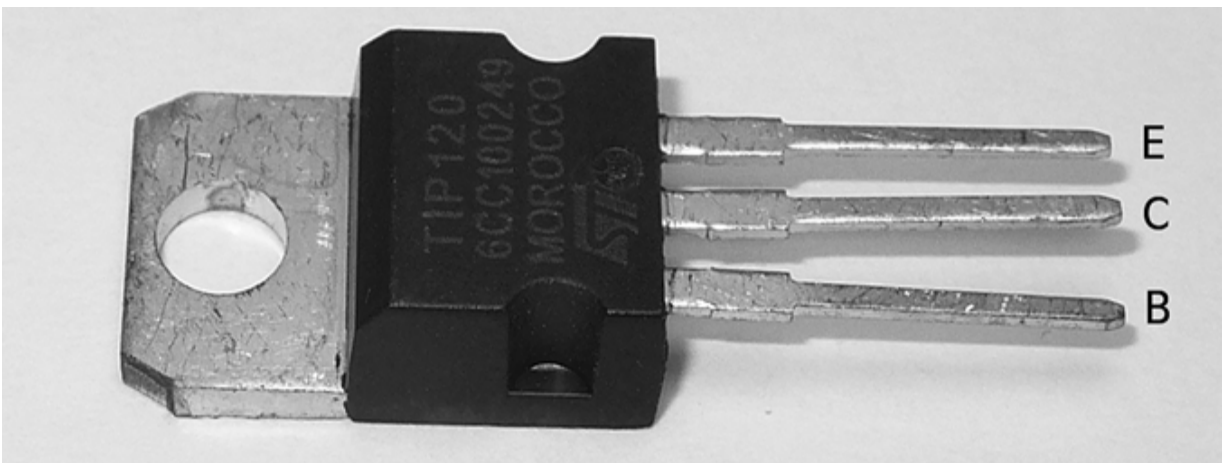


Figure 14-7: The TIP120

When you're looking at the TIP120 from the labeled side, the pins from left to right are base (B), collector (C), and emitter (E). The metal heat sink tab is also connected to the collector.

Project #38: Controlling the Motor

In this project, we'll control the motor by adjusting the speed.

The Hardware

The following hardware is required:

One small 3 V electric motor

One 1 k Ω resistor (R1)

One breadboard

One 1N4004 diode

One TIP120 Darlington transistor

A separate 3 V power source

Various connecting wires

Arduino and USB cable

You must use a separate power source for motors, because the Arduino cannot supply enough current for the motor in all possible situations. If the motor becomes stuck, then it will draw up to its *stall current*, which could be more than 1 A. That's more than the Arduino can supply, and if it attempts to supply that much current the Arduino could be permanently damaged.

A separate battery holder is a simple solution. For a 3 V supply, a two-cell AA battery holder with flying leads will suffice, such as the one shown in [*Figure 14-8*](#).

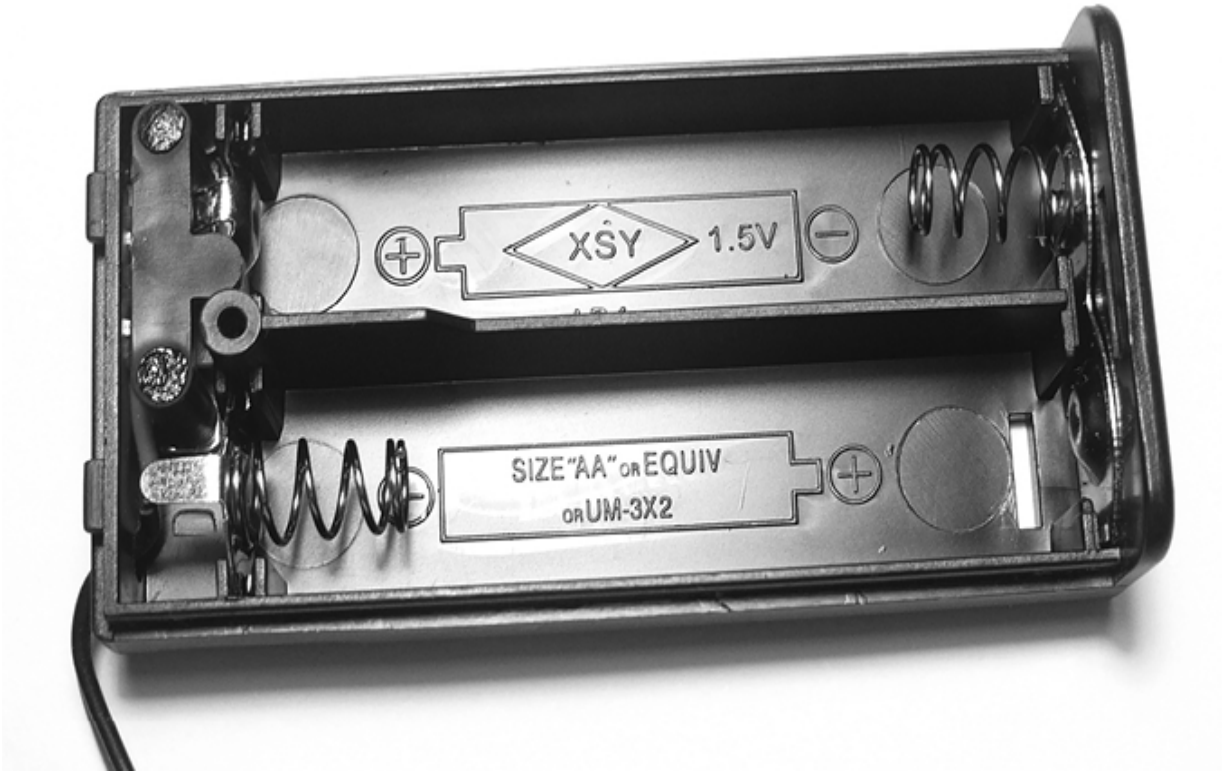


Figure 14-8: A two-cell AA battery holder

The Schematic

Assemble the circuit as shown in the schematic in [Figure 14-9](#).

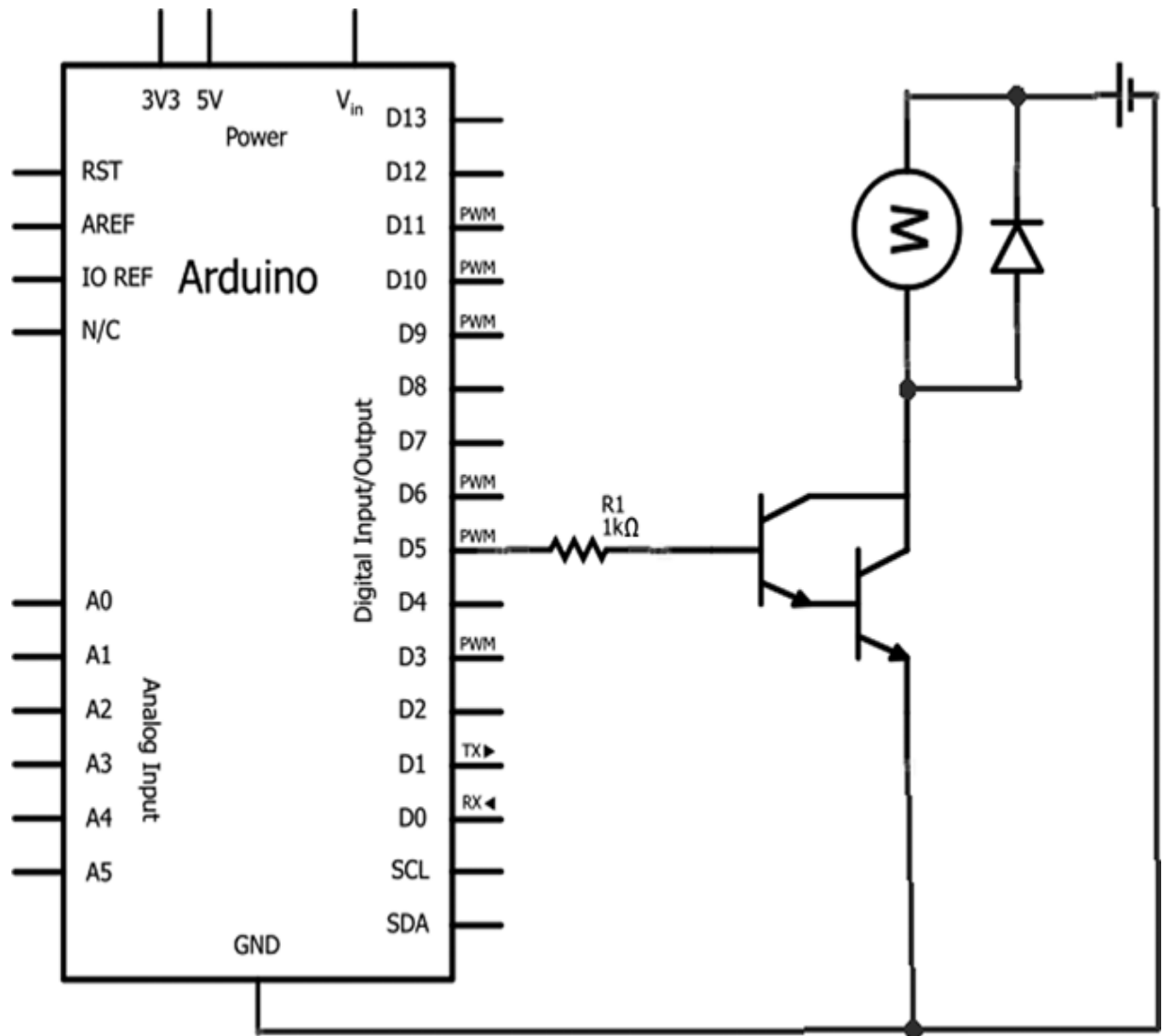


Figure 14-9: Schematic for Project 38

The Sketch

In this project, we'll adjust the speed of the motor from stopped (zero) to the maximum and then reduce it back to zero. Enter and upload the following sketch:

```
// Project 38 - Controlling the Motor

void setup()
{
  pinMode(5, OUTPUT);
}
```

```
void loop()
{
1   for (int a=0; a<256; a++)
    {
        analogWrite(5, a);
2       delay(100);
    }
3   delay(5000);
4   for (int a=255; a>=0; a--)
    {
        analogWrite(5,a);
        delay(100);
    }
    delay(5000);
}
```

We control the speed of the motor using pulse-width modulation (as demonstrated in Project 3 in Chapter 3). Recall that we can do this only with digital pins 3, 5, 6, 9, 10, and 11. Using this method, current is applied to the motor in short bursts: the longer the burst, the faster the speed, as the motor is on more than it is off during a set period of time. So at 1, the motor speed starts at zero and increases slowly; you can control the acceleration by changing the delay value at 2. At 3, the motor is running as fast as possible and holds that speed for 5 seconds. Then, from 4, the process reverses, and the motor slows to a stop.

NOTE

When it starts moving, you may hear a whine from the motor, which sounds like the sound of an electric train or a tram when it moves away from a station. This is normal and nothing to worry about.

The diode is used in the same way it was with the relay control circuit described in Figure 3-19 on page 42 to protect the circuit. When the current is switched off from the motor, stray current exists for a brief time inside the motor's coil and has to go somewhere. The diode allows the stray current to loop around through the coil until it dissipates as a tiny amount of heat.

Using Small Stepper Motors

Stepper motors are different from regular DC motors, in that they divide a full rotation of the motor into a fixed number of steps. They do this by using two coil windings that are independently controlled. So instead of controlling a rotation with varying voltage as with a regular DC motor, you instead turn on or off the coils in a stepper motor in a certain pattern to rotate the shaft in either direction a set number of times. This control makes steppers ideal for jobs that need precise motor positioning. They are quite commonly found in devices from computer printers to advanced manufacturing devices.

We will demonstrate stepper motor operation using the model 28BYJ-48, as shown in [Figure 14-10](#). This type of stepper motor can be controlled to rotate to one of 4,096 positions; that is, one full rotation is divided into 4,096 steps.

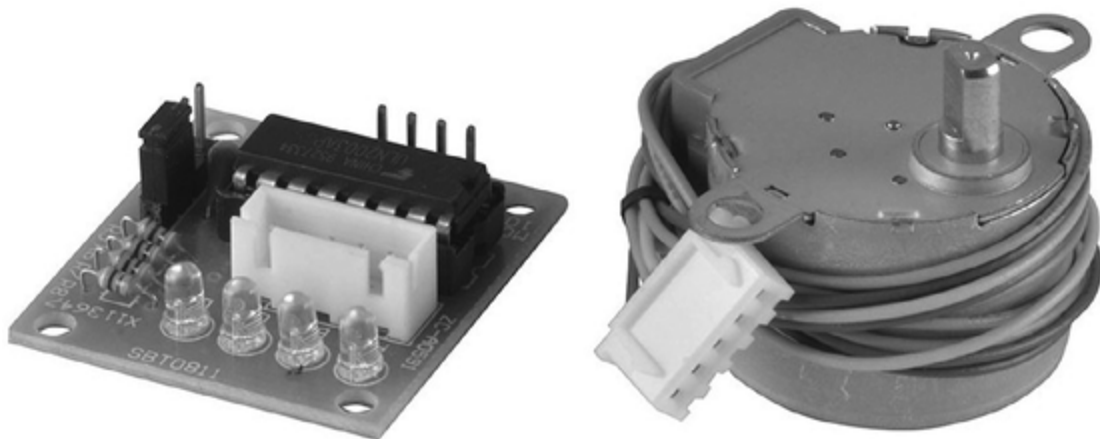


Figure 14-10: A stepper motor and controller board

The board next to the motor is used as an interface between your Arduino and the stepper motor, making connection easy and fast. It is usually supplied along with the stepper motor. A close-up is shown in [Figure 14-11](#).

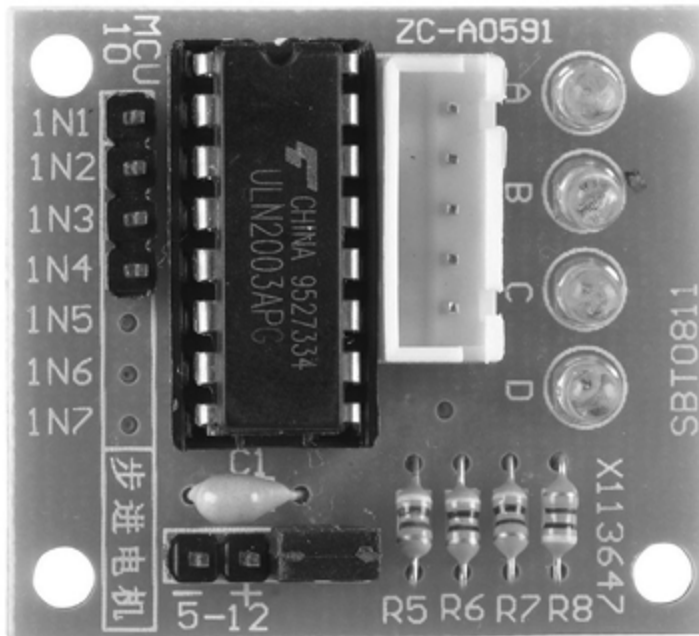


Figure 14-11: The stepper motor controller board

Now to connect your stepper motor to the Arduino. Make the connections as shown in [Table 14-1](#).

Table 14-1: *Connections Between the Stepper Motor Controller Board and the Arduino*

Control board pin label	Arduino pin
IN1	D8
IN2	D9
IN3	D10
IN4	D11
5-12 V+	5 V
5-12 V-	GND

You can briefly run the stepper motor using power from your Arduino if nothing else is drawing power from the Arduino. However, this is not recommended. Instead, use an external 5 V power supply such as a plug pack or other convenient source. As the controller board doesn't have a DC socket, you can use an external socket with terminal blocks to make easy, solderless connections, as shown in [Figure 14-12](#).



Figure 14-12: DC socket terminal block

You can then connect jumper wires from the + and – connectors on the terminal blocks to those on the stepper motor controller board. To simplify controlling the stepper motor in our Arduino sketches, you can use a neat Arduino library called CheapStepper. You can download it from <https://github.com/tyhenry/CheapStepper/archive/master.zip> and install it using the method described in Chapter 7.

Once you have successfully installed the library and connected your stepper motor as described earlier, enter and upload [Listing 14-2](#).

```
// Listing 14-2
1 #include <CheapStepper.h>
2 CheapStepper stepper (8, 9, 10, 11);
3 boolean clockwise = true;
  boolean cclockwise = false;

4 void setup()
  {
```

```

    stepper.setRpm(20);
    Serial.begin(9600);
}

void loop()
{
    Serial.println("stepper.moveTo (Clockwise, 0)");
5    stepper.moveTo (clockwise, 0);
    delay(1000);

    Serial.println("stepper.moveTo (Clockwise, 1024)");
5    stepper.moveTo (clockwise, 1024);
    delay(1000);

    Serial.println("stepper.moveTo (Clockwise, 2048)");
    stepper.moveTo (clockwise, 2048);
    delay(1000);

    Serial.println("stepper.moveTo (Clockwise, 3072)");
    stepper.moveTo (clockwise, 3072);
    delay(1000);

    Serial.println("stepper.moveTo (CClockwise, 512)");
5    stepper.moveTo (cclockwise, 512);
    delay(1000);

    Serial.println("stepper.moveTo (CClockwise, 1536)");
5    stepper.moveTo (cclockwise, 1536);
    delay(1000);

    Serial.println("stepper.moveTo (CClockwise, 2560)");
    stepper.moveTo (cclockwise, 2560);
    delay(1000);

    Serial.println("stepper.moveTo (CClockwise, 3584)");
    stepper.moveTo (cclockwise, 3584);
    delay(1000);
}

```

Listing 14-2: Testing the stepper motor

Operation of the stepper motor is quite simple. We first include the library at 1 and create an instance of the motor at 2. (If you wish to change the digital pins used for the controller board, update them here.) The control

function uses `true` and `false` for clockwise and counterclockwise rotation, respectively, so we assign these to Boolean variables at 3 to make things clearer. Finally, the motor can be instructed to rotate to one of the 4,096 positions using the function:

```
Stepper.moveTo(direction, location);
```

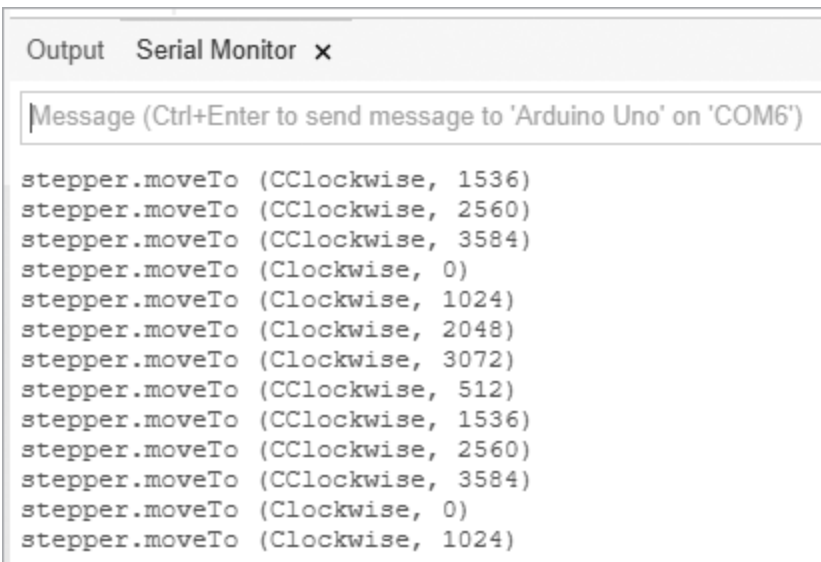
where the `direction` is either `clockwise` or `cclockwise` and the `location` is a value between 0 and 4,095. This is done starting at 5 and repeatedly through the end of the sketch.

Furthermore, in `void setup()` at 4, we set the motor's rotational speed to 20 RPM using:

```
stepper.setRpm(20);
```

This is the recommended speed for our stepper motor. Other motors will vary, so you should check with the supplier for these details.

A few moments after you upload the sketch, your stepper motor will start rotating to various positions, and you can see the commands echoed in the Serial Monitor, as shown in [Figure 14-13](#).

A screenshot of the Arduino IDE Serial Monitor window. The title bar says "Output Serial Monitor x". Below the title bar is a text input field with the placeholder "Message (Ctrl+Enter to send message to 'Arduino Uno' on 'COM6')". The main area of the window displays a list of 15 commands sent to the stepper motor, each on a new line. The commands are: `stepper.moveTo (CClockwise, 1536)`, `stepper.moveTo (CClockwise, 2560)`, `stepper.moveTo (CClockwise, 3584)`, `stepper.moveTo (Clockwise, 0)`, `stepper.moveTo (Clockwise, 1024)`, `stepper.moveTo (Clockwise, 2048)`, `stepper.moveTo (Clockwise, 3072)`, `stepper.moveTo (CClockwise, 512)`, `stepper.moveTo (CClockwise, 1536)`, `stepper.moveTo (CClockwise, 2560)`, `stepper.moveTo (CClockwise, 3584)`, `stepper.moveTo (Clockwise, 0)`, and `stepper.moveTo (Clockwise, 1024)`.

```
stepper.moveTo (CClockwise, 1536)
stepper.moveTo (CClockwise, 2560)
stepper.moveTo (CClockwise, 3584)
stepper.moveTo (Clockwise, 0)
stepper.moveTo (Clockwise, 1024)
stepper.moveTo (Clockwise, 2048)
stepper.moveTo (Clockwise, 3072)
stepper.moveTo (CClockwise, 512)
stepper.moveTo (CClockwise, 1536)
stepper.moveTo (CClockwise, 2560)
stepper.moveTo (CClockwise, 3584)
stepper.moveTo (Clockwise, 0)
stepper.moveTo (Clockwise, 1024)
```

[Figure 14-13](#): Commands sent to the stepper motor

Project #39: Building and Controlling a Robot Vehicle

Although controlling the speed of one DC motor can be useful, let's move into more interesting territory by controlling four DC motors at once and affecting their speed *and* direction. Our goal is to construct a four-wheeled vehicle-style robot that we'll continue to work on in the next few chapters. Here I'll describe the construction and basic control of our robot.

Our robot has four motors that each control one wheel, allowing it to travel at various speeds as well as rotate in place. You will be able to control the speed and direction of travel, and you will also learn how to add parts to enable collision avoidance and remote control. Once you have completed the projects in this book, you will have a solid foundation for creating your own versions of this robot and bringing your ideas to life.

The Hardware

You'll need the following hardware :

Robot vehicle chassis with four DC motors and wheels

Four-cell AA battery holder with wired output

Four alkaline AA cells

L293D Motor Drive Shield for Arduino

Arduino and USB cable

The Chassis

The foundation of any robot is a solid chassis containing the motors, drivetrain, and power supply. An Arduino-powered robot also needs to have room to mount the Arduino and various external parts.

You can choose from many chassis models available on the market. To keep things simple, we're using an inexpensive robot chassis that includes four small DC motors that operate at around 6 V DC and matching wheels, as shown in [*Figure 14-14*](#).



Figure 14-14: Our robot chassis

The task of physically assembling the robot chassis will vary between models, and you may need a few basic tools such as screwdrivers and pliers. If you're not sure about your final design but wish to get your robot moving, a favored technique is to hold the electronics to the chassis with sticky products such as Blu Tack.

The Power Supply

The motors included with the robot chassis typically operate at around 6 V DC, so we'll use a four-cell AA battery holder to power our robot, as shown in [Figure 14-15](#).



Figure 14-15: A battery holder for four AA cells

Some AA cell battery holders will not have the wiring needed to connect to our project and instead will have connections for a 9 V battery snap (as our unit in [Figure 14-15](#) does). In this case, you'll need a battery snap like the one in [Figure 14-16](#).

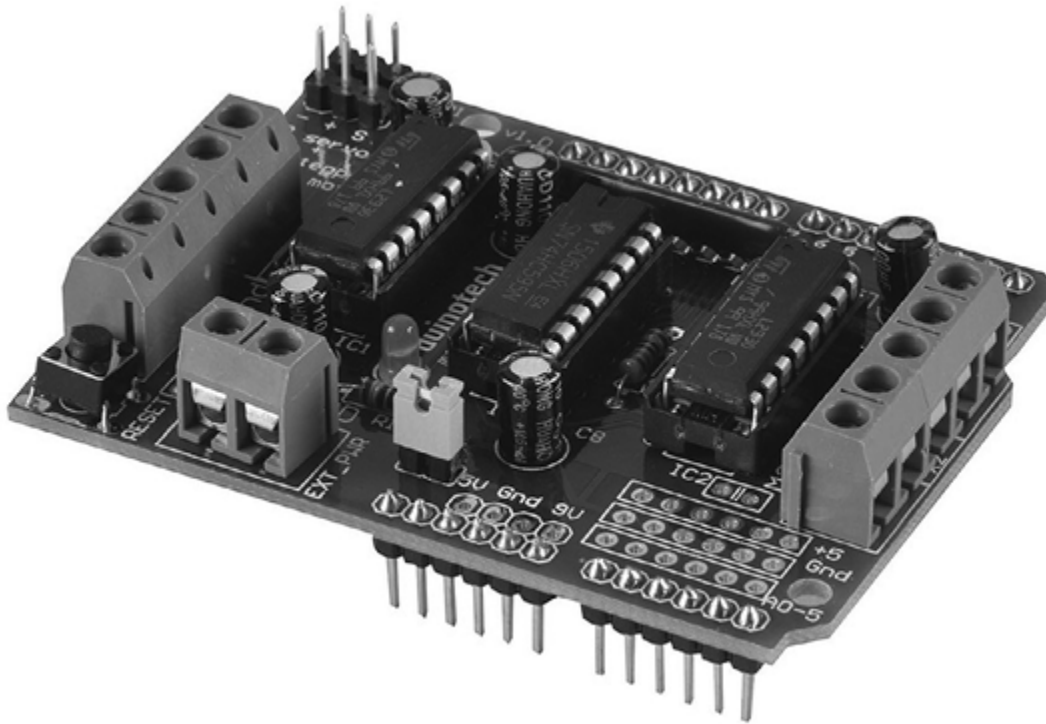


Figure 14-16: A battery cable used to connect the battery holder to the Arduino

The Schematic

The final requirement is to create the circuitry to control the four motors in the chassis. Although we could use the circuitry shown in [Figure 14-9](#) for each of the motors, this wouldn't allow us to control the direction of the

motors and could be somewhat inconvenient to wire up ourselves. Instead, we'll use a *motor shield*. A motor shield contains the circuitry we need to handle the higher current drawn by the motors and accepts commands from the Arduino to control both the speed and direction of the motors. For our robot, we'll use an L293D Motor Drive Shield for Arduino, as shown in [Figure 14-17](#).



[Figure 14-17](#): An L293D Motor Drive Shield for Arduino

Connecting the Motor Shield

Making the required connections to the motor shield is simple: connect the wires from the battery holder to the terminal block at the bottom left of the shield, as shown in [Figure 14-18](#). The black wire (negative) must be on the right side and the red wire on the left.

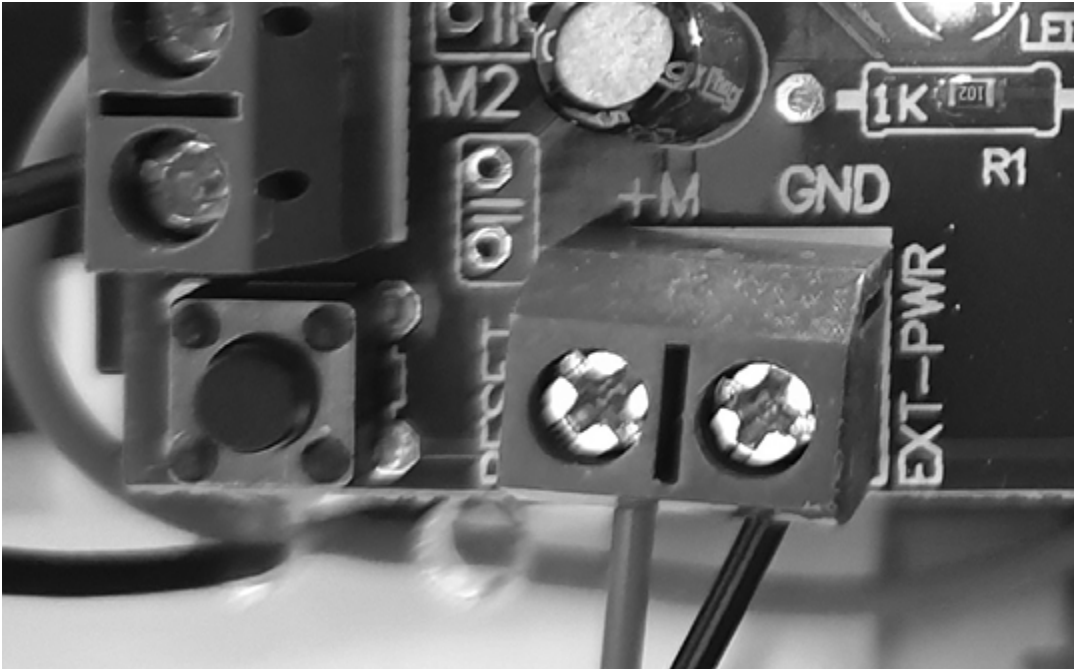


Figure 14-18: DC power connection

Next you need to connect each DC motor to the motor shield. We'll refer to the two DC motors at the front of the chassis as motor 2 (left) and motor 1 (right) and the two DC motors at the rear as motor 3 (left) and motor 4 (right). Each motor will have a red and a black wire, so connect them to the matching terminal blocks on the left-hand and right-hand side of the motor shield, as shown in [Figure 14-19](#).

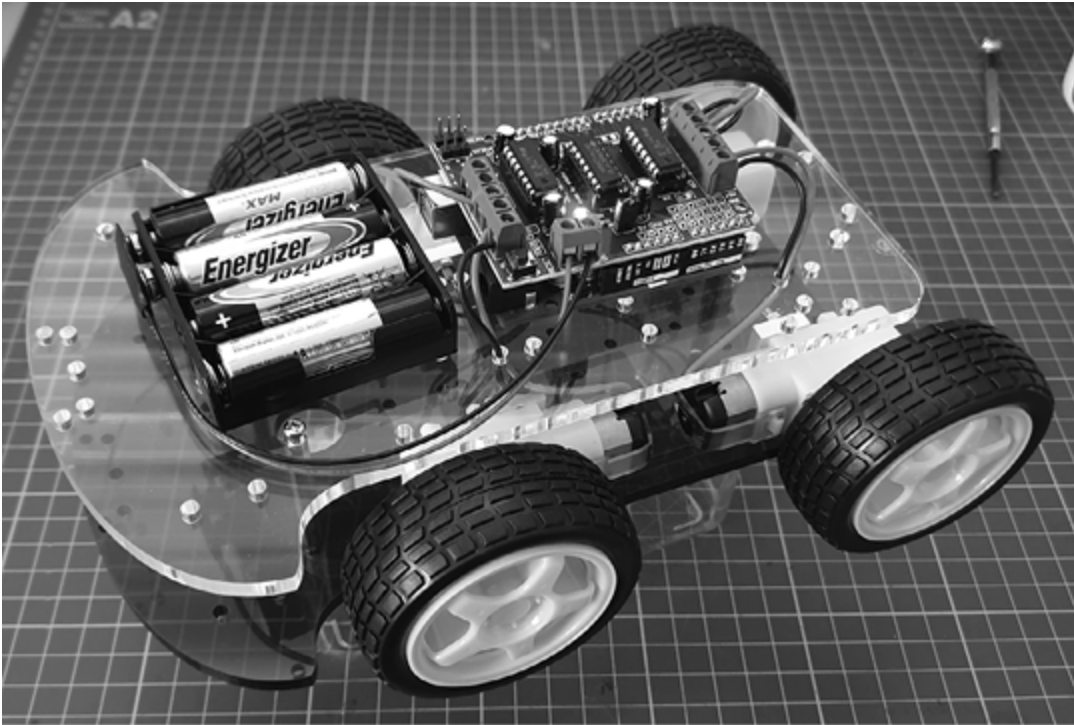


Figure 14-20: Our robot is ready for action!

The Sketch

Now to get the robot moving. To simplify its operation, we first need to download and install the Arduino library for the motor drive shield. Follow the instructions outlined in Chapter 7. In the Library Manager, find and then install the “Adafruit Motor Shield library by Adafruit.”

After a moment, the Adafruit Motor Shield library v1 will appear. Click **Install** and wait for the library to be installed. You can then close the Library Manager window.

Now we’ll create some functions to operate our robot. Because two motors are involved, we’ll need four movements:

Forward motion

Reverse motion

Rotate clockwise

Rotate counterclockwise

Thus, we'll need four functions in our sketch to match our four movements: `goForward()`, `goBackward()`, `rotateLeft()`, and `rotateRight()`. Each accepts a value in milliseconds, which is the length of time required to operate the movement, and a PWM value between 0 and 255. For example, to move forward for 2 seconds at full speed, we'd use `goForward(255, 2000)`.

Enter and save the following sketch (but don't upload it just yet):

```
// Project 39 - Building and Controlling a Robot Vehicle
#include <AFMotor.h>

1 AF_DCMotor motor1(1); // set up instances of each motor
  AF_DCMotor motor2(2);
  AF_DCMotor motor3(3);
  AF_DCMotor motor4(4);

2 void goForward(int speed, int duration)
  {
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(FORWARD);
    motor2.run(FORWARD);
    motor3.run(FORWARD);
    motor4.run(FORWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
  }

2 void goBackward(int speed, int duration)
  {
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(BACKWARD);
    motor2.run(BACKWARD);
    motor3.run(BACKWARD);
    motor4.run(BACKWARD);
    delay(duration);
  }
```

```
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}
```

```
2 void rotateLeft(int speed, int duration)
{
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(FORWARD);
    motor2.run(BACKWARD);
    motor3.run(BACKWARD);
    motor4.run(FORWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}
```

```
2 void rotateRight(int speed, int duration)
{
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(BACKWARD);
    motor2.run(FORWARD);
    motor3.run(FORWARD);
    motor4.run(BACKWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}
```

```
void setup()
{
    delay(5000);
}
```

```
void loop()
{
```

```
    goForward(127, 5000);  
    delay(1000);  
    rotateLeft(127, 2000);  
    delay(1000);  
    goBackward(127, 5000);  
    delay(1000);  
    rotateRight(127, 5000);  
    delay(5000);  
}
```

Controlling the robot is easy thanks to the four custom functions in the sketch. Each custom function makes use of the library functions used to control a motor. Before you can use these functions, you need to create an instance for each motor, as shown at 1.

The direction of travel for each motor is set using:

```
Motor.run(direction)
```

The value of *direction* is either FORWARD, REVERSE, or RELEASE, to set the motor's rotational direction forward or backward or cut power to the motor, respectively.

To set the speed of the motor, we use:

```
Motor.setSpeed(speed)
```

The value of *speed* is between 0 and 255; it is the range of PWM used to control the rotational speed of the motor.

Therefore, in each of our four custom functions at 2, we use the combination of the motor speed and directional controls to control all four motors at once. Each of the custom functions accepts two parameters: speed (our PWM value) and duration (the amount of time to run the motor).

WARNING

When you're ready to upload the sketch, position the robot either by holding it off your work surface or by propping it up so that its treads aren't in contact with a surface. If you don't do this, then when the sketch upload completes, the robot will burst into life and might leap off your desk immediately!

Upload the sketch, remove the USB cable, and connect the battery cable to the Arduino power socket. Then place the robot on carpet or a clean surface and let it drive about. Experiment with the movement functions in the sketch to control your robot; this will help you become familiar with the time delays and how they relate to distance traveled.

Connecting Extra Hardware to the Robot

Some motor drive shields for Arduino may not have stacking header sockets to enable you to put another shield on top, and they might not allow easy connection of wires from sensors, etc. In this case, you should use a *terminal shield* for Arduino, an example of which is shown in [Figure 14-21](#).

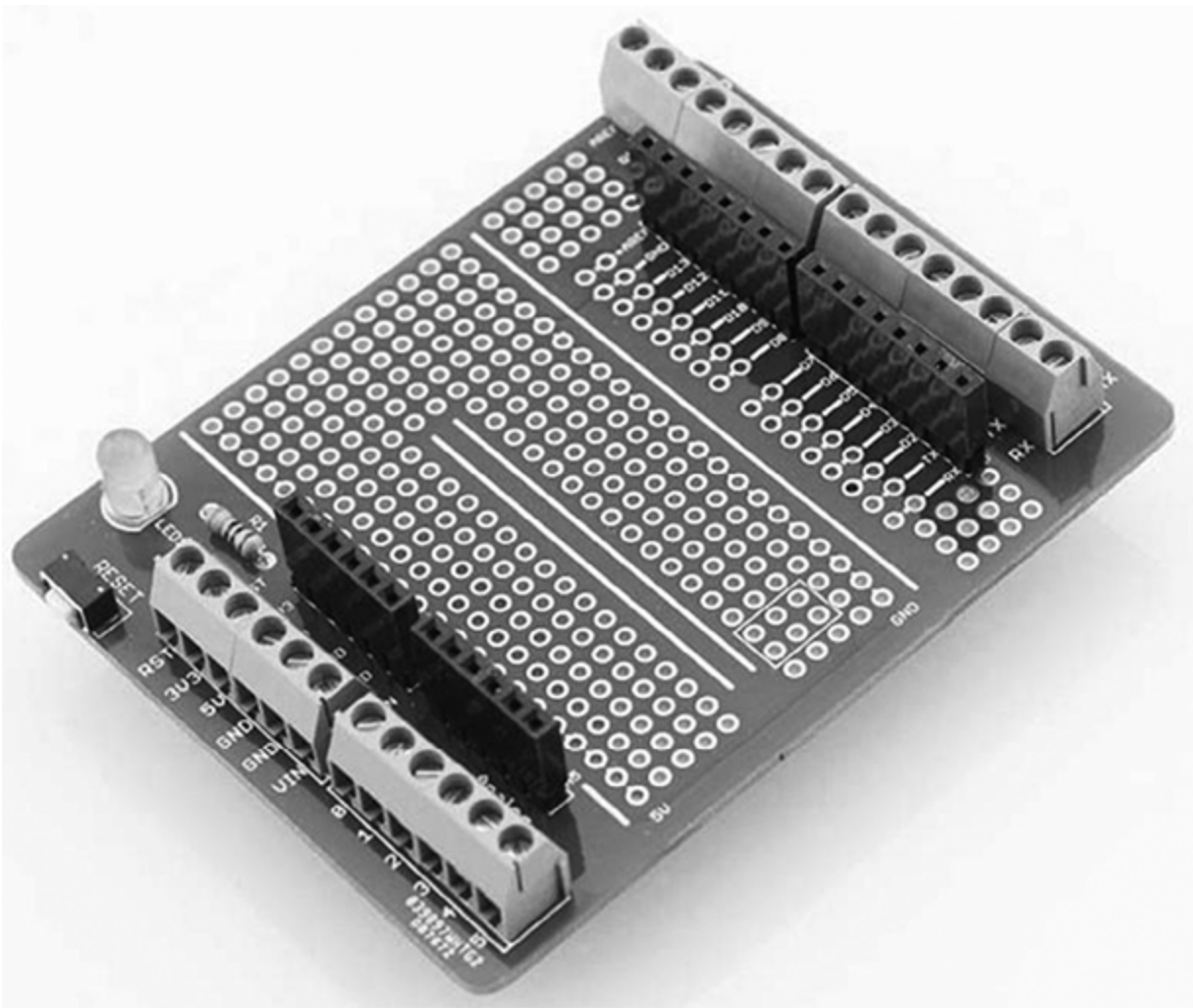


Figure 14-21: A terminal shield for Arduino

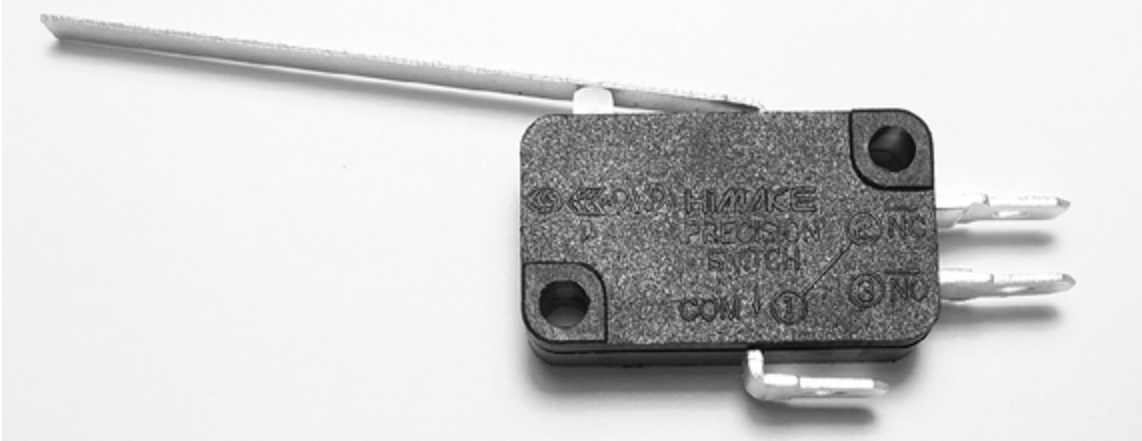
Terminal shields allow for easy wiring of hardware or sensors to the Arduino's input and output pins without any soldering, and they can also be used to build your own circuitry for more permanent uses later.

Sensing Collisions

Now that our robot can move, we can start to add sensors. These will tell the robot when it has bumped into something, or they will measure the distance between the robot and an object in its path so that it can avoid a crash. We'll use three methods of collision avoidance: microswitches, infrared, and ultrasonic.

Project #40: Detecting Robot Vehicle Collisions with a Microswitch

A *microswitch* can act like the simple push button we used in Chapter 4, but the microswitch component is physically larger and includes a large metal bar that serves as the actuator (see [Figure 14-22](#)).



[Figure 14-22](#): A microswitch

When using a microswitch, you connect one wire to the bottom contact and the other to the contact labeled NO (normally open) to ensure that current flows only when the bar is pressed. We'll mount the microswitch on the front of our robot so that when the robot hits an object, the bar will be pressed, causing current to flow and making the robot reverse direction or take another action.

The Schematic

The microswitch hardware is wired like a single push button, as shown in [Figure 14-23](#).

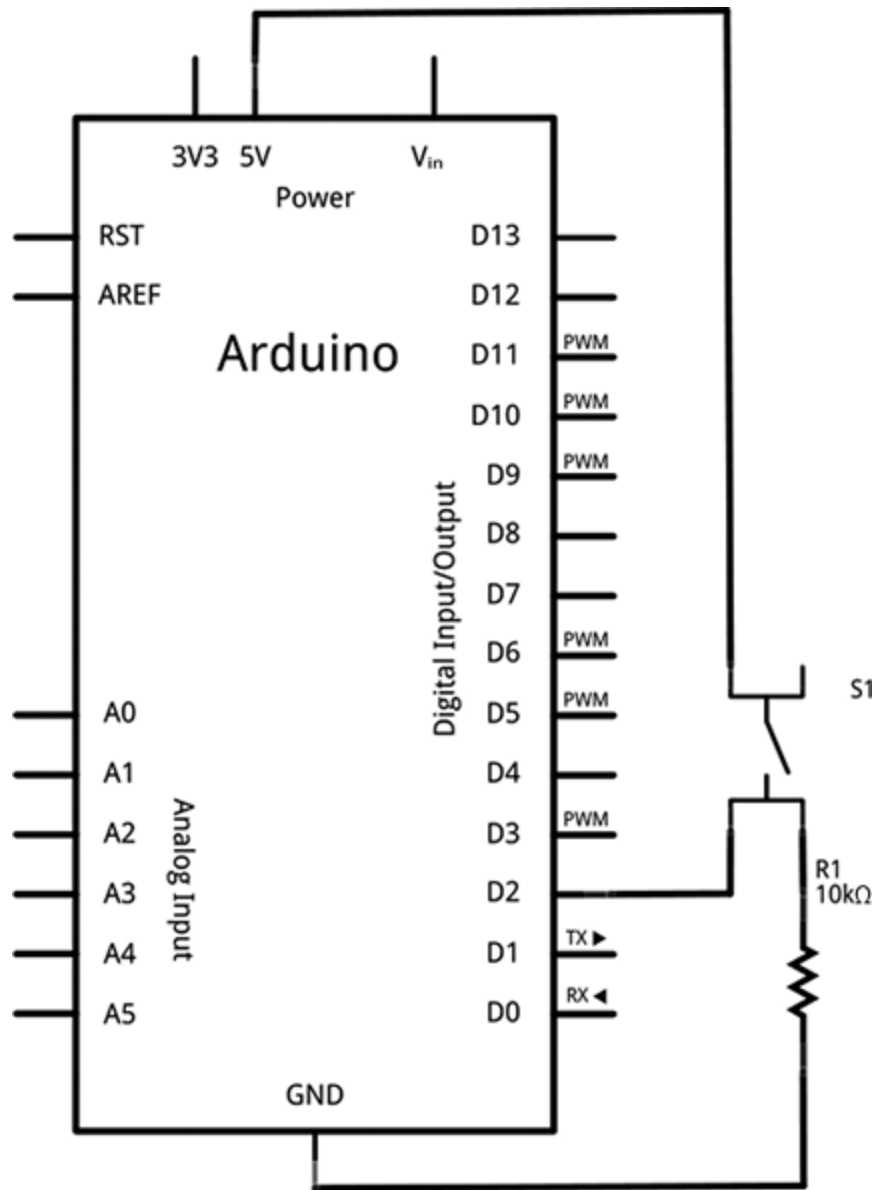


Figure 14-23: Schematic for Project 40

The Sketch

We connect the microswitch to an interrupt port (digital pin 2). Although you might think we should have a function called by the interrupt to make the robot reverse for a few moments, that's not possible, because the `delay()` function doesn't operate inside functions called by interrupts. We must think a little differently in this case.

Instead, the function `goForward()` will turn on the motors if two conditions are met for the variables `crash` and the Boolean `move`. If `crash` is true, the

motors will reverse at a slower speed for 2 seconds to back away from a collision situation.

We can't use `delay()` because of the interrupt, so we measure the amount of time that the motors run by reading `millis()` at the start and comparing that against the current value of `millis()`. When the difference is greater than or equal to the required duration, `move` is set to `false` and the motors stop.

Enter and upload the following sketch:

```
// Project 40 - Detecting Robot Vehicle Collisions with a
Microswitch
#include <AFMotor.h>

AF_DCMotor motor1(1); // set up instances of each motor
AF_DCMotor motor2(2);
AF_DCMotor motor3(3);
AF_DCMotor motor4(4);

boolean crash = false;

void goBackward(int speed, int duration)
{
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(BACKWARD);
    motor2.run(BACKWARD);
    motor3.run(BACKWARD);
    motor4.run(BACKWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}

1 void backOut()
{
    crash = true;
}

void goForward(int duration, int speed)
{
```

```

    long a, b;
    boolean move = true;
2   a = millis();
    do
    {
        if (crash == false)
        {
            motor1.setSpeed(speed);
            motor2.setSpeed(speed);
            motor3.setSpeed(speed);
            motor4.setSpeed(speed);
            motor1.run(FORWARD);
            motor2.run(FORWARD);
            motor3.run(FORWARD);
            motor4.run(FORWARD);
        }
        if (crash == true)
        {
3           goBackward(200, 2000);
            crash = false;
        }
4       b = millis() - a;
        if (b >= duration)
        {
            move = false;
        }
    }
    while (move != false);
    // stop motors
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}

void setup()
{
    attachInterrupt(0, backOut, RISING);
    delay(5000);
}

void loop()
{
    goForward(5000, 127);
    delay(2000);
}

```

This sketch uses an advanced method of moving forward, in that two variables are used to monitor movement while the robot is in motion. The first is the Boolean variable `crash`. If the robot bumps into something and activates the microswitch, then an interrupt is called, which runs the function `backOut()` at 1. It is here that the variable `crash` is changed from `false` to `true`. The second variable that is monitored is the Boolean variable `move`. In the function `goForward()`, we use `millis()` at 2 to calculate constantly whether the robot has finished moving for the required period of time (set by the parameter `duration`).

At 4, the function calculates whether the elapsed time is less than the required time, and if so, the variable `move` is set to `true`. Therefore, the robot is allowed to move forward only if it has not crashed and not run out of time. If a crash has been detected, the function `goBackward()` at 3 is called, at which point the robot will reverse slowly for 2 seconds and then resume as normal.

NOTE

You can add the other movement functions from Project 39 to expand or modify this example.

Infrared Distance Sensors

Our next method of collision avoidance uses an infrared (IR) distance sensor. This sensor bounces an infrared light signal off a surface in front of it and returns a voltage that is relative to the distance between the sensor and the surface. Infrared sensors are useful for collision detection because they are inexpensive, but they're not ideal for *exact* distance measuring. We'll use the Sharp GP2Y0A21YK0F analog sensor, shown in [Figure 14-24](#), for our project.



Figure 14-24: The Sharp IR sensor

Wiring It Up

To wire the sensor, connect the red and black wires on the sensor to 5 V and GND, respectively, with the white wire connecting to an analog input pin on your Arduino. We'll use `analogRead()` to measure the voltage returned from the sensor. The graph in [Figure 14-25](#) shows the relationship between the distance measured and the output voltage.

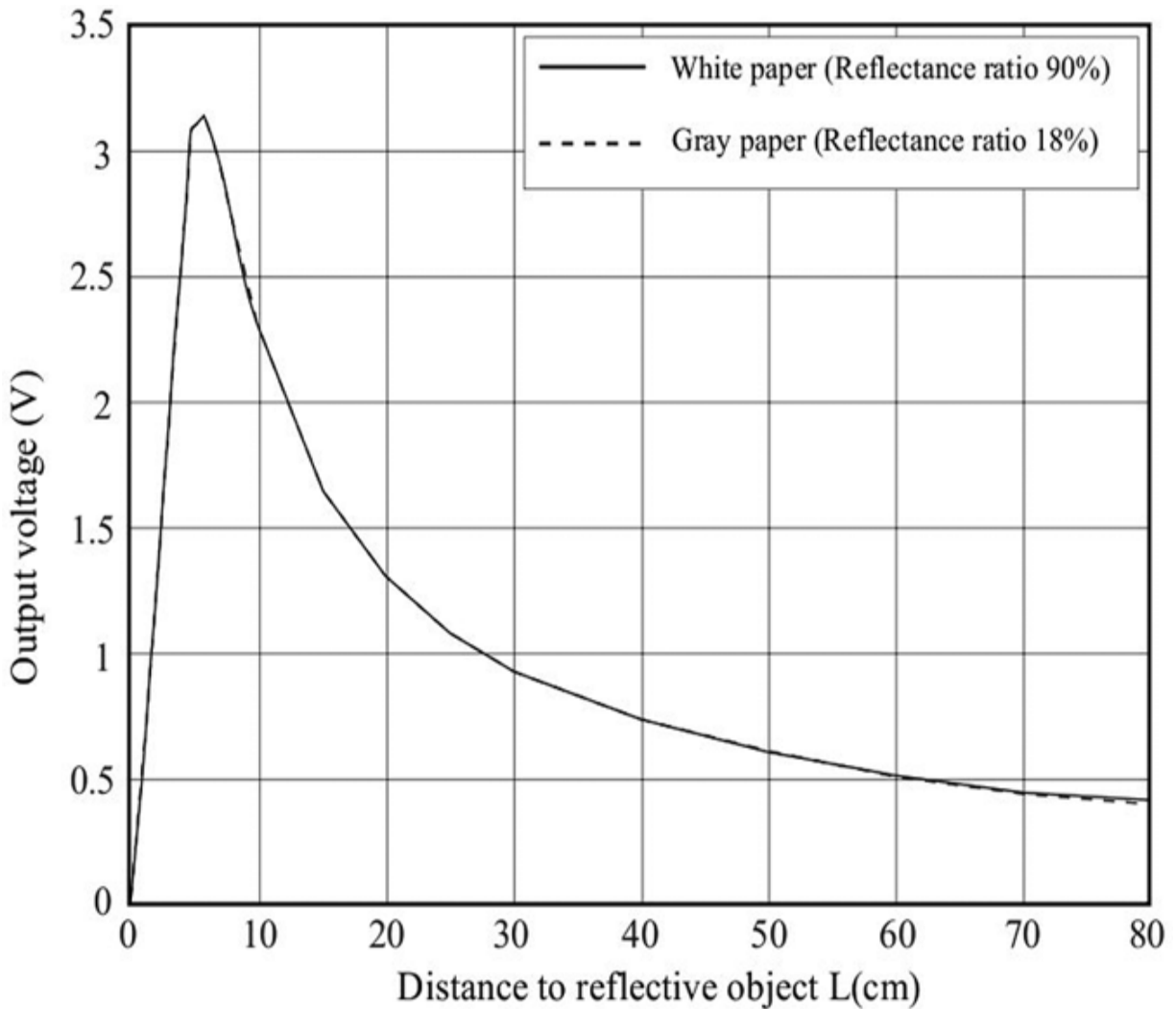


Figure 14-25: Graph of IR sensor distance versus output voltage

Testing the IR Distance Sensor

Because the relationship between distance and output is not easily represented with an equation, we'll categorize the readings into 5 cm stages. To demonstrate this, we'll use a simple example. Connect your infrared sensor's white lead to analog pin 0, the red lead to 5 V, and the black lead to GND. Then enter and upload the sketch shown in [Listing 14-3](#).

// Listing 14-3

```
float sensor = 0;  
int cm = 0;
```

```

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

void loop()
{
1  sensor = analogRead(0);
2  if (sensor<=90)
    {
      Serial.println("Infinite distance!");
    } else if (sensor<100) // 80 cm
    {
      cm = 80;
    } else if (sensor<110) // 70 cm
    {
      cm = 70;
    } else if (sensor<118) // 60 cm
    {
      cm = 60;
    } else if (sensor<147) // 50 cm
    {
      cm = 50;
    } else if (sensor<188) // 40 cm
    {
      cm = 40;
    } else if (sensor<230) // 30 cm
    {
      cm = 30;
    } else if (sensor<302) // 25 cm
    {
      cm = 25;
    } else if (sensor<360) // 20 cm
    {
      cm = 20;
    } else if (sensor<505) // 15 cm
    {
      cm = 15;
    } else if (sensor<510) // 10 cm
    {
      cm = 10;
    } else if (sensor>=510) // too close!
    {
      Serial.println("Too close!");
    }
    Serial.print("Distance: ");
    Serial.print(cm);

```



```
Serial.println(" cm");  
delay(250);  
}
```

Listing 14-3: IR sensor demonstration sketch

The sketch reads the voltage from the IR sensor at 1 and then uses a series of if statements at 2 to choose which approximate distance is being returned. We determine the distance from the voltage returned by the sensor using two parameters. The first is the voltage-to-distance relationship, as displayed in [Figure 14-25](#). Then, using the knowledge (from Project 6 in Chapter 4) that `analogRead()` returns a value between 0 and 1,023 relative to a voltage between 0 V and around 5 V, we can calculate the approximate distance returned by the sensor.

After uploading the sketch, open the Serial Monitor and experiment by moving your hand or a piece of paper at various distances from the sensor. The Serial Monitor should return the approximate distance, as shown in [Figure 14-26](#).

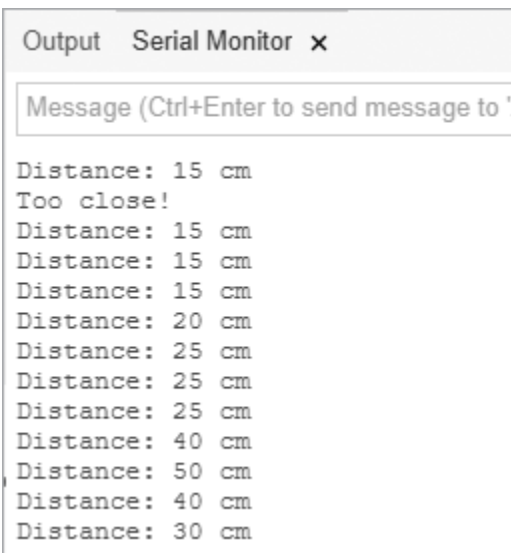


Figure 14-26: Results of [Listing 14-3](#)

Project #41: Detecting Robot Vehicle Collisions with an IR Distance Sensor

Now let's use the IR sensor with our robot vehicle instead of the microswitch. We'll use a slightly modified version of Project 40. Instead of using an interrupt, we'll create the function `checkDistance()`, which changes the variable `crash` to `true` if the distance measured by the IR sensor is around 20 cm or less. We'll use this in the `goForward()` forward motion do-while loop.

The Sketch

Connect the IR sensor to your robot and then enter and upload this sketch:

```
// Project 41 - Detecting Robot Vehicle Collisions with an IR
Distance Sensor
#include <AFMotor.h>

AF_DCMotor motor1(1); // set up instances of each motor
AF_DCMotor motor2(2);
AF_DCMotor motor3(3);
AF_DCMotor motor4(4);
boolean crash = false;

void goBackward(int speed, int duration)
{
  motor1.setSpeed(speed);
  motor2.setSpeed(speed);
  motor3.setSpeed(speed);
  motor4.setSpeed(speed);
  motor1.run(BACKWARD);
  motor2.run(BACKWARD);
  motor3.run(BACKWARD);
  motor4.run(BACKWARD);
  delay(duration);
  motor1.run(RELEASE);
  motor2.run(RELEASE);
  motor3.run(RELEASE);
  motor4.run(RELEASE);
}

void checkDistance()
{
1  if (analogRead(0) > 460)
    {
      crash = true;
    }
}
```

```

}

void goForward(int duration, int speed)
{
    long a, b;
    boolean move = true;
    a = millis();
    do
    {
        checkDistance();
        if (crash == false)
        {
            motor1.setSpeed(speed);
            motor2.setSpeed(speed);
            motor3.setSpeed(speed);
            motor4.setSpeed(speed);
            motor1.run(FORWARD);
            motor2.run(FORWARD);
            motor3.run(FORWARD);
            motor4.run(FORWARD);
        }
        if (crash == true)
        {
            goBackward(200, 2000);
            crash = false;
        }
        b = millis() - a;
        if (b >= duration)
        {
            move = false;
        }
    }
    while (move != false);
    // stop motors
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}

void setup()
{
    delay(5000);
}

void loop()
{
    goForward(5000, 255);
}

```

```
    delay(2000);  
}
```

This sketch operates using the same methods used in Project 40, except this version constantly takes distance measurements at 1 and sets the crash variable to `true` if the distance between the IR sensor and an object is less than about 20 cm.

Modifying the Sketch: Adding More Sensors

After running the robot and using this sensor, you should see the benefits of using a non-contact collision sensor. It's simple to add more sensors to the same robot, such as sensors at the front and rear or at each corner. You should be able to add code to check each sensor in turn and make a decision based on the returned distance value.

Ultrasonic Distance Sensors

Our final method of collision avoidance uses an *ultrasonic distance sensor*. This sensor bounces a sound wave of an ultra-high frequency (that cannot be heard by the human ear) off a surface and measures the amount of time it takes for the sound to return to the sensor. We'll use the common HC-SR04-type ultrasonic distance sensor, shown in [Figure 14-27](#), for this project, because it's inexpensive and accurate to around 2 cm.



Figure 14-27: The HC-SR04 ultrasonic distance sensor

An ultrasonic sensor's accuracy and range mean it can measure distances between about 2 and 450 cm. However, because the sound wave needs to be reflected back to the sensor, the sensor must be angled less than 15 degrees away from the direction of travel.

Connecting the Ultrasonic Sensor

To connect the sensor, attach the Vcc (5 V) and GND leads to their connectors on the motor drive shield, attach the Trig pin to digital pin D2, and attach the Echo pin to digital pin D13. We use D2 and D13 as they are not used by the motor drive shield. However, if you're just testing or experimenting with the sensor without the robot, you can connect the wires directly to your Arduino board.

To simplify operation of the sensor, download the Arduino library from <https://github.com/Martinsos/arduino-lib-hc-sr04/archive/master.zip> and install it as explained in Chapter 7. Once the library is installed, you can run the test sketch in [Listing 14-4](#) to see how the sensor works.

```
// Listing 14-4
#include <HCSR04.h>
```

```
1 UltraSonicDistanceSensor HCSR04(2, 13); // trig - D2, echo -  
  D13  
2 float distance;  
  
  void setup ()  
  {  
    Serial.begin(9600);  
  }  
  
  void loop ()  
  {  
3    distance = HCSR04.measureDistanceCm();  
      Serial.print("Distance: ");  
      Serial.print(distance);  
      Serial.println(" cm");  
      delay(500);  
  }
```

Listing 14-4: Ultrasonic sensor demonstration sketch

Retrieving the distance from the sensor is quite simple thanks to the library. At 1, we create an instance and declare which digital pins are connected to the sensor. Then at 2, we have a floating-point variable used to store the distance returned from the sensor's library function. Finally, the distance is generated at 3 for display in the Serial Monitor.

Testing the Ultrasonic Sensor

After uploading the sketch, open the Serial Monitor and move an object toward and away from the sensor. The distance to the object should be returned in centimeters. See how it works in [Figure 14-28](#).

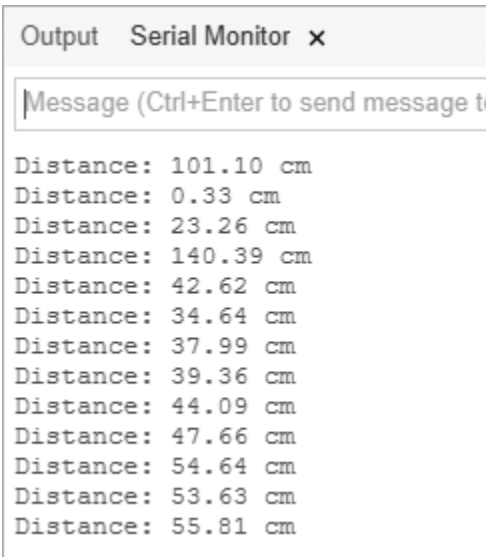


Figure 14-28: Results from [Listing 14-4](#)

Project #42: Detecting Collisions with an Ultrasonic Distance Sensor

Now that you understand how the sensor works, let's use it with our robot.

The Sketch

In the following sketch, we check for distances between the robot and an object of 5 cm or less, which will give the robot a reason to back up. Enter and upload the following sketch to see for yourself:

```
// Project 42 - Detecting Collisions with an Ultrasonic
Distance Sensor
#include <AFMotor.h>
#include <HCSR04.h>

// set up instances of each motor
AF_DCMotor motor1(1);
AF_DCMotor motor2(2);
AF_DCMotor motor3(3);
AF_DCMotor motor4(4);

// set up ultrasonic sensor
UltrasonicDistanceSensor HCSR04(2, 13); // trig - D2, echo -
D13
```

```

boolean crash=false;

void checkDistance()
{
    float distance;
    distance = HCSR04.measureDistanceCm();
1   if (distance < 5) // crash distance is 5 cm or less
    {
        crash = true;
    }
}

void goBackward(int speed, int duration)
{
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(BACKWARD);
    motor2.run(BACKWARD);
    motor3.run(BACKWARD);
    motor4.run(BACKWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}

void goForward(int duration, int speed)
{
    long a, b;
    boolean move = true;
    a = millis();
    do
    {
        checkDistance();
        if (crash == false)
        {
            motor1.setSpeed(speed);
            motor2.setSpeed(speed);
            motor3.setSpeed(speed);
            motor4.setSpeed(speed);
            motor1.run(FORWARD);
            motor2.run(FORWARD);
            motor3.run(FORWARD);

```



```

        motor4.run(FORWARD);
    }
    if (crash == true)
    {
        goBackward(200, 2000);
        crash = false;
    }
    b = millis() - a;
    if (b >= duration)
    {
        move = false;
    }
}
while (move != false);
// stop motors
motor1.run(RELEASE);
motor2.run(RELEASE);
motor3.run(RELEASE);
motor4.run(RELEASE);
}

void setup()
{
    delay(5000);
}

void loop()
{
    goForward(1000, 255);
}

```

The operation of this sketch should be quite familiar by now. Once again, we constantly measure the distance at 1 and then change the variable `crash` to `true` if the distance between the ultrasonic sensor and an object in its path is less than 5 cm. Watching the robot magically avoid colliding with things or having a battle of wits with a pet can be quite amazing.

Looking Ahead

In this chapter, you learned how to introduce your Arduino-based projects to the world of movement. Using simple motors, or pairs of motors, with a motor shield, you can create projects that can move on their own and even avoid obstacles. We used three types of sensors to demonstrate a range of

accuracies and sensor costs, so you can now make decisions based on your requirements and project budget.

By now, I hope you are experiencing and enjoying the ability to design and construct such things. But it doesn't stop here. In the next chapter, we move outdoors and harness the power of satellite navigation.

15

USING GPS WITH YOUR ARDUINO

In this chapter, you will

Learn how to connect a GPS shield

Create a simple GPS coordinates display

Show the actual position of GPS coordinates on a map

Build an accurate clock

Record the position of a moving object over time

You'll learn how to use an inexpensive GPS shield to determine location, create an accurate clock, and make a logging device that records the position of your gadget over time onto a microSD card, which can then be plotted over a map to display movement history.

What Is GPS?

The *Global Positioning System (GPS)* is a satellite-based navigation system that sends data from satellites orbiting Earth to GPS receivers on the ground that can use that data to determine the current position and time anywhere on Earth. You are probably already familiar with GPS navigation devices used in cars or on your smartphone.

Although we can't create detailed map navigation systems with our Arduinos, you can use a GPS module to determine your position, time, and approximate speed (if you're in motion). When shopping around for a GPS module, you will generally find two types available. The first is an

independent, inexpensive GPS receiver on a module with an external aerial, as shown in [Figure 15-1](#).

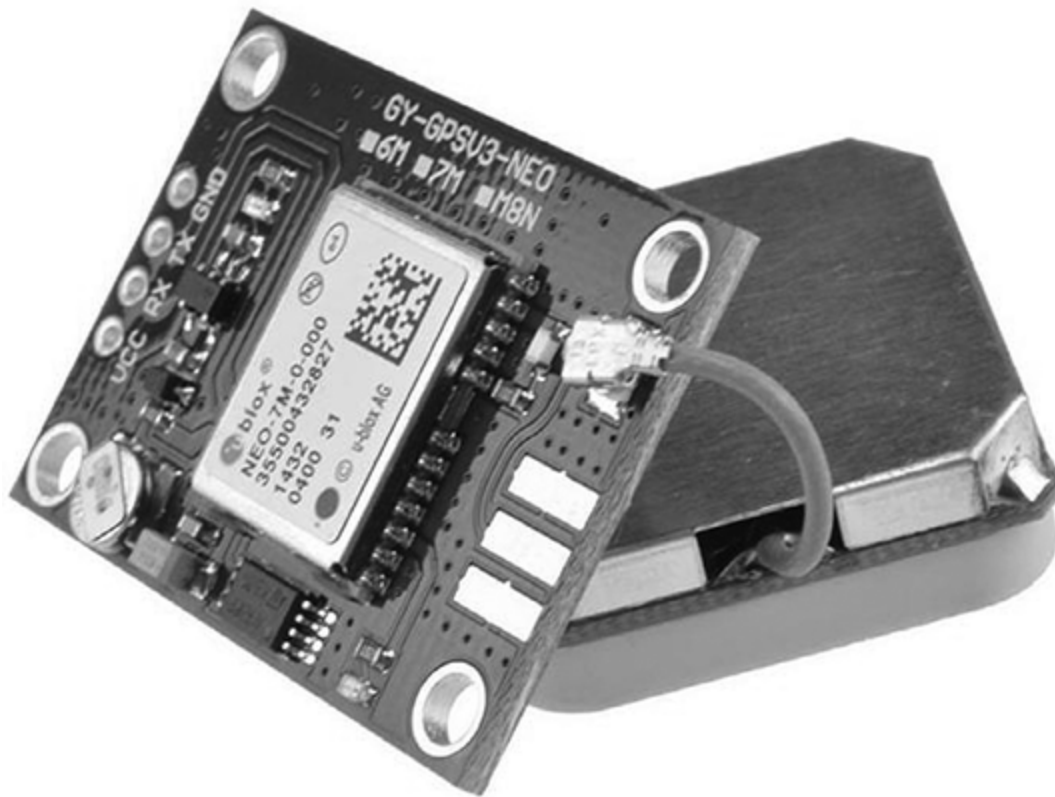


Figure 15-1: A GPS receiver module

The second type you will come across is a GPS shield for Arduino, as shown in [Figure 15-2](#). These shields are convenient, since all the wiring is done for you; they also include a microSD card socket that is ideal for logging data, as demonstrated later in this chapter.

Ensure your GPS shield allows connection of the GPS receiver's TX and RX lines to Arduino digital pins D2 and D3, or has jumpers to allow manually setting these (like the shield in [Figure 15-2](#)). Check with the supplier for more details. You can use either type of device in this chapter. However, I highly recommend the shield, especially as you can effortlessly connect an LCD shield on top of the GPS shield as a display.

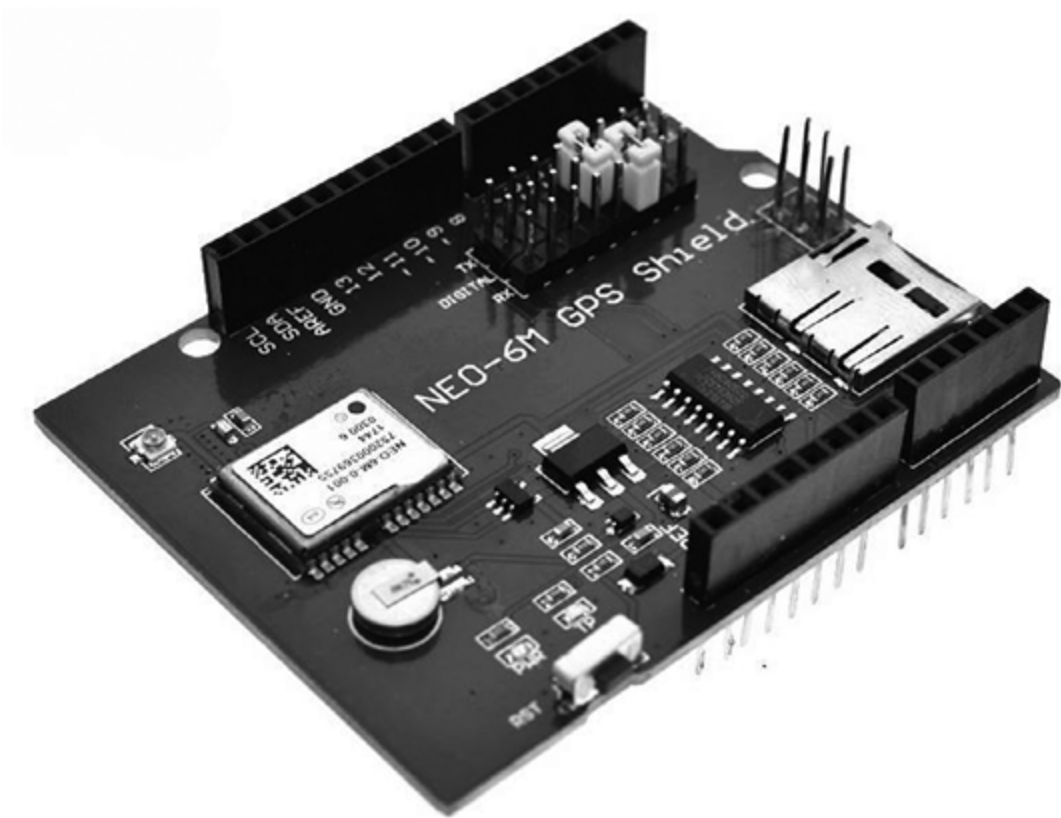


Figure 15-2: A GPS shield for Arduino

USING A SOFTWARE SERIAL PORT

To make use of our GPS modules, whose data is sent to the Arduino via a serial port, we need to use a “software serial port.” Along with the normal serial port on your Arduino, found on digital pins D0 and D1, you can also allocate two other digital pins to be another serial port. This is emulated in software thanks to the SoftwareSerial Arduino library. This arrangement allows you to use two devices that require a serial connection at once without causing problems.

Creating a software serial port is simple: you first include the library and then create another port:

```
#include <SoftwareSerial.h>
SoftwareSerial Serial2(x, y);
```

where x is the digital pin for transmit (TX) and y is the digital pin for receive (RX).

You can then easily use the software serial port just like the hardware serial port. For example, to initialize it, use the following line in void setup():

```
Serial2.begin(9600);
```

You will see the software serial port demonstrated throughout the sketches in this chapter.

Testing the GPS Shield

After you buy a GPS kit, it’s a good idea to make sure that it’s working and that you can receive GPS signals. GPS receivers require a line of sight to the sky, but their signals can pass through windows. So, while it’s usually best to perform this test outdoors, your GPS receiver will probably work just fine through an unobstructed window or skylight. To test reception, you’ll set up the shield or module and run a basic sketch that displays the raw received data.

If you are using a GPS shield, ensure that the GPS TX pin is jumpered to Arduino digital pin D2 and the RX pin is jumpered to Arduino digital pin D3. If you are using a GPS module, as shown in [Figure 15-1](#), connect the Vcc and GND to Arduino 5 V and GND, respectively; then connect TX to Arduino digital pin D2 and RX to Arduino digital pin D3.

To perform the test, enter and upload the sketch in [Listing 15-1](#).

```
// Listing 15-1
#include <SoftwareSerial.h>

// GPS TX to D2, RX to D3
SoftwareSerial Serial2(2, 3);

byte gpsData;

void setup()
{
  // Open the Arduino Serial Monitor
  Serial.begin(9600);
  // Open the GPS
1  Serial2.begin(9600);
  }

void loop()
{
2  while (Serial2.available() > 0)
    {
      // get the byte data from the GPS
      gpsData = Serial2.read();
3    Serial.write(gpsData);
    }
}
```

Listing 15-1: Basic GPS test sketch

This sketch listens to the software serial port at 2, and when a byte of data is received from the GPS module or shield, it is sent to the Serial Monitor at 3. (Notice that we start the software serial port at 9,600 bps at 1 to match the data speed of the GPS receiver.)

Once you've uploaded the sketch, you may need to wait around 30 seconds; this is to allow the GPS receiver time to start receiving signals from one or more GPS satellites. The GPS shield or module will have an onboard LED, which will start flashing once the receiver has started finding GPS signals. After the LED starts blinking, open the Serial Monitor window in the IDE and set the data speed to 9,600 baud. You should see a constant stream of data similar to the output shown in [Figure 15-3](#).

```

$GPRMC,002807.00,A,2734.93850,S,15305.76781,E,0.013,,030820,,,A*63
$GPVTG,,T,,M,0.013,N,0.024,K,A*27
$GPGGA,002807.00,2734.93850,S,15305.76781,E,1,08,0.95,42.4,M,38.3,M,,*7A
$GPGSA,A,3,22,03,04,26,09,16,07,27,,,,,1.81,0.95,1.55*08
$GPGSV,3,1,12,01,04,346,,03,63,355,30,04,58,177,33,06,07,244,10*70
$GPGSV,3,2,12,07,26,282,35,08,06,029,12,09,33,222,25,16,56,099,33*78
$GPGSV,3,3,12,22,40,010,28,26,26,134,28,27,10,060,32,30,00,294,*76
$GPGLL,2734.93850,S,15305.76781,E,002807.00,A,A*71
$GPRMC,002808.00,A,2734.93856,S,15305.76783,E,0.026,,030820,,,A*6E
$GPVTG,,T,,M,0.026,N,0.049,K,A*2A
$GPGGA,002808.00,2734.93856,S,15305.76783,E,1,08,0.95,42.5,M,38.3,M,,*70
$GPGSA,A,3,22,03,04,26,09,16,07,27,,,,,1.81,0.95,1.55*08
$GPGSV,3,1,12,01,04,346,,03,63,355,29,04,58,177,33,06,07,244,09*70
$GPGS

```

Figure 15-3: Raw data from GPS satellites

The data is sent from the GPS receiver to the Arduino one character at a time, and then it is sent to the Serial Monitor. But this raw data (called *GPS sentences*) is not very useful as it is, so we need to use a new library that extracts information from the raw data and converts it to a usable form. To do this, download and install the TinyGPS library from <http://www.arduiniiana.org/libraries/tinygps/> using the method described in Chapter 7.

Project #43: Creating a Simple GPS Receiver

We'll start by creating a simple GPS receiver. Because you'll usually use your GPS outdoors—and to make things a little easier—we'll add an LCD module to display the data, similar to the one shown in [Figure 15-4](#).



Figure 15-4: The Freetronics LCD & Keypad Shield

NOTE

Our examples are based on using the Freetronics LCD & Keypad Shield. For more information on this shield, see <http://www.freetronics.com.au/collections/display/products/lcd-keypad-shield/>. If you choose to use a different display module, be sure to substitute the correct values into the `LiquidCrystal()` function in your sketches.

The result will be a very basic portable GPS that can be powered by a 9 V battery and connector, which will display the coordinates of your current position on the LCD.

The Hardware

The required hardware is minimal:

Arduino and USB cable

LCD module or Freetronics LCD & Keypad Shield

One 9 V battery-to-DC socket cable

GPS module and screw shield for Arduino or GPS shield for Arduino

The Sketch

Enter and upload the following sketch:

```
// Project 43 - Creating a Simple GPS Receiver
1 #include <LiquidCrystal.h>
  LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

  #include <TinyGPS.h>
  #include <SoftwareSerial.h>

  // GPS TX to D2, RX to D3

  SoftwareSerial Serial2(2, 3);

  TinyGPS gps;
  void getgps(TinyGPS &gps);
```

```

byte gpsData;

2 void getgps(TinyGPS &gps)
  // The getgps function will display the required data on the
  LCD
  {
    float latitude, longitude;
    // decode and display position data
3   gps.f_get_position(&latitude, &longitude);
    lcd.setCursor(0, 0);
    lcd.print("Lat:");
    lcd.print(latitude, 5);
    lcd.print(" ");
    lcd.setCursor(0, 1);
    lcd.print("Long:");
    lcd.print(longitude, 5);
    lcd.print(" ");
    delay(3000); // wait for 3 seconds
    lcd.clear();
  }

void setup()
{
  Serial2.begin(9600);
}

void loop()
{
  while (Serial2.available() > 0)
  {
    // get the byte data from the GPS
    gpsData = Serial2.read();
    if (gps.encode(gpsData))
    {
4      getgps(gps);
    }
  }
}

```

From 1 to 2, the sketch introduces the required libraries for the LCD and GPS. In `void loop()`, we send the characters received from the GPS receiver to the function `getgps()` at 4. The data is obtained by using `gps.f_get_position()` at 3 to insert the position values in the byte variables `&latitude` and `&longitude`, which we display on the LCD.

Running the Sketch

After the sketch has been uploaded and the GPS starts receiving data, your current position in decimal latitude and longitude should be displayed on your LCD, as shown in [Figure 15-5](#).

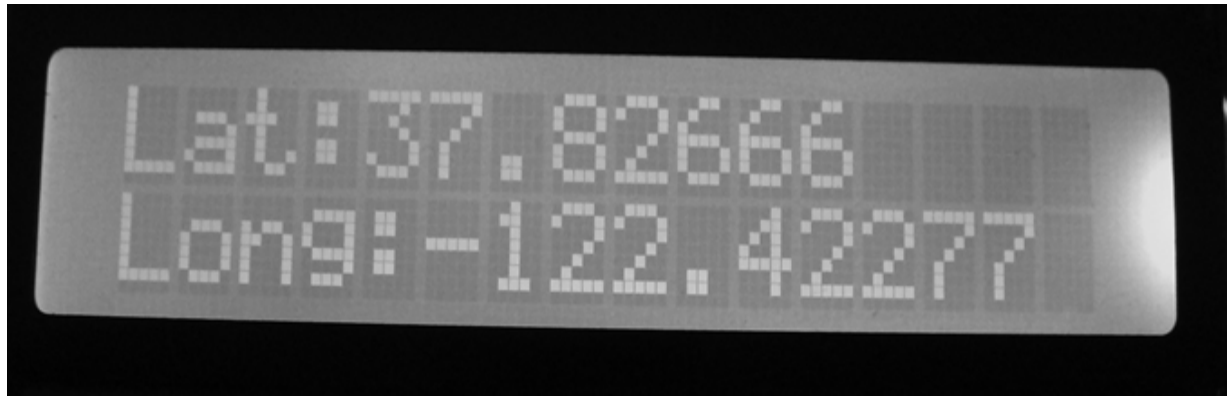


Figure 15-5: Latitude and longitude display from Project 43

But where on Earth is this? We can determine exactly where it is by using Google Maps (<http://maps.google.com/>). On the website, enter the latitude and longitude, separated by a comma and a space, into the search field, and Google Maps will return the location. For example, using the coordinates returned in [Figure 15-5](#) produces a map like the one shown in [Figure 15-6](#).

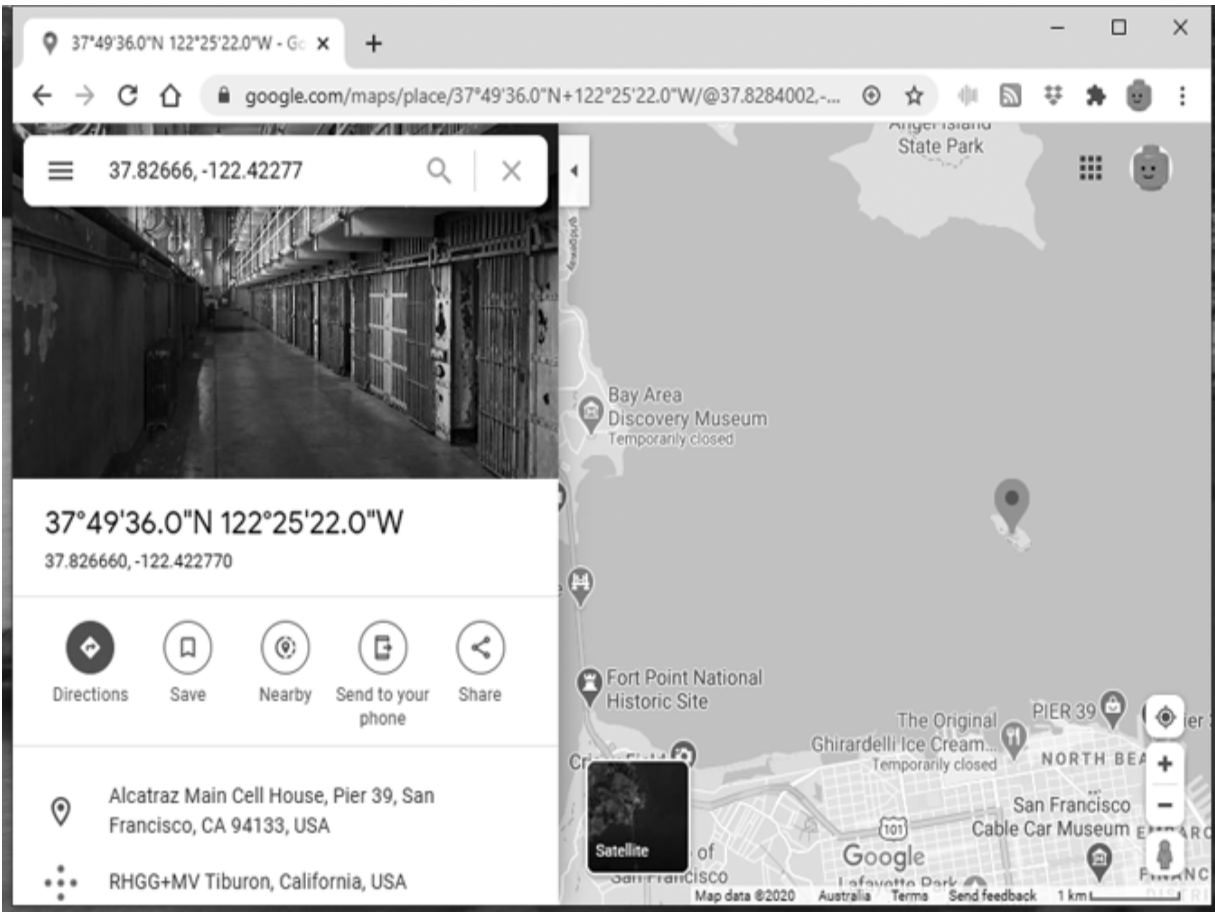


Figure 15-6: The GPS coordinates displayed in [Figure 15-5](#) place us on Alcatraz Island.

Project #44: Creating an Accurate GPS-Based Clock

There is more to using a GPS than finding a location; the system also transmits time data that can be used to make a very accurate clock.

The Hardware

For this project, we'll use the same hardware as in Project 43.

The Sketch

Enter and upload the following sketch to build a GPS clock:

```

// Project 44 - Creating an Accurate GPS-Based Clock
#include <LiquidCrystal.h>
#include <TinyGPS.h>
#include <SoftwareSerial.h>
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

// GPS RX to D3, GPS TX to D2
SoftwareSerial Serial2(2, 3);

TinyGPS gps;
void getgps(TinyGPS &gps);

byte gpsData;

void getgps(TinyGPS &gps)
{
    byte month, day, hour, minute, second, hundredths;
1   gps.crack_datetime(&year,&month,&day,&hour,&minute,&second,&h
undredths);
2   hour=hour+10; // change the offset so it is correct for your
time zone
    if (hour>23)
    {
        hour=hour-24;
    }
    lcd.setCursor(0,0); // print the date and time
3   lcd.print("Current time: ");
    lcd.setCursor(4,1);
    if (hour<10)
    {
        lcd.print("0");
    }
    lcd.print(hour, DEC);
    lcd.print(":");
    if (minute<10)
    {
        lcd.print("0");
    }
    lcd.print(minute, DEC);
    lcd.print(":");
    if (second<10)
    {
        lcd.print("0");
    }
    lcd.print(second, DEC);

```

```

}

void setup()
{
  Serial2.begin(9600);
}

void loop()
{
  while (Serial2.available() > 0)
  {
    // get the byte data from the GPS
    gpsData = Serial2.read();
    if (gps.encode(gpsData))
    {
      getgps(gps);
    }
  }
}

```

This example works in a similar way to the sketch in Project 43, except that instead of extracting the position data, it extracts the time (always at Greenwich Mean Time, more commonly known as UTC) at 1. At 2, you can either add or subtract a number of hours to bring the clock into line with your current time zone. The time should then be formatted clearly and displayed on the LCD at 3. [Figure 15-7](#) shows an example of the clock.



Figure 15-7: Project 44 at work

Project #45: Recording the Position of a Moving Object over Time

Now that we know how to receive GPS coordinates and convert them into normal variables, we can use this information with a microSD or SD card, introduced in Chapter 7, to build a GPS logger. Our logger will record our position over time by logging the GPS data over time. The addition of the memory card will allow you to record the movement of a car, truck, boat, or any other moving object that allows GPS signal reception; later, you can review the information on a computer.

The Hardware

If you have a GPS shield for Arduino, as recommended earlier in this chapter, the required hardware is the same as that used for the previous examples, except that you can remove the LCD shield. If you're using a GPS receiver module, you will need the screw shield to allow connection of the GPS and the SD card module. No matter which method you use, you will need external power for this project. In our example, we'll record the time, position information, and estimated speed of travel.

The Sketch

After assembling your hardware, enter and upload the following sketch:

```
// Project 45 - Recording the Position of a Moving Object
over Time

#include <TinyGPS.h>
#include <SoftwareSerial.h>
#include <SD.h>

// GPS TX to D2, RX to D3

SoftwareSerial Serial2(2, 3);

TinyGPS gps;
void getgps(TinyGPS &gps);

byte gpsData;

void getgps(TinyGPS &gps)
{
    float latitude, longitude;
    int year;
```

```

byte month, day, hour, minute, second, hundredths;

// create/open the file for writing
File dataFile = SD.open("DATA.TXT", FILE_WRITE);
// if the file is ready, write to it:
1  if (dataFile)
    {
2      gps.f_get_position(&latitude, &longitude);
      dataFile.print("Lat: ");
      dataFile.print(latitude, 5);
      dataFile.print(" ");
      dataFile.print("Long: ");
      dataFile.print(longitude, 5);
      dataFile.print(" ");
      // decode and display time data
      gps.crack_datetime(&year, &month, &day, &hour, &minute,
&second,
                          &hundredths);
      // correct for your time zone as in Project 44
      hour = hour + 10;
      if (hour > 23)
      {
          hour = hour - 24;
      }
      if (hour < 10)
      {
          dataFile.print("0");
      }
      dataFile.print(hour, DEC);
      dataFile.print(":");
      if (minute < 10)
      {
          dataFile.print("0");
      }
      dataFile.print(minute, DEC);
      dataFile.print(":");
      if (second < 10)
      {
          dataFile.print("0");
      }
      dataFile.print(second, DEC);
      dataFile.print(" ");
      dataFile.print(gps.f_speed_kmph());
3      dataFile.println("km/h");
      dataFile.close();
4      delay(15000); // record a measurement around every 15

```



```

seconds
}
// if the file isn't ready, show an error:
else
{
    Serial.println("error opening DATA.TXT");
}
}

void setup()
{
    Serial.begin(9600);
    Serial2.begin(9600);
    Serial.println("Initializing SD card...");
    pinMode(10, OUTPUT);
    // check that the memory card exists and is usable
    if (!SD.begin(10))
    {
        Serial.println("Card failed, or not present");
        // stop sketch
        return;
    }
    Serial.println("memory card is ready");
}

void loop()
{
    while (Serial2.available() > 0)
    {
        // get the byte data from the GPS
        gpsData = Serial2.read();
        if (gps.encode(gpsData))
        {
5           getgps(gps);
        }
    }
}

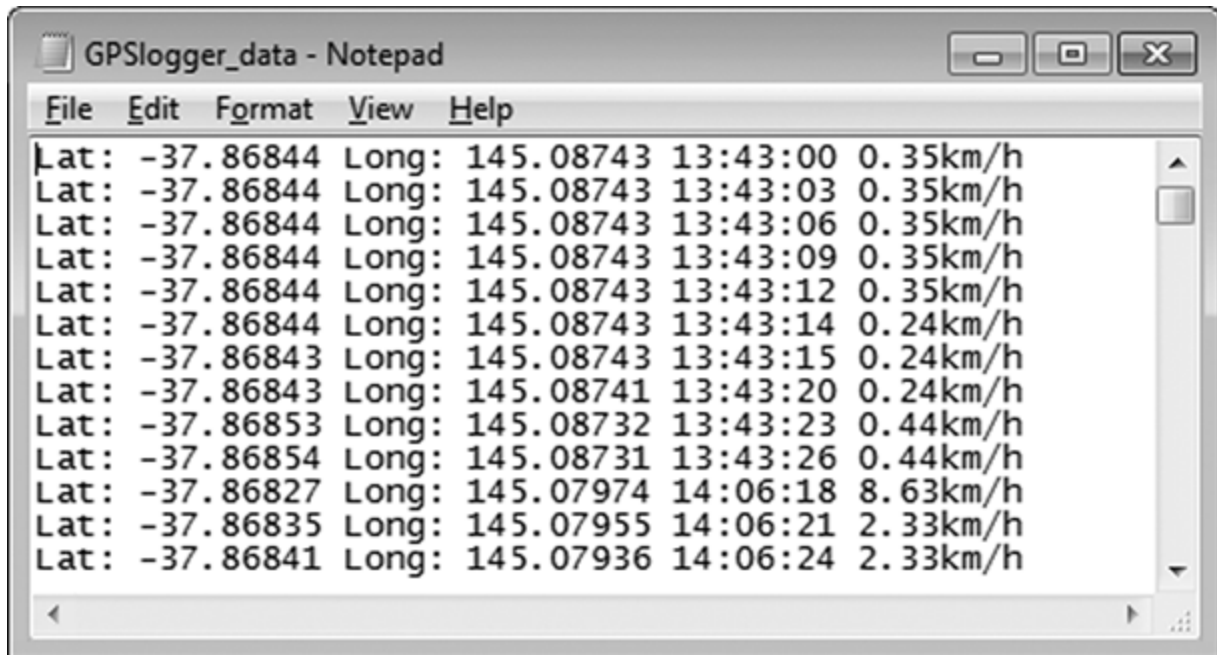
```

This sketch uses the same code used in Projects 43 and 44 in `void loop()` to receive data from the GPS receiver and pass it on to other functions. At 5, the data from the GPS receiver is passed into the TinyGPS library to decode it into useful variables. At 1, the memory card is checked to determine whether data can be written to it, and from 2 to 3, the relevant GPS data is written to the text file on the microSD card. Because the file is closed after every write, you can remove the power source from the

Arduino without warning the sketch, and you should do so before inserting or removing the microSD card. Finally, you can set the interval between data recordings at 4 by changing the value in the `delay()` function.

Running the Sketch

After operating your GPS logger, the resulting text file should look similar to [Figure 15-8](#).



[Figure 15-8](#): Results from Project 45

Once you have this data, you can enter it into Google Maps manually and review the path taken by the GPS logger, point by point. But a more interesting method is to display the entire route taken on one map. To do this, open the text file as a spreadsheet, separate the position data, and add a header row, as shown in [Figure 15-9](#). Then save it as a .csv file.

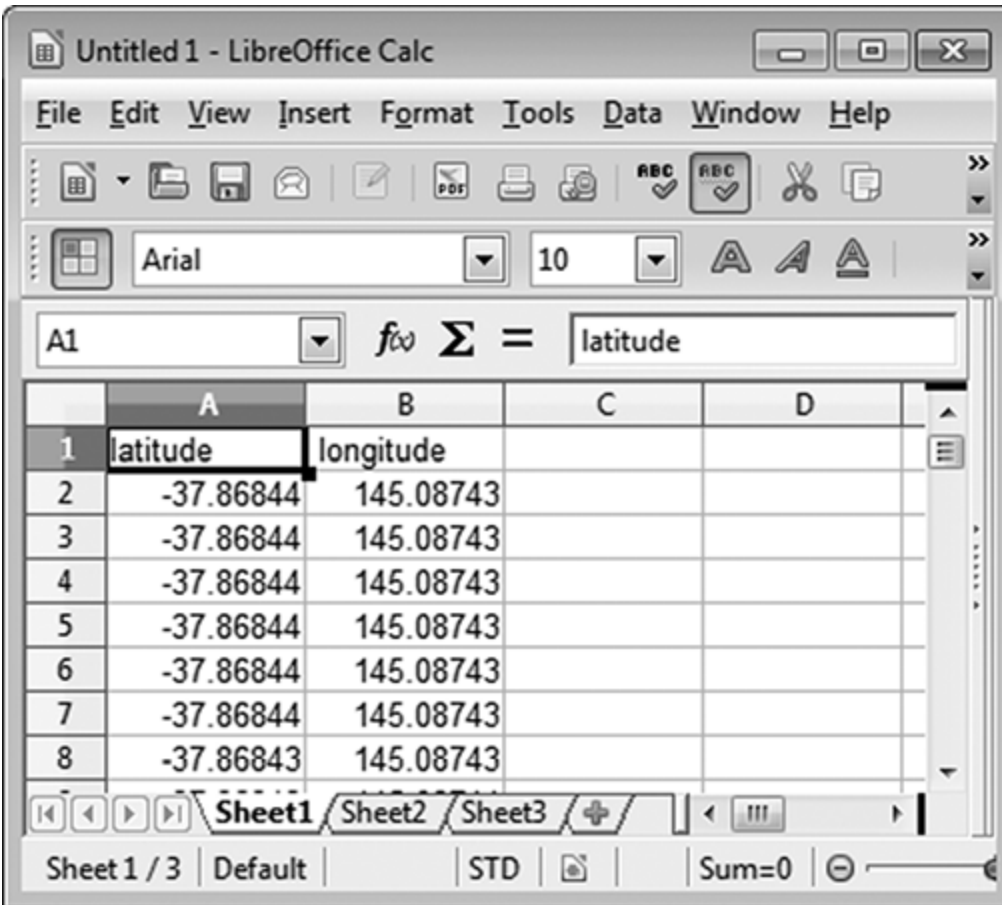


Figure 15-9: Captured position data

Now visit the GPS Visualizer website (<http://www.gpsvisualizer.com/>). In the Get Started Now box, click **Choose File** and select your data file. Choose **Google Maps** as the output format and then click **Map It**. The movement of your GPS logger should be shown on a map similar to the one in [Figure 15-10](#), which you can then adjust and explore.

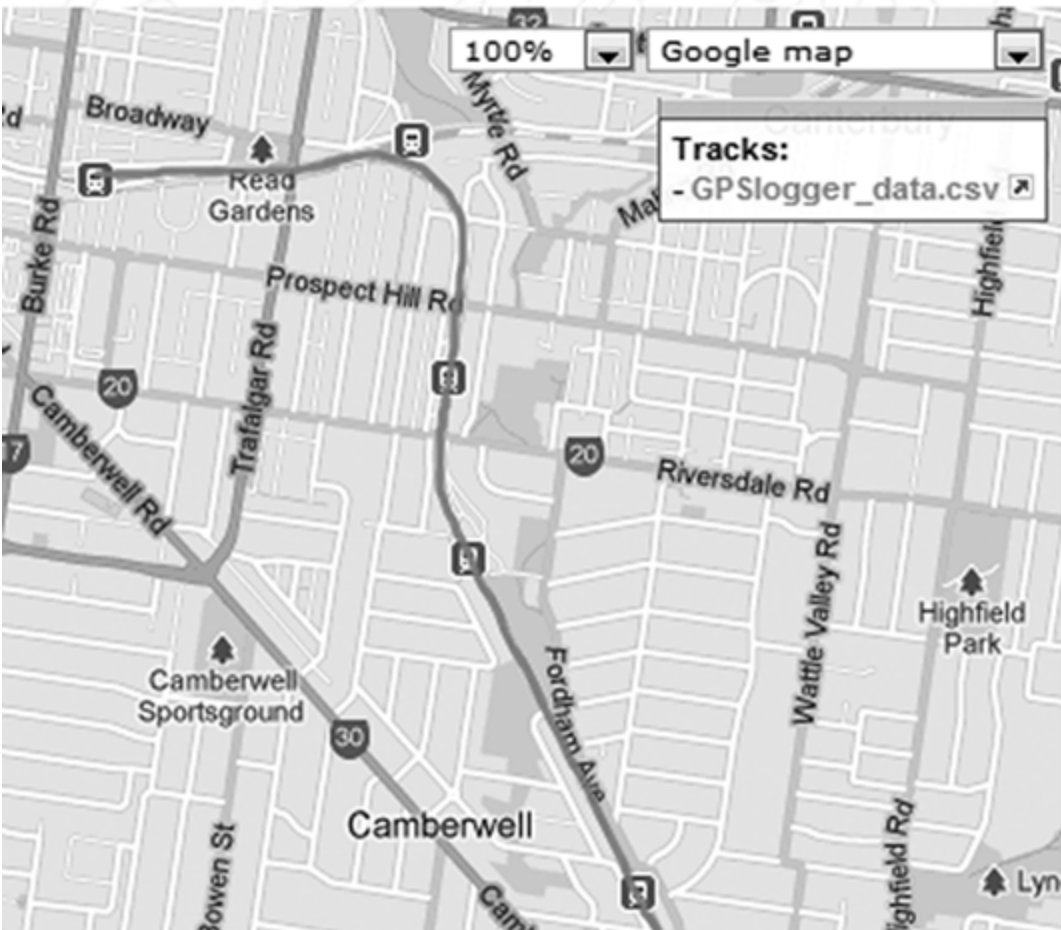


Figure 15-10: Mapped GPS logger data

Looking Ahead

As you can see, something that you might have thought too complex, such as working with GPS receivers, can be made simple with your Arduino. Continuing with that theme, in the next chapter you'll learn how to create your own wireless data links and direct things via remote control.

16

WIRELESS DATA

In this chapter, you'll learn how to send and receive instructions and data using various types of wireless transmission hardware. Specifically, you'll learn how to

Send digital output signals using low-cost wireless modules

Create a simple and inexpensive wireless remote control system

Use LoRa wireless data receivers and transceivers

Create a remote control temperature sensor

Using Low-Cost Wireless Modules

It's easy to send text information in one direction using a wireless link between two Arduino-controlled systems that have inexpensive radio frequency (RF) data modules, such as the transmitter and receiver modules shown in [Figure 16-1](#). These modules are usually sold in pairs and are known as *RF Link* modules or kits. Good examples are part 44910433 from PMD Way or parts WRL-10534 and WRL-10532 from SparkFun. We'll use the most common module types that run on the 433 MHz radio frequency in our projects.

The connections shown at the bottom of the transmitter in [Figure 16-2](#) are, from left to right, data in, 5 V, and GND. A connection for an external antenna is at the top-right corner of the board. The antenna can be a single

length of wire, or it can be omitted entirely for short transmission distances. (Each brand of module can vary slightly, so check the connections on your particular device before moving forward.)

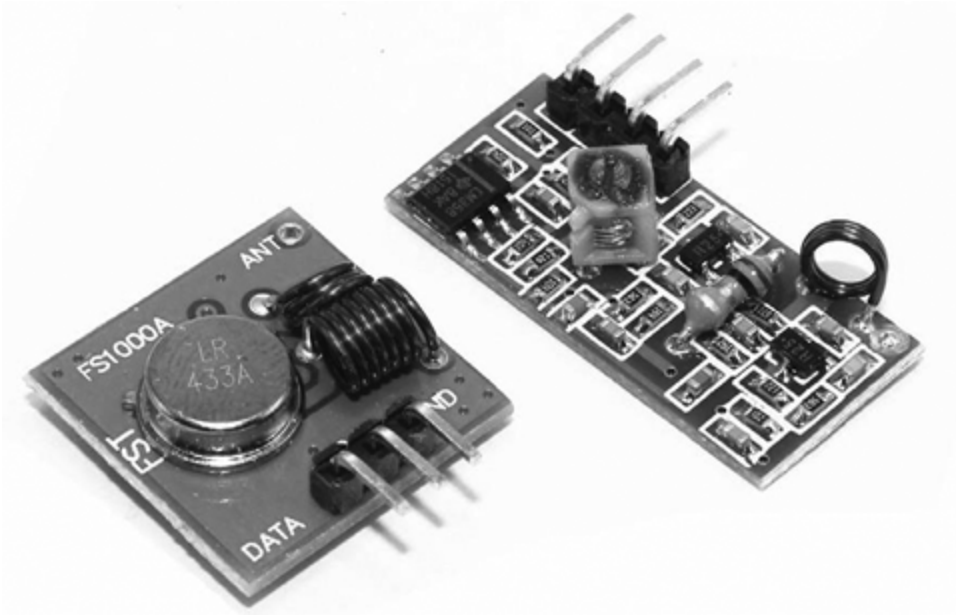


Figure 16-1: RF Link transmitter and receiver set

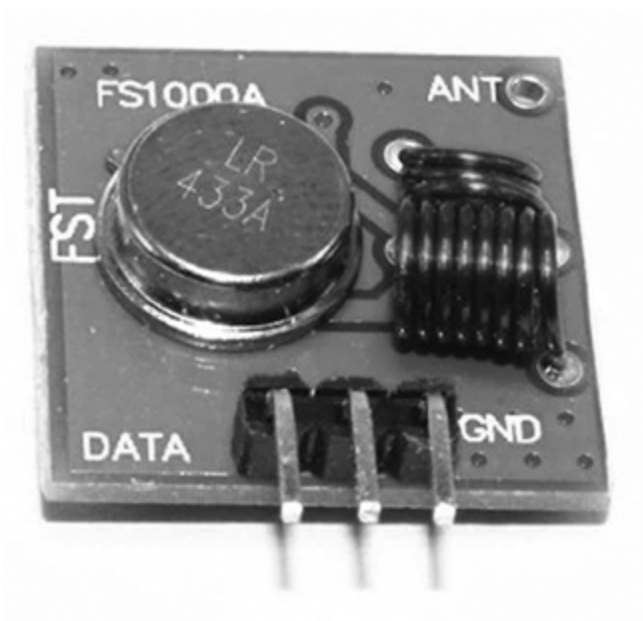


Figure 16-2: Transmitter RF Link module

[Figure 16-3](#) shows the receiver module, which is slightly larger than the transmitter module.

NOTE

The RF Link modules are inexpensive and easy to use, but they have no error-checking capability to ensure that the data being sent is received correctly. Therefore, I recommend that you use them only for simple tasks such as this basic remote control project. If your project calls for more accurate and reliable data transmission, use something like the LoRa modules instead, which are discussed later in this chapter.

Project #46: Creating a Wireless Remote Control

We'll remotely control two digital outputs: you'll press buttons connected to one Arduino board to control matching digital output pins on another Arduino located some distance away. This project will show you how to use the RF Link modules. You'll also learn how to determine how far away you can be and remotely control the Arduino. It's important to know this before you commit to using the modules for more complex tasks. (In open air, the distance you can achieve is generally about 100 meters, but the distance will be less when you are indoors or when the modules are separated by obstacles.)

The Transmitter Circuit Hardware

The following hardware is required for the transmitter circuit:

Arduino and USB cable

AA battery holder and wiring (as used in Chapter 14)

One 433 MHz RF Link transmitter module

Two 10 k Ω resistors (R1 and R2)

Two 100 nF capacitors (C1 and C2)

Two push buttons

One breadboard

The Transmitter Schematic

The transmitter circuit consists of two push buttons with debounce circuitry connected to digital pins 2 and 3, as well as the transmitter module wired as described earlier ([Figure 16-4](#)).

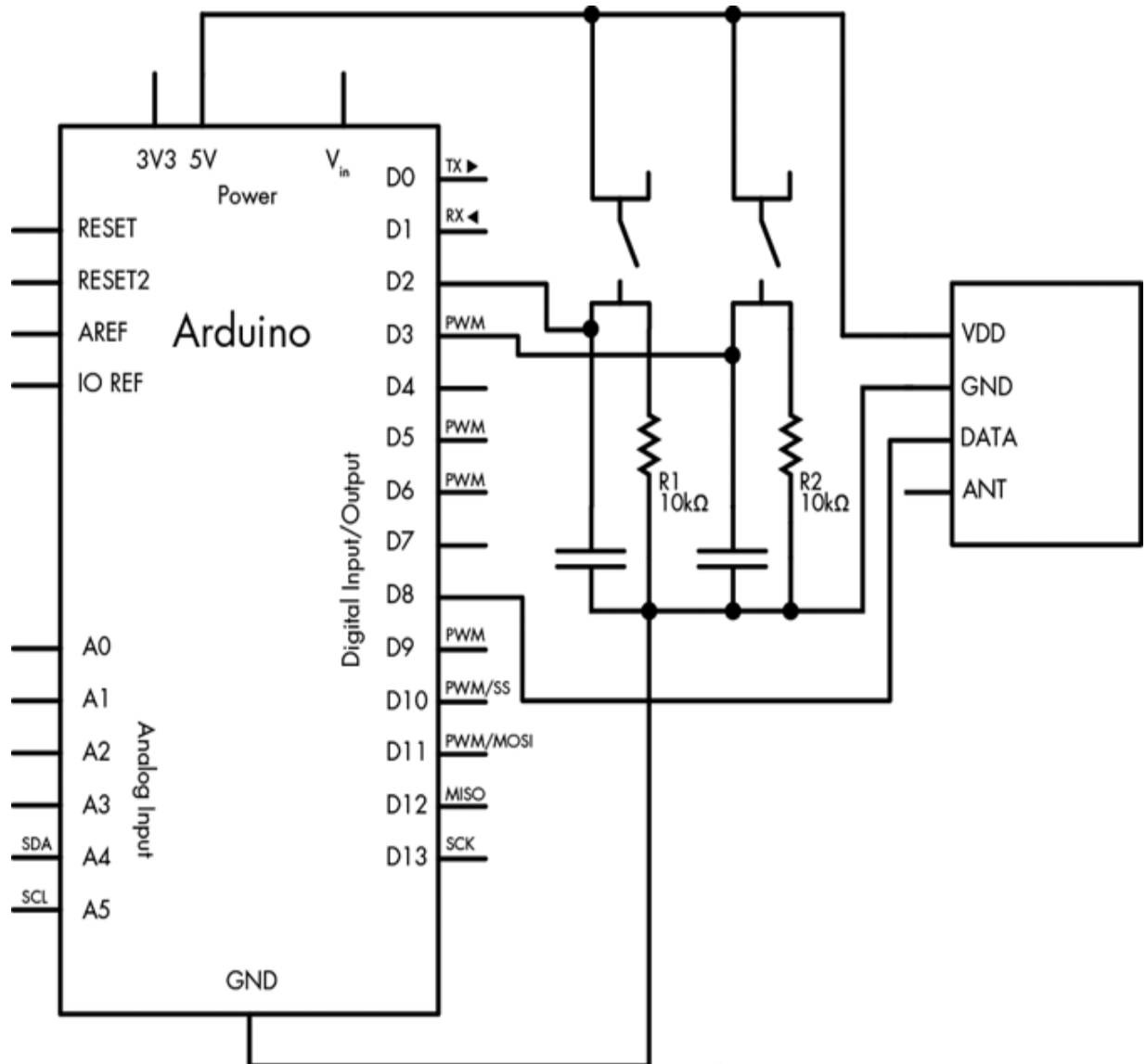


Figure 16-4: Transmitter schematic for Project 46

The Receiver Circuit Hardware

The following hardware is required for the receiver circuit:

Arduino and USB cable

AA battery holder and wiring (as used in Chapter 14)

One 433 MHz RF Link receiver module

One breadboard

Two LEDs of your choice

Two 560 Ω resistors (R1 and R2)

The Receiver Schematic

The receiver circuit consists of two LEDs on digital pins 6 and 7 and the data pin from the RF Link receiver module connected to digital pin 8, as shown in [*Figure 16-5*](#).

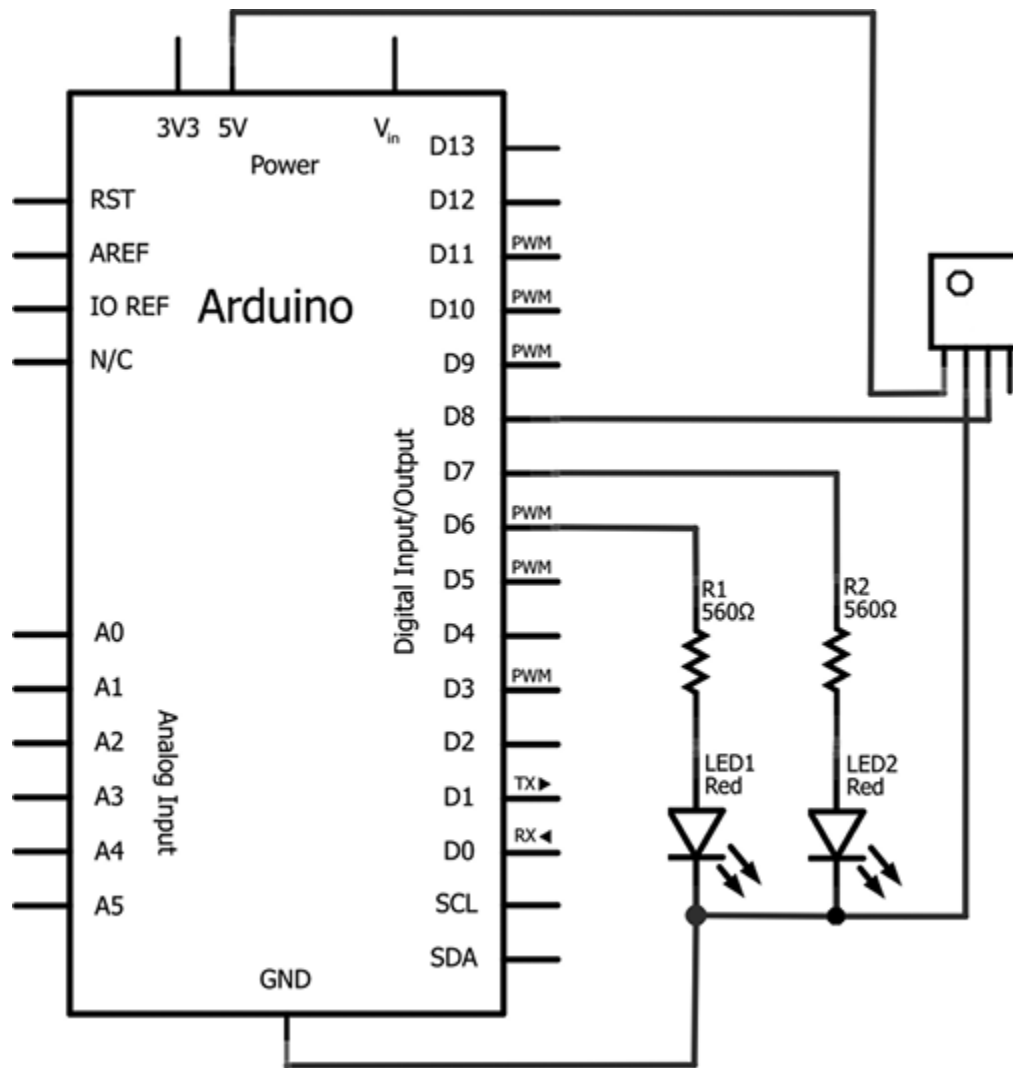


Figure 16-5: Receiver schematic for Project 46

You can substitute the breadboard, LEDs, resistors, and receiver module with a Freetronics 433 MHz receiver shield, shown in [Figure 16-6](#).

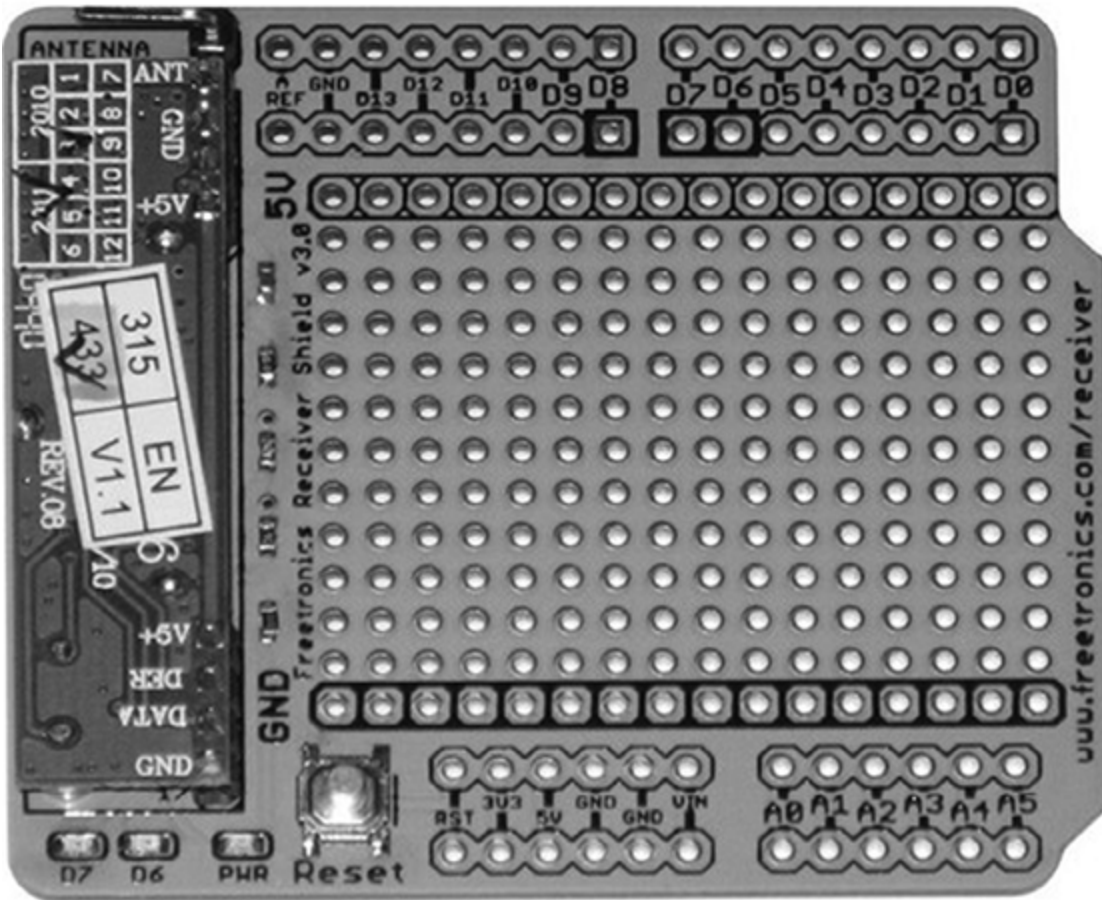


Figure 16-6: A Freetronics 433 MHz receiver shield

The Transmitter Sketch

Now let's examine the sketch for the transmitter. Enter and upload the following sketch to the Arduino with the transmitter circuit: // Project 46

```
- Creating a Wireless Remote Control, Transmitter Sketch
#include <VirtualWire.h> uint8_t buf[VW_MAX_MESSAGE_LEN];
uint8_t buflen = VW_MAX_MESSAGE_LEN; const char *on2 = "a";
const char *off2 = "b"; const char *on3 = "c"; const char *off3
= "d"; void setup() { vw_set_ptt_inverted(true); // Required for
RF Link modules vw_setup(300); // set data speed
vw_set_tx_pin(8); pinMode(2, INPUT); pinMode(3, INPUT); } void
loop() { if (digitalRead(2) == HIGH) { vw_send((uint8_t *)on2,
strlen(on2)); // send data out to the world vw_wait_tx(); //
wait a moment delay(200); } if (digitalRead(2) == LOW) {
```

```

vw_send((uint8_t *)off2, strlen(off2)); vw_wait_tx();
delay(200); } if (digitalRead(3)==HIGH) { vw_send((uint8_t
*)on3, strlen(on3)); vw_wait_tx(); delay(200); } if
(digitalRead(3)==LOW) { vw_send((uint8_t *)off3, strlen(off3));
vw_wait_tx(); delay(200); } }

```

We include the VirtualWire library at 1 and use its functions at 3 to set up the RF Link transmitter module and set the data speed. At 4, we set digital pin 8, which is used to connect the Arduino to the data pin of the transmitter module and to control the speed of the data transmission. (You can use any other digital pins if necessary, except 0 and 1, which would interfere with the serial line.) The transmitter sketch reads the status of the two buttons connected to digital pins 2 and 3 and sends a single text character to the RF Link module that matches the state of the buttons. For example, when the button on digital pin 2 is HIGH, the Arduino sends the character *a*, and when the button is LOW, it sends the character *b*. When the button on digital pin 3 is HIGH, the Arduino sends the character *c*, and when the button is LOW, it sends the character *d*. The four states are declared starting at 2.

The transmission of the text character is handled using one of the four sections' if statements, starting at 5—for example, the contents of the if-then statement at 6. The variable transmitted is used twice, as shown here with on2, for example: `vw_send((uint8_t *)on2, strlen(on2));`

The function `vw_send()` sends the contents of the variable on2, but it needs to know the length of the variable in characters, so we use `strlen()` to accomplish this.

The Receiver Sketch

Now let's add the receiver sketch. Enter and upload the following sketch to the Arduino with the receiver circuit: // Project 46 - Creating a

```

Wireless Remote Control, Receiver Sketch #include
<VirtualWire.h> uint8_t buf[VW_MAX_MESSAGE_LEN]; uint8_t buflen
= VW_MAX_MESSAGE_LEN; void setup()1{vw_set_ptt_inverted(true);
// Required for RF Link modules vw_setup(3002);vw_set_rx_pin(8);
vw_rx_start(); pinMode(6, OUTPUT); pinMode(7, OUTPUT); } void
loop()3{if (vw_get_message(buf, &buflen))4{switch(buf[0]) {
case 'a': digitalWrite(6, HIGH); break; case 'b':

```

```
digitalWrite(6, LOW); break; case 'c': digitalWrite(7, HIGH);  
break; case 'd': digitalWrite(7, LOW); break; } } }
```

As with the transmitter circuit, we use the VirtualWire functions at 1 to set up the RF Link receiver module and set the data speed. At 2 we set the Arduino digital pin to which the link's data output pin is connected (pin 8).

When the sketch is running, the characters sent from the transmitter circuit are received by the RF Link module and sent to the Arduino. The function `vw_get_message()` at 3 takes the characters received by the Arduino, which are interpreted by the switch case statement at 4. For example, pressing button S1 on the transmitter circuit will send the character *a*. This character is received by the transmitter, which sets digital pin 6 to HIGH, turning on the LED.

You can use this simple pair of demonstration circuits to create more complex controls for Arduino systems by sending codes as basic characters to be interpreted by a receiver circuit.

Using LoRa Wireless Data Modules for Greater Range and Faster Speed

When you need a wireless data link with greater range and a faster data speed than what the basic wireless modules used earlier can provide, LoRa data modules may be the right choice. LoRa is short for “long range,” and these modules work at long range with low power consumption. The modules are *transceivers*, which are devices that can both transmit and receive data, so you don't need a separate transmitter and receiver. A further benefit of using LoRa modules is that different types of modules can communicate, allowing you, the designer, to create control and data networks that range from the simple to the complex. In this chapter, you will create several basic modules that can be built upon for various purposes.

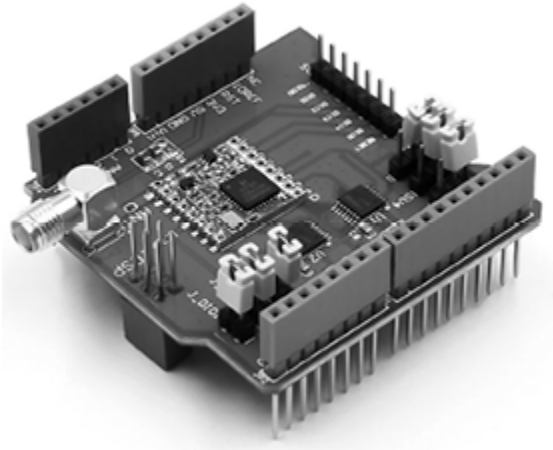


Figure 16-7: A LoRa shield for Arduino

For convenience, we'll be using two LoRa shields for Arduino, such as PMD Way part number 14290433, shown in [Figure 16-7](#).

When purchasing your LoRa shields, you will need to select an operating frequency. The correct frequency will vary depending on your country of use. This is to ensure that your data transmissions don't interfere with other devices in your area. LoRa products are available in three operating frequency bands:

433 MHz Used in United States and Canada

868 MHz Used in United Kingdom and Europe

915 MHz Used in Australia and New Zealand

You can find a full list of countries and the frequency ranges you need to use in each at <https://www.thethingsnetwork.org/docs/lorawan/frequencies-by-country.html>.

Finally, you need to download and install the Arduino library, which can be found at <https://github.com/sandeepmistry/arduino-LoRa/archive/master.zip>.

Project #47: Remote Control over LoRa Wireless

This project will demonstrate simple data transmission from one LoRA-equipped Arduino to another to allow remote control of a digital output pin. Our transmitter has two buttons to turn the receiver circuit's output pin on and off.

The Transmitter Circuit Hardware

The following hardware is required for the transmitter circuit:

Arduino and USB cable

LoRa shield for Arduino

Two 10 k Ω resistors (R1 and R2)

Two 100 nF capacitors (C1 and C2)

Two push buttons

AA battery holder and wiring (as used in Chapter 14)

The Transmitter Schematic

The transmitter circuit, as shown in [Figure 16-8](#), consists of two push buttons with debounce circuitry connected to digital pins 2 and 3. The LoRa shield is placed on the Arduino Uno. Once the sketch has been uploaded, power is provided by the AA battery holder and wiring.

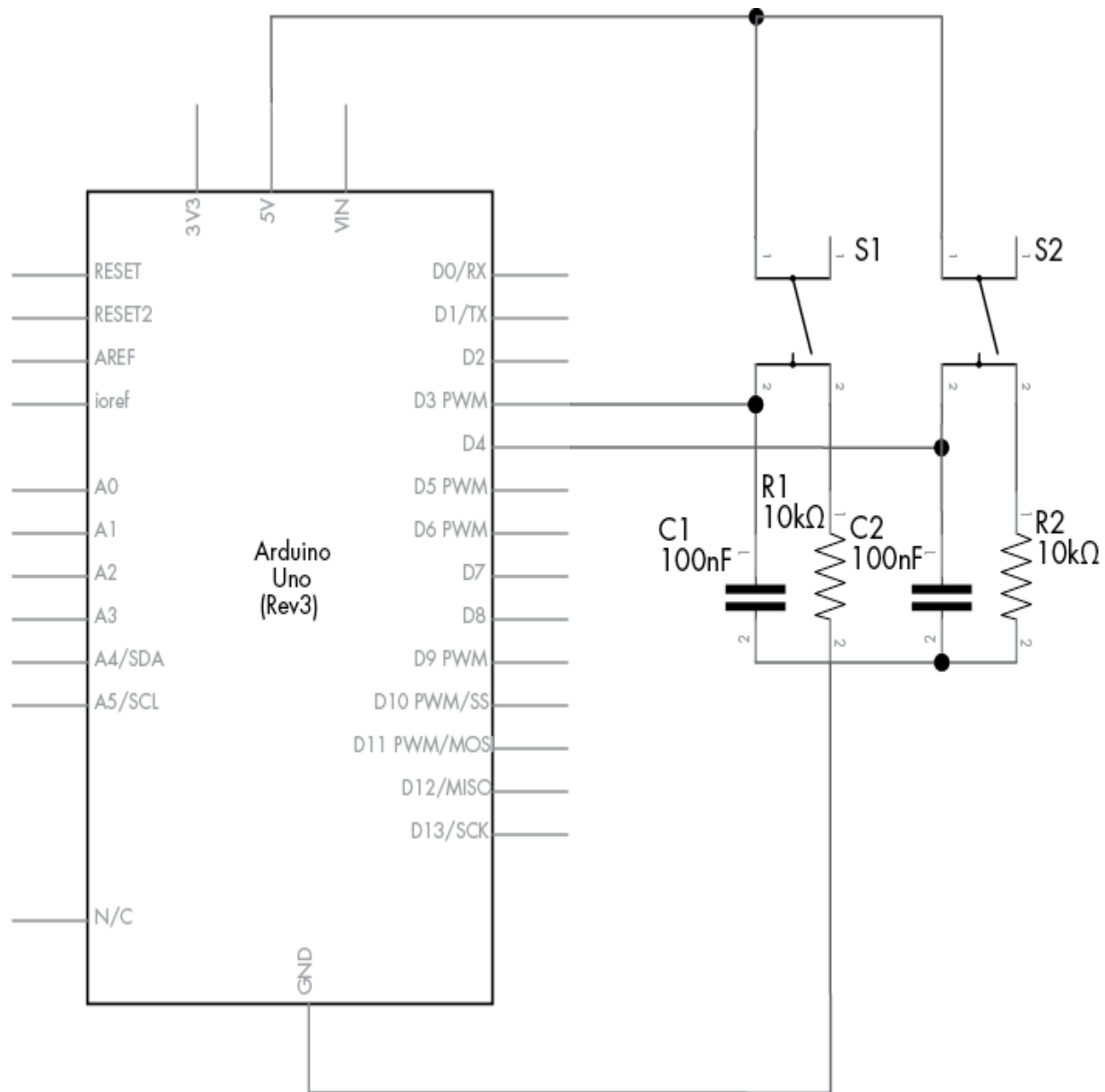


Figure 16-8: Transmitter schematic for Project 47

Before using your LoRa shield, there are three header jumpers, shown in [Figure 16-9](#), that need to be removed from the shield. If you don't remove them, they will interfere with other digital pins. You can remove them completely or just connect the headers to one of the two pins.

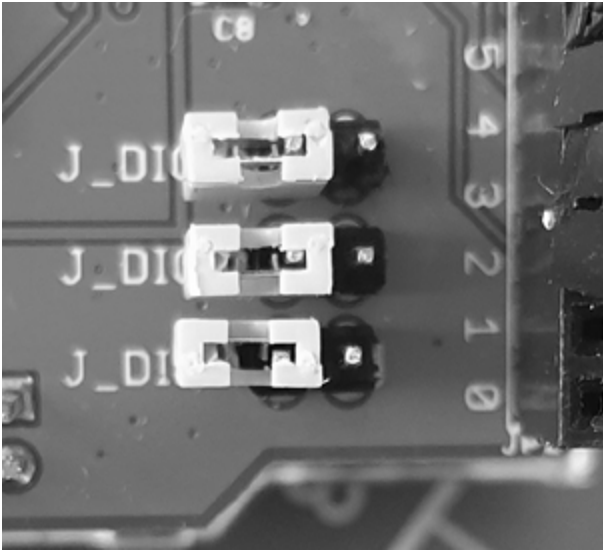


Figure 16-9: Header jumpers to remove from the LoRa shield

The Receiver Circuit Hardware

The following hardware is required for the receiver circuit:

Arduino and USB cable

LoRa shield for Arduino

One LED

One 560 Ω resistor (R1)

The Receiver Schematic

The receiver circuit, shown in [Figure 16-10](#), consists of one LED and a current-limiting resistor connected between digital pin 7 and GND. We leave this connected to the PC via USB, so no external power is required.

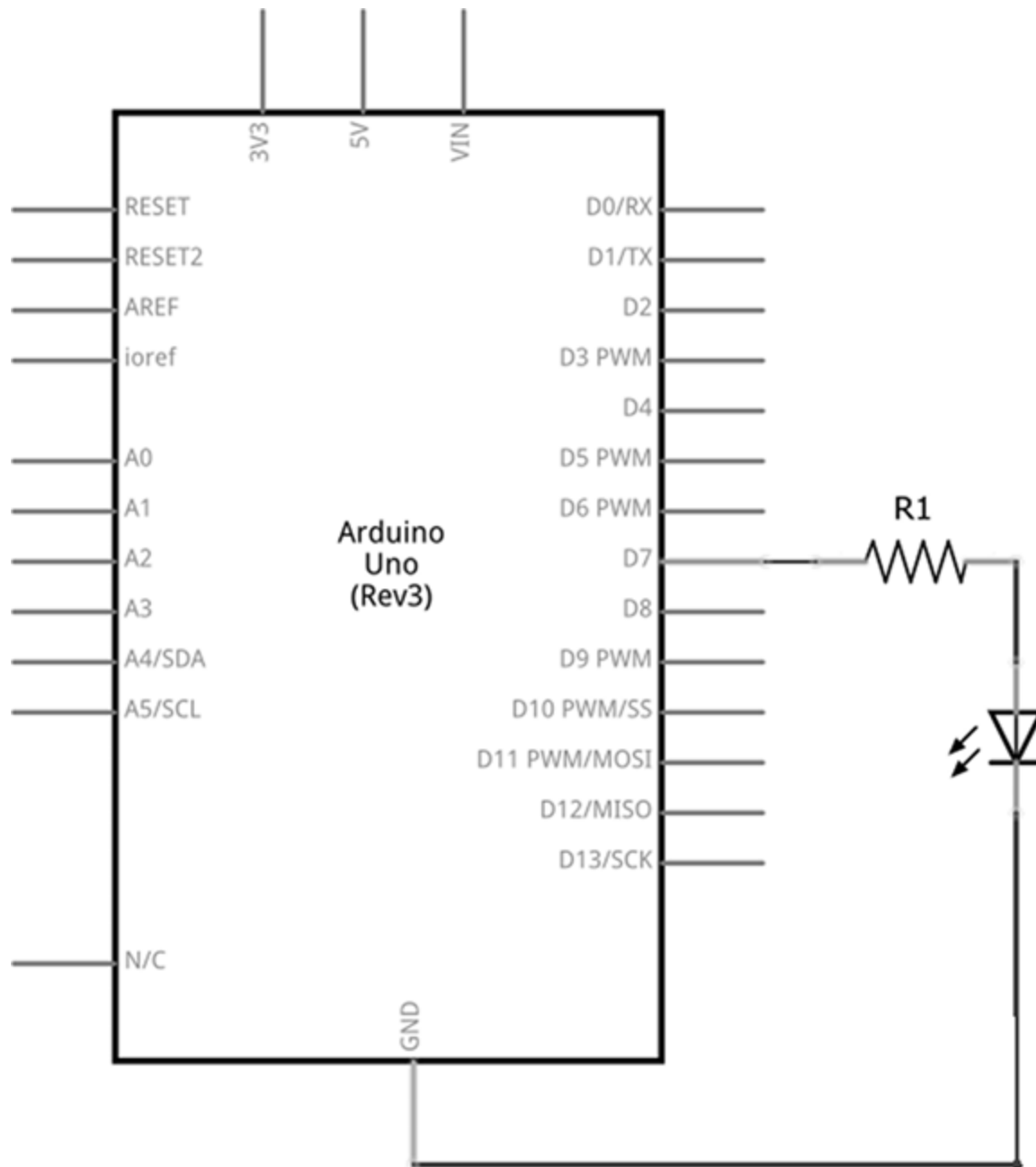


Figure 16-10: Receiver schematic for Project 47

The Transmitter Sketch

Now let's examine the sketch for the transmitter. Enter and upload the following sketch to the Arduino with the transmitter circuit: // Project 47 - Remote Control over LoRa Wireless, Transmitter Sketch

```
// Project 47
- Remote Control over LoRa Wireless, Transmitter Sketch
#define LORAFREQ (915000000L)
#include <LoRa.h>
#include <SPI.h>
void loraSend(int controlCode)
{
  LoRa.beginPacket(); // start sending
```

```

data LoRa.print("ABC"); // "ABC" is our three-character code
for receiver LoRa.print(controlCode); // send our instructions
(controlCode code5)LoRa.endPacket(); // finished sending data
LoRa.receive(); // start listening } void setup() { pinMode(4,
INPUT); // on button pinMode(3, INPUT); // off button
LoRa.begin(LORAFREQ); // start up LoRa at specified frequency }
void loop() { // check for button presses to control receiver
if (digitalRead(4) == HIGH) { loraSend(1); // '1' is code for
turn receiver digital pin 5 HIGH delay(500); // allow time to
send } if (digitalRead(3) == HIGH) { loraSend(0); // '0' is
code for turn receiver digital pin 5 LOW delay(500); // allow
time to send } }

```

The operating frequency is selected at 1. Our example is using 915 MHz, so you may need to change this to either 433000000L or 868000000L depending on your country and shield. We include the Arduino LoRa library at 2, and it's activated at 4. The SPI library is also included, as the LoRa shield uses the SPI bus to communicate with the Arduino. At 6, the LoRa transceiver is activated at the appropriate frequency, after the digital pins are prepared to be inputs for the buttons.

At 3, we have the custom function `loraSend(int controlCode)`. This is called when a button is pressed. It first sends a three-character code—in this case ABC—out on the LoRa airwaves, followed by a control code. The character code allows you to direct the control to a particular receiver circuit. Otherwise, if you're using two or more receivers, there would be confusion as to which one would be controlled by the transmitter. You will see that the receiver will act only if ABC is sent. The control codes in our example are 1 and 0 (to turn the receiver's digital output on or off, respectively).

At 4, the LoRa module is switched to transmit mode, and then the character and control codes are sent over the airwaves. At 5, the LoRa module is told to stop transmitting and gets switched back to receiving data. Once you have uploaded the transmitter sketch, the transmitter hardware can be disconnected from the computer and powered using the battery pack.

The Receiver Sketch

Now let's examine the receiver sketch. Enter and upload the following sketch to the Arduino with the receiver circuit: // Project 47 - Remote Control over LoRa Wireless, Receiver Sketch

```
1#define LORAFREQ
(915000000L)
2#include <LoRa.h> #include <SPI.h> void
takeAction(int packetSize) // things to do when data received
over LoRa wireless
3{char incoming[4] = ""; int k; for (int i =
0; i < packetSize; i++) { k = i; if (k > 6) { k = 6; // make
sure we don't write past end of string } incoming[k] =
(char)LoRa.read(4);} // check the three-character code sent from
transmitter is correct
4if (incoming[0] != 'A') { return; // if
not 'A', stop function and go back to void loop()
5}if
(incoming[1] != 'B') { return; // if not 'B', stop function and
go back to void loop()
6}if (incoming[2] != 'C') { return; // if
not 'C', stop function and go back to void loop() } // If made
it this far, correct code has been received from transmitter.
// Now to do something... if (incoming[3] == '1') {
digitalWrite(7, HIGH); } if (incoming[3] == '0') {
digitalWrite(7, LOW); } } void setup() { pinMode(7, OUTPUT);
LoRa.begin(LORAFREQ); // start up LoRa at specified frequency
LoRa.onReceive(takeAction); // call function "takeAction" when
data received
7LoRa.receive(); // start receiving } void loop() {
}
```

Once again we include the Arduino LoRa library at 2, and it's activated at 6. The operating frequency is also selected at 1. Our example is using 915 MHz, so you may need to change this to either 433000000L or 868000000L depending on your country and shield. The SPI library is also included, as the LoRa shield uses the SPI bus to communicate with the Arduino. At 7, we tell the sketch to run a certain function—in this case void takeAction()—when data is received over the airwaves. Then at 8, the LoRa module is switched to receive mode.

When operating, the receiver simply waits for data to be received by the LoRa module. At that point, the function takeAction() is called. This takes each character of data from the transmitter and places it into an array of characters called incoming[4] between 3 and 4. Next, the receiver checks each character of the code (in our case ABC) at 5 to ensure the transmission is for this particular receiver. Finally, if this is successful, the control

character is checked. If it's a 1, digital pin 7 is set to HIGH, and if it's a 0, digital pin 7 is set to LOW.

Now you have the basic framework for a longer-distance remote control. Furthermore, by assigning different character codes to multiple receivers, you can expand your system to control more than one receiver unit from one transmitter.

However, for serious applications, you may want confirmation that an instruction from the transmitter has been successfully completed by the receiver, so we'll add a confirmation function in the next project.

Project #48: Remote Control over LoRa Wireless with Confirmation

This project adds a confirmation system to the receiver-transmitter setup created in Project 47, creating a two-way data system. An LED on the transmitter circuit will turn on when the receiver output is set to HIGH and turn off when the receiver output is set to LOW.

The Transmitter Circuit Hardware

The following hardware is required for the transmitter circuit:

Arduino and USB cable

LoRa Shield for Arduino

Two 10 k Ω resistors (R1 and R2)

One 560 Ω resistor (R3)

One LED

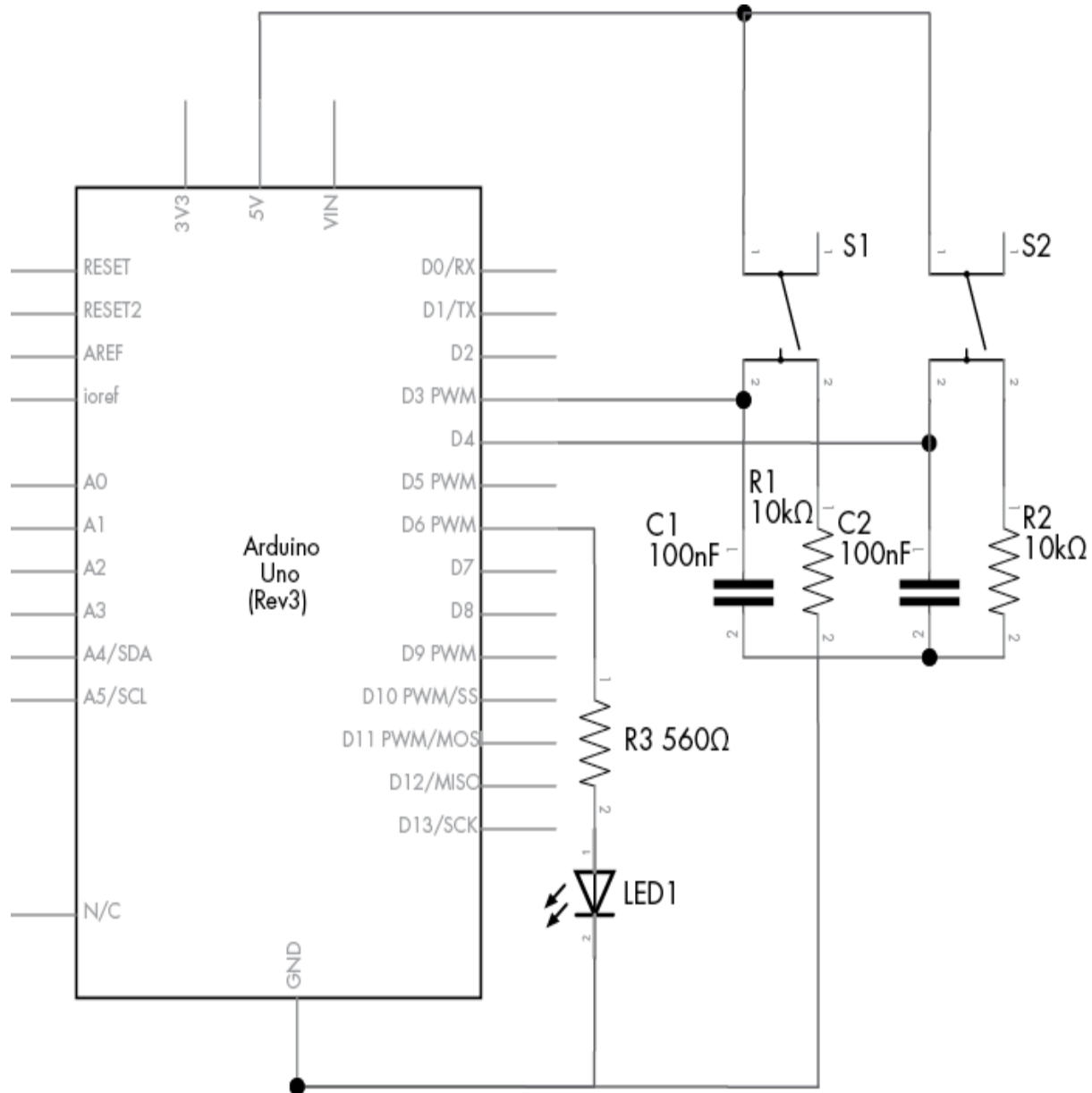
Two 100 nF capacitors (C1 and C2)

Two push buttons

AA battery holder and wiring (as used in Chapter 14)

The Transmitter Schematic

The transmitter circuit, shown in [Figure 16-11](#), consists of two push buttons with debounce circuitry connected to digital pins 3 and 4, and an LED and current-limiting resistor on digital pin 6. The LoRa shield is placed on the Arduino Uno. Once the sketch has been uploaded, power is provided by the AA battery holder and wiring.



[Figure 16-11](#): Transmitter schematic for Project 48

The receiver circuit and schematic for this project are identical to those used for Project 47.

The Transmitter Sketch

Now let's examine the sketch for the transmitter. Enter and upload the following sketch to the Arduino with the transmitter circuit: // Project 48

```
- Remote Control over LoRa Wireless with Confirmation, //
Transmitter Sketch #define LORAFREQ (915000000L) #include
<SPI.h> #include <LoRa.h> void loraSend(int controlCode) {
LoRa.beginPacket(); // start sending data LoRa.print("DEF"); //
"DEF" is our three-character code for the receiver. // Needs to
be matched on RX. LoRa.print(controlCode); // send our
instructions (controlCode codes) LoRa.endPacket(); // finished
sending data LoRa.receive(); // start listening1}void
takeAction(int packetSize) // things to do when data received
over LoRa wireless { char incoming[4] = ""; int k; for (int i =
0; i < packetSize; i++) { k = i; if (k > 6) { k = 18; // make
sure we don't write past end of string } incoming[k] =
(char)LoRa.read(); } // check the three-character code sent
from receiver is correct if (incoming[0] != 'D') { return; //
if not 'D', stop function and go back to void loop() } if
(incoming[1] != 'E') { return; // if not 'E', stop function and
go back to void loop() } if (incoming[2] != 'F') { return; //
if not 'F', stop function and go back to void loop() } // If
made it this far, correct code has been received from receiver.
// Now to do something.2.if (incoming[3] == '1') {
digitalWrite(6, HIGH); // receiver has turned output on and has
sent a signal confirming this2}if (incoming[3] == '0') {
digitalWrite(6, LOW); // receiver has turned output off and has
sent a signal confirming this } } void setup() { pinMode(4,
INPUT); // on button pinMode(3, INPUT); // off button
pinMode(6, OUTPUT); // status LED LoRa.begin(LORAFREQ); //
start up LoRa at specified frequency
LoRa.onReceive(takeAction); // call function "takeAction" when
data received // over LoRa wireless } void loop() { // check
for button presses to control receiver if (digitalRead(4) ==
HIGH) { loraSend(1); // '1' is code for turn receiver digital
pin 7 HIGH delay(500); // button debounce } if (digitalRead(3)
```



```
== HIGH) { loraSend(0); // '0' is code for turn receiver  
digital pin 7 LOW delay(500); // button debounce } }
```

Our transmitter circuit operates in the same way as in Project 47, by first sending a character code for identification and then a control code to turn the receiver's output on or off. In this project, however, the transmitter listens for a signal from the receiver, and once the receiver has completed the control instruction from the transmitter, the receiver sends a character code and control code back to the transmitter.

So at 1, we have a new function, `takeAction()`, that checks for the character code `DEF` from the receiver circuit. The receiver then sends a 1 if it has turned on its output pin or a 0 if the output has been turned off. Our transmitter circuit can then display this status by controlling the LED on digital pin 6 via the code at 2.

The Receiver Sketch

Finally, let's examine the sketch for the receiver. Enter and upload the following sketch to the Arduino with the receiver circuit: // Project 48 - Remote Control over LoRa Wireless with Confirmation, Receiver

```
// Sketch #define LORAFREQ (915000000L) #include <SPI.h>  
#include <LoRa.h> void loraSend(int controlCode) {  
LoRa.beginPacket(); // start sending data LoRa.print("DEF"); //  
"DEF" is our three-character code for the // transmitter  
LoRa.print(controlCode); // send our instructions (controlCode  
codes) LoRa.endPacket(); // finished sending data  
LoRa.receive(); // start listening } void takeAction(int  
packetSize) // things to do when data received over LoRa  
wireless { char incoming[4] = ""; int k; for (int i = 0; i <  
packetSize; i++) { k = i; if (k > 6) { k = 18; // make sure we  
don't write past end of string } incoming[k] =  
(char)LoRa.read(); } // check the three-character code sent  
from transmitter is correct if (incoming[0] != 'A') { return;  
// if not 'A', stop function and go back to void loop() } if  
(incoming[1] != 'B') { return; // if not 'B', stop function and  
go back to void loop() } if (incoming[2] != 'C') { return; //  
if not 'C', stop function and go back to void loop() } // If
```

made it this far, correct code has been received from transmitter. // Now to do something... if (incoming[3] == '1') { digitalWrite(7, HIGH); loraSend(1); // tell the transmitter that the output has been turned on } if (incoming[3] == '0') { digitalWrite(7, LOW); loraSend(0); // tell the transmitter that the output has been turned off } } void setup() { pinMode(7, OUTPUT); LoRa.begin(LORAFREQ); // start up LoRa at specified frequency LoRa.onReceive(takeAction); // call function "takeAction" when data received // over LoRa wireless LoRa.receive(); // start receiving } void loop() { }

Our receiver operates in the same manner as the one for Project 47, except in this case, the receiver sends back the character code DEF to the transmitter, followed by a 1 or a 0 to indicate that the output pin has been turned on or off. This is done at 1 using the `loraSend()` function.

At this point, you have two example projects that show how you can not only control digital output pins wirelessly across a greater distance than with the earlier projects but also confirm that the actions have taken place. You can now expand on these examples to create your own remote-control projects. But next, we'll experiment with sending sensor data over a LoRa wireless link with Project 49.

Project #49: Sending Remote Sensor Data Using LoRa Wireless

This project builds on our previous efforts by using your computer to request temperature data from a remote sensor.

The Transmitter Circuit Hardware

The following hardware is required for the transmitter circuit:

Arduino and USB cable

LoRa shield for Arduino

This project uses the Serial Monitor on your PC for control, so the transmitter circuit is simply the Arduino and LoRa shield connected to the

PC via the USB cable.

The Receiver Circuit Hardware

The following hardware is required for the receiver circuit:

Arduino and USB cable

LoRa shield for Arduino

TMP36 temperature sensor

Solderless breadboard

External power for Arduino

Male-to-male jumper wires

The Receiver Schematic

Our circuit is simply the TMP36 temperature sensor connected to analog pin A0, along with the LoRa shield placed on the Arduino, as shown in [Figure 16-12](#).

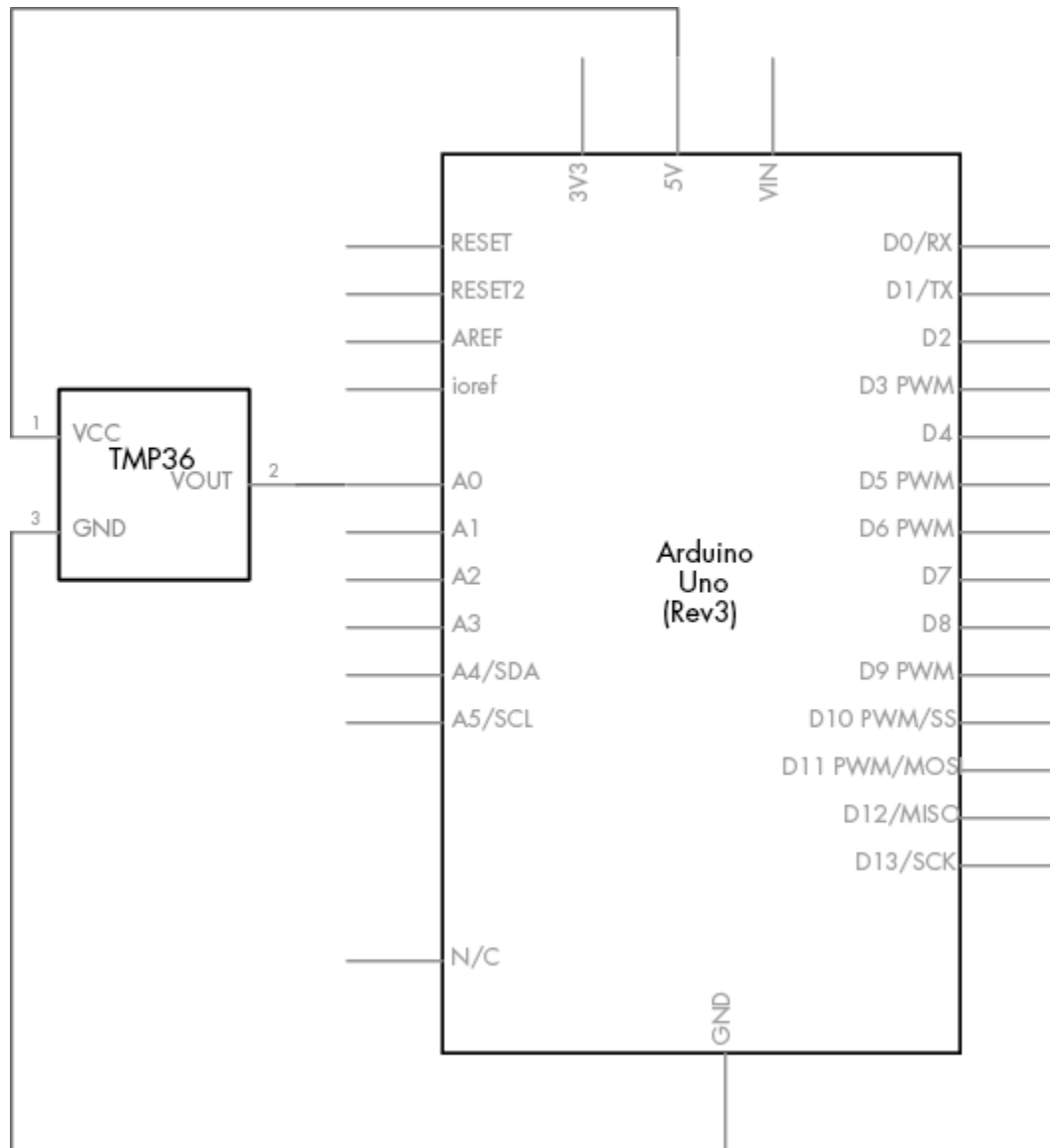


Figure 16-12: Receiver schematic for Project 49

The receiver circuit may be some distance from the computer, so you can harness a USB power supply or the battery solution used in earlier projects.

The Transmitter Sketch

Now let's examine the sketch for the transmitter. Enter and upload the following sketch to the Arduino with the transmitter circuit: // Project 49 - Sending Remote Sensor Data Using LoRa Wireless, Transmitter

```
// Sketch
#define LORAFREQ (915000000L)
#include <SPI.h>
#include <LoRa.h>
char command;
void loraSend(int controlCode)
```

```

{ LoRa.beginPacket(); // start sending data
LoRa.print("ABC");
// "ABC" is our three-character code for the // transmitter
LoRa.print(controlCode); // send our instructions (controlCode
codes) LoRa.endPacket(); // finished sending data
LoRa.receive(); // start listening } void takeAction(int
packetSize) // send text received from sensor Arduino via LoRa
to Serial Monitor { char incoming[31] = ""; int k; for (int i =
0; i < packetSize; i++) { k = i; if (k > 31) { k = 31; // make
sure we don't write past end of string } incoming[k] =
(char)LoRa.read(); Serial.print(incoming[k]); // display temp
information from sensor board } Serial.println(); } void
setup()2{LoRa.begin(LORAFREQ); // start up LoRa at specified
frequency LoRa.onReceive(takeAction); // call function
"takeAction" when data received // over LoRa wireless
LoRa.receive(); // start receiving Serial.begin(9600); } void
loop()3{Serial.print("Enter 1 for Celsius or 2 for Fahrenheit
then Enter: "); Serial.flush(); // clear any "junk" out of the
serial buffer before waiting4while (Serial.available() == 0) {
// do nothing until something enters the serial buffer } while
(Serial.available() > 0) { command = Serial.read() - '0'; //
read the number in the serial buffer, // remove the ASCII text
offset for zero: '0' } Serial.println(5;loraSend(command);
delay(2000); }

```

As with the earlier projects in this chapter, we initialize the LoRa hardware and the Serial Monitor at 2. However, instead of hardware buttons, we use the Serial Monitor to accept commands from the user and send those to the receiver hardware. In this project, the user is prompted to enter 1 or 2 in the Serial Monitor's input box to retrieve the temperature from the receiver hardware in Celsius or Fahrenheit, respectively. This happens at 3. The computer waits for user input at 4, then sends out either command to the receiver hardware via `loraSend()` at 5. Again, we use a three-character code to keep the transmission exclusively for the receiver board at 1.

The Receiver Sketch

Now let's examine the sketch for the receiver. Enter and upload the following sketch to the Arduino with the receiver circuit: // Project 49 -

```

Sending Remote Sensor Data Using LoRa Wireless, Receiver //
Sketch #define LORAFREQ (915000000L) #include <SPI.h> #include
<LoRa.h> float sensor = 0; float voltage = 0; float celsius =
0; float fahrenheit = 0; void loraSendC() { LoRa.beginPacket();
// start sending data sensor = analogRead(0); voltage =
((sensor * 5000) / 1024); voltage = voltage - 500; celsius =
voltage / 10; fahrenheit = ((celsius * 1.8) + 32);
LoRa.print("Temperature: "); LoRa.print(celsius, 2);
LoRa.print(" degrees C");LoRa.endPacket(); // finished sending
data LoRa.receive(); // start listening } void loraSendF() //
send temperature in Fahrenheit { LoRa.beginPacket(); // start
sending data sensor = analogRead(0); voltage = ((sensor * 5000)
/ 1024); voltage = voltage - 500; celsius = voltage / 10;
fahrenheit = ((celsius * 1.8) + 32);LoRa.print("Temperature:
"); LoRa.print(fahrenheit, 2); LoRa.print(" degrees F");
LoRa.endPacket(); // finished sending data LoRa.receive(); //
start listening } void takeAction(int packetSize) // things to
do when data received over LoRa wireless { char incoming[6] =
""; int k; for (int i = 0; i < packetSize; i++) { k = i; if (k
> 6) { k = 6; // make sure we don't write past end of string }
incoming[k] = (char)LoRa.read();3} // check the three-character
code sent from transmitter is correct if (incoming[0] != 'A') {
return; // if not 'A', stop function and go back to void loop()
} if (incoming[1] != 'B') { return; // if not 'B', stop
function and go back to void loop() } if (incoming[2] != 'C') {
return; // if not 'C', stop function and go back to void loop()
} // If made it this far, correct code has been received from
transmitter if (incoming[3] == '1')4{loraSendC(); } if
(incoming[3] == '2')5{loraSendF(); } } void setup() {
LoRa.begin(LORAFREQ); // start up LoRa at specified frequency
LoRa.onReceive(takeAction); // call function "takeAction" when
data received LoRa.receive(); // start receiving } void loop()
{ }

```

Using the same method as in Project 48, our receiver hardware decodes the transmission from the transmitter to ensure the data is meant for it by checking the character code sent at 3. If this is correct, the receiver board

calls one of either `loraSendC()` or `loraSendF()` at 4 or 5, respectively. Those two functions calculate the temperature from the TMP36 sensor and, between 1 and 2, send a string of text back to the transmitter board containing the temperature and measurement type.

Once you have assembled the hardware for both circuits and uploaded both sketches, place the powered receiver circuit (with the sensor) where you'd like to measure temperature from your computer. Ensure the transmitter circuit is connected to the computer. Open the Serial Monitor in the IDE and follow the instructions to check the temperature. An example is shown in [Figure 16-13](#).

```
Enter 1 for Celsius or 2 for Fahrenheit then Enter:
Temperature: 19.34 degrees C
Enter 1 for Celsius or 2 for Fahrenheit then Enter:
Temperature: 65.93 degrees F
Enter 1 for Celsius or 2 for Fahrenheit then Enter:
Temperature: 18.85 degrees C
Enter 1 for Celsius or 2 for Fahrenheit then Enter:
Temperature: 19.34 degrees C
Enter 1 for Celsius or 2 for Fahrenheit then Enter:
Temperature: 66.80 degrees F
Enter 1 for Celsius or 2 for Fahrenheit then Enter:
```

Figure 16-13: Example output for Project 49

Looking Ahead

This chapter showed how simple it is to control multi-Arduino systems remotely. For example, you can control digital outputs by sending characters from one Arduino to another, and you can use LoRa wireless technology to create more complex, multi-Arduino control systems that include data return. With the knowledge you've gained so far, many creative options are available to you.

But there's still much more to investigate in terms of wireless data transmission, so keep reading and working along with the examples as you learn to use simple television remote controls with the Arduino in the next chapter.

17

INFRARED REMOTE CONTROL

In this chapter you will

Create and test a simple infrared receiver

Remotely control Arduino digital output pins

Add a remote control system to the robot vehicle we created in Chapter 14

As you'll see, with the addition of an inexpensive receiver module, your Arduino can receive the signals from an infrared remote and act upon them.

What Is Infrared?

Many people use infrared remote controls in a variety of daily actions, and most don't know how they work. Infrared (IR) signals are beams of light at a wavelength that cannot be seen by the naked eye. So when you look at the little LED poking out of a remote control and press a button, you won't see the LED light up.

That's because IR remote controls contain one or more special infrared light-generating LEDs that transmit the IR signals. When you press a button on the remote, the LED turns on and off repeatedly in a pattern that is unique for each button pressed. This signal is received by a special IR receiver on the device being controlled and converted to pulses of electrical current that are read as data by the receiver's electronics. If you are curious about these patterns, you can view them by looking at the IR LED on a remote through the viewfinder of a phone camera or digital camera.

Setting Up for Infrared

Before moving forward, we need to install the Arduino IRremote library, so visit <https://github.com/z3t0/Arduino-IRremote/archive/master.zip> to download the required files and install them using the method described in Chapter 7.

The IR Receiver

The next step is to set up the IR receiver and test that it is working. You can choose either an independent IR receiver (shown in [Figure 17-1](#)) or a prewired module (shown in [Figure 17-2](#)), whichever is easier for you.

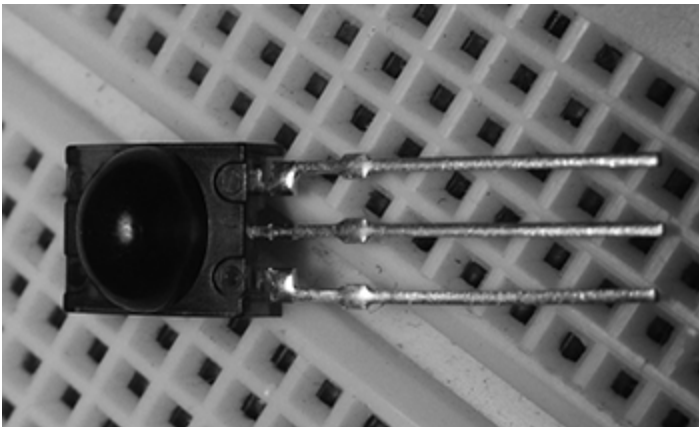


Figure 17-1: An IR receiver



Figure 17-2: A prewired IR receiver module

The independent IR receiver shown in [Figure 17-1](#) is a Vishay TSOP4138. The bottom leg of the receiver (as shown in the figure) connects to an Arduino digital pin, the center leg to GND, and the top leg to 5 V.

[Figure 17-2](#) shows a prewired IR module. Prewired receiver modules are available from PMD Way and other retailers. The benefit of using these modules is that they include connection wires and are labeled for easy reference.

Regardless of your choice of module, in all of the following examples, you'll connect D (the data line) to Arduino digital pin 2, VCC to 5 V, and GND to GND.

The Remote Control

Finally, you will need a remote control. I've used a surplus Sony TV remote like the one shown in [Figure 17-3](#). If you don't have access to a Sony remote, any inexpensive universal remote control can be used after you reset it to Sony codes. See the instructions included with your remote control to do this.



Figure 17-3: A typical Sony remote control

A Test Sketch

Now let's make sure that everything works. After connecting your IR receiver to the Arduino, enter and upload the sketch in [Listing 17-1](#).

// Listing 17-1

```
1 #include <IRremote.h>          // use the library

2 IRrecv irrecv(receiverpin); // create instance of irrecv
3 decode_results results;
  int receiverpin = 2;           // pin 1 of IR receiver to
  Arduino digital pin 2

  void setup()
  {
    Serial.begin(9600);
    irrecv.enableIRIn();        // start the IR receiver
  }

  void loop()
  {
4   if (irrecv.decode(&results))    // have we received an
    IR signal?
    {
5     Serial.print(results.value, HEX); // display IR code in
    the Serial Monitor
      Serial.print(" ");
      irrecv.resume();              // receive the next
    value
    }
  }
```

Listing 17-1: IR receiver test

This sketch is relatively simple, because most of the work is done in the background by the IR library. At 4, we check whether a signal has been received from the remote control. If so, it is displayed in the Serial Monitor in hexadecimal at 5. The lines at 1, 2, and 3 activate the IR library and create an instance of the infrared library function to refer to in the rest of the sketch.

Testing the Setup

Once you've uploaded the sketch, open the Serial Monitor, aim the remote at the receiver, and start pressing buttons. You should see codes for the

buttons displayed in the Serial Monitor after each button press. For example, [Figure 17-4](#) shows the results of pressing 1, 2, and 3, once each.

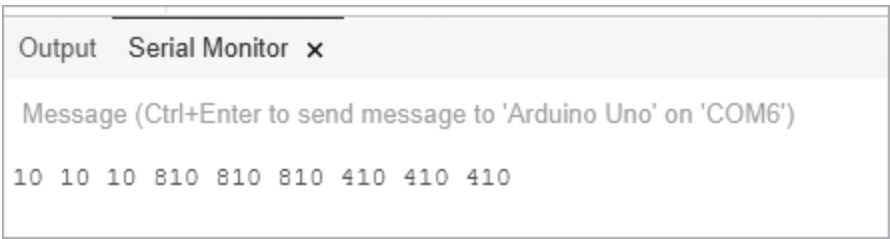


Figure 17-4: Results of pressing buttons after running the code in [Listing 17-1](#)

[Table 17-1](#) lists the codes from a basic Sony remote control that we’ll use in upcoming sketches. However, when running [Listing 17-1](#), notice that each code number repeats three times. This is an idiosyncrasy of Sony IR systems, which send the code three times for each button press. You can ignore these repeats with some clever coding, but for now, let’s skip to remote controlling with the next project.

Table 17-1: *Example Sony IR codes*

Button	Code	Button	Code
Power	A90	7	610
Mute	290	8	E10
1	10	9	110
2	810	0	910
3	410	Volume up	490
4	C10	Volume down	C90
5	210	Channel up	90
6	A10	Channel down	890

Project #50: Creating an IR Remote Control Arduino

This project will demonstrate how to control digital output pins using an IR remote control. You’ll control digital pins 3 through 7 with the matching numerical buttons 3 through 7 on a Sony remote control. When you press a button on the remote control, the matching digital output pin will change

state to HIGH for 1 second and then return to LOW. You'll be able to use this project as a base or guide to add remote control capabilities to your other projects.

The Hardware

The following hardware is required for this project:

Arduino and USB cable

Five LEDs

Five 560 Ω resistors

Infrared receiver or module

Solderless breadboard

Various jumper wires

The Schematic

The circuit consists of the infrared receiver with the output connected to digital pin 2 and five LEDs with current-limiting resistors connected to digital pins 3 to 7 inclusive, as shown in [*Figure 17-5*](#).

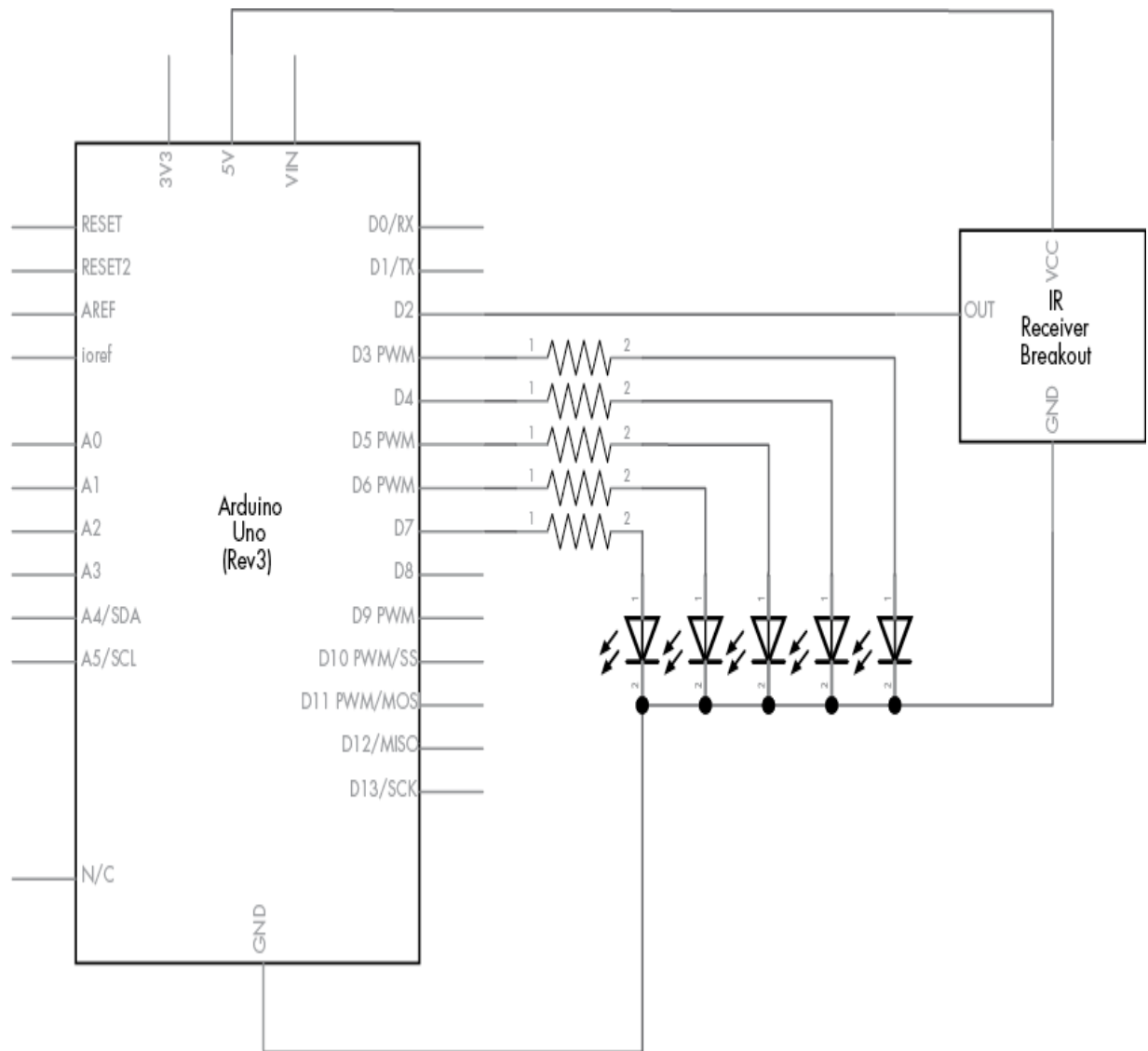


Figure 17-5: Schematic for Project 50

The Sketch

Enter and upload the following sketch:

```
// Project 50 - Creating an IR Remote Control Arduino

#include <IRremote.h>
IRrecv irrecv(receiverpin); // create instance of irrecv
decode_results results;
int receiverpin = 2;          // pin 1 of IR receiver to
Arduino digital pin 2
void setup()
```

```

{
  irrecv.enableIRIn();      // start the receiver
  for (int z = 3 ; z < 8 ; z++) // set up digital pins
  {
    pinMode(z, OUTPUT);
  }
}

1 void translateIR()
  // takes action based on IR code received
  // uses Sony IR codes
  {
    switch(results.value)
    {
2      case 0x410:  pinOn(3); break; // 3
        case 0xC10:  pinOn(4); break; // 4
        case 0x210:  pinOn(5); break; // 5
        case 0xA10:  pinOn(6); break; // 6
        case 0x610:  pinOn(7); break; // 7
    }
  }

3 void pinOn(int pin) // turns on digital pin "pin" for 1 second
  {
    digitalWrite(pin, HIGH);
    delay(1000);
    digitalWrite(pin, LOW);
  }

  void loop()
  {
4    if (irrecv.decode(&results))    // have we received an IR
    signal?
    {
      translateIR();
5      for (int z = 0 ; z < 2 ; z++) // ignore the 2nd and 3rd
      repeated codes
      {
        irrecv.resume();           // receive the next value
      }
    }
  }
}

```

This sketch has three major parts. First, it waits for a signal from the remote at 4. When a signal is received, the signal is tested in the function

translateIR() at 1 to determine which button was pressed and what action to take.

Notice at 2 that we compare the hexadecimal codes returned by the IR library. These are the codes returned by the test conducted in [Listing 17-1](#). When the codes for buttons 3 through 7 are received, the function pinOn() at 3 is called, which turns on the matching digital pin for 1 second.

As mentioned, Sony remotes send the code three times for each button press, so we use a small loop at 5 to ignore the second and third codes. Finally, note the addition of 0x in front of the hexadecimal numbers used in the case statements at 2.

NOTE

Hexadecimal numbers are base 16 and use the digits 0 through 9 and then A through F, before moving on to the next column. For example, decimal 10 in hexadecimal is A, decimal 15 in hexadecimal is F, decimal 16 is 10 in hexadecimal, and so on. When using a hexadecimal number in a sketch, preface it with 0x.

Modifying the Sketch

You can expand the options or controls available for controlling your receiving device by testing more buttons. To do so, use [Listing 17-1](#) to determine which button creates which code and then add each new code to the switch case statement.

Project #51: Creating an IR Remote Control Robot Vehicle

To show you how to integrate an IR remote control into an existing project, we'll add IR to the robot described in Project 39 in Chapter 14. In this project, instead of presetting the robot's direction and distances, the sketch will show you how to control these actions with a simple Sony TV remote.

The Hardware

The required hardware is the same as that required for the robot you built for Project 39, with the addition of the IR receiver module described earlier in this chapter. In the following sketch, the robot will respond to the buttons that you press on the remote control as follows: press 2 for forward, 8 for backward, 4 for rotate left, and 6 for rotate right.

The Sketch

After reassembling your vehicle and adding the IR receiver, enter and upload the following sketch:

```
// Project 51 - Creating an IR Remote Control Robot Vehicle
int receiverpin = 2; // pin 1 of IR receiver to Arduino
digital pin 11
#include <IRremote.h>
IRrecv irrecv(receiverpin); // create instance of irrecv
decode_results results;
#include <AFMotor.h>
AF_DCMotor motor1(1); // set up instances of each motor
AF_DCMotor motor2(2);
AF_DCMotor motor3(3);
AF_DCMotor motor4(4);

void goForward(int speed, int duration)
{
  motor1.setSpeed(speed);
  motor2.setSpeed(speed);
  motor3.setSpeed(speed);
  motor4.setSpeed(speed);
  motor1.run(FORWARD);
  motor2.run(FORWARD);
  motor3.run(FORWARD);
  motor4.run(FORWARD);
  delay(duration);
  motor1.run(RELEASE);
  motor2.run(RELEASE);
  motor3.run(RELEASE);
  motor4.run(RELEASE);
}

void goBackward(int speed, int duration)
{
  motor1.setSpeed(speed);
```

```

    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(BACKWARD);
    motor2.run(BACKWARD);
    motor3.run(BACKWARD);
    motor4.run(BACKWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}

void rotateLeft(int speed, int duration)
{
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(FORWARD);
    motor2.run(BACKWARD);
    motor3.run(BACKWARD);
    motor4.run(FORWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}

void rotateRight(int speed, int duration)
{
    motor1.setSpeed(speed);
    motor2.setSpeed(speed);
    motor3.setSpeed(speed);
    motor4.setSpeed(speed);
    motor1.run(BACKWARD);
    motor2.run(FORWARD);
    motor3.run(FORWARD);
    motor4.run(BACKWARD);
    delay(duration);
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
}

```

```

// translateIR takes action based on IR code received, uses
Sony IR codes
void translateIR()
{
    switch (results.value)
    {
        case 0x810:
            goForward(255, 250);
            break; // 2
        case 0xC10:
            rotateLeft(255, 250);
            break; // 4
        case 0xA10:
            rotateRight(255, 250);
            break; // 6
        case 0xE10:
            goBackward(255, 250);
            break; // 8
    }
}

void setup()
{
    delay(5000);
    irrecv.enableIRIn();           // start IR receiver
}

void loop()
{
    if (irrecv.decode(&results)) // have we received an IR
signal?
    {
        translateIR();
        for (int z = 0 ; z < 2 ; z++) // ignore the repeated
codes
        {
            irrecv.resume();           // receive the next value
        }
    }
}

```

This sketch should look somewhat familiar to you. Basically, instead of lighting up LEDs on digital pins, it calls the motor control functions that were used in the robot vehicle from Chapter 14.

Looking Ahead

Having worked through the projects in this chapter, you should understand how to send commands to your Arduino via an infrared remote control device. Using these skills and your knowledge from previous chapters, you now can replace physical forms of input such as buttons with a remote control.

But the fun doesn't stop here. In the next chapter, we'll use an Arduino to harness something that, to the untrained eye, is fascinating and futuristic: radio frequency identification systems.

18

READING RFID TAGS

In this chapter you will

Learn how to implement RFID readers with your Arduino

See how to save variables in the Arduino EEPROM

Design the framework for an Arduino-based RFID access system

Radio-frequency identification (RFID) is a wireless system that uses electromagnetic fields to transfer data from one object to another, without the two objects touching. You can build an Arduino that reads common RFID tags and cards to create access systems and to control digital outputs. You may have used an RFID card before, such as an access card that you use to unlock a door or a public transport card that you hold in front of a reader on the bus. [*Figure 18-1*](#) shows some examples of RFID tags and cards.



Figure 18-1: Example RFID devices

Inside RFID Devices

Inside an RFID tag is a tiny integrated circuit with memory that can be accessed by a specialized reader. Most tags don't have a battery inside; instead, a wire coil antenna in the RFID reader broadcasts a jolt of electromagnetic energy to the tags. They absorb this energy and use it to power their own circuitry, which broadcasts a response back to the RFID reader. [Figure 18-2](#) shows the antenna coil of the RFID reader that we'll use in this chapter.

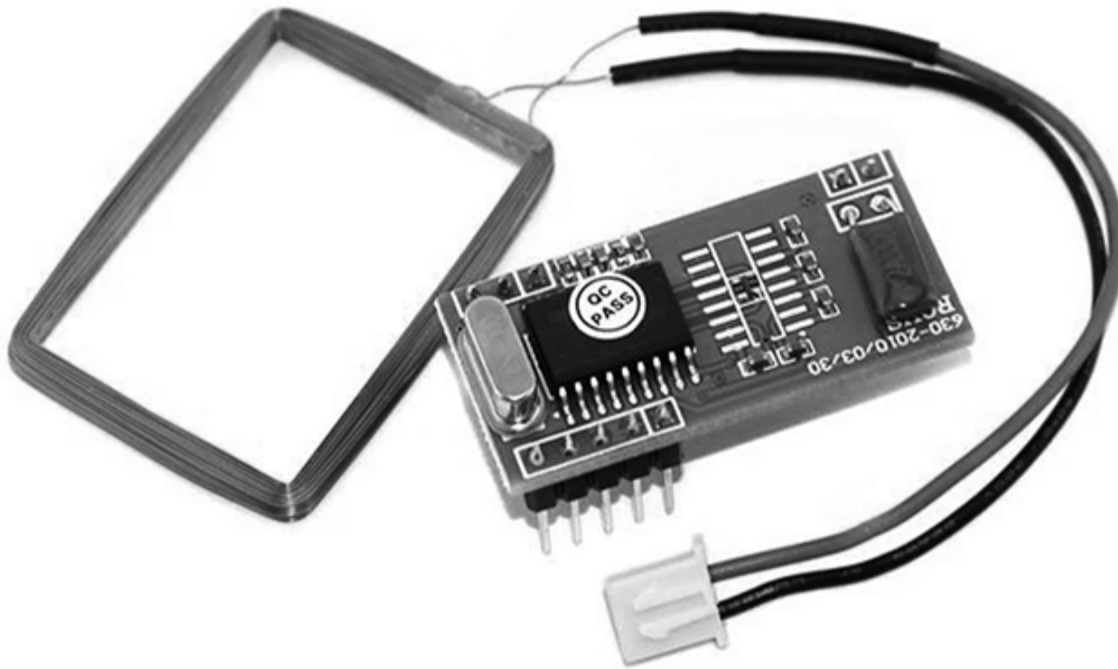


Figure 18-2: Our RFID reader

The card reader we'll use in this chapter is from PMD Way (part number 113990014). It's cheap and easy to use, and it operates at 125 kHz; be sure to purchase two or more RFID tags that match that frequency, such as those found at <https://pmdway.com/collections/rfid-tags/>.

Testing the Hardware

In this section, you'll connect the RFID reader to the Arduino. Then you'll test that it's working with a simple sketch that reads RFID cards and sends the data to the Serial Monitor. To avoid conflict with the serial port between the PC and Arduino, the RFID will be connected to other digital pins and use SoftwareSerial, as we did in Chapter 15 with the GPS receiver module.

The Schematic

[Figure 18-3](#) shows a diagram of the RFID module connections, looking at the top side of the module.

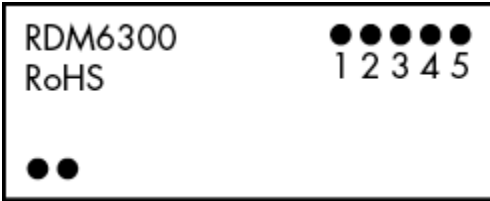


Figure 18-3: RFID module connections

Testing the Schematic

To make the connections between the RFID reader and the Arduino, follow these steps, using female-to-male jumper wires:

- Connect the included coil plug to the antenna pins at the bottom left of the RFID reader board. They are not polarized and can connect either way.
- Connect the reader's GND (pin 2) to Arduino GND.
- Connect the reader's 5 V (pin 1) to Arduino 5 V.
- Connect the reader's RX (pin 4) to Arduino pin D3.
- Connect the reader's TX (pin 5) to Arduino pin D2.

The Test Sketch

Enter and upload [Listing 18-1](#).

```
// Listing 18-1
#include <SoftwareSerial.h>
SoftwareSerial Serial2(2, 3);
int data1 = 0;

void setup()
{
  Serial.begin(9600);
  Serial2.begin(9600);
}

void loop()
{
  if (Serial2.available() > 0) {
    data1 = Serial2.read();
    // display incoming number
    Serial.print(" ");
```



```
    Serial.print(data1, DEC);  
  }
```

Listing 18-1: RFID test sketch

Displaying the RFID Tag ID Number

Open the Serial Monitor window and wave an RFID tag over the coil. The results should look similar to [Figure 18-4](#).

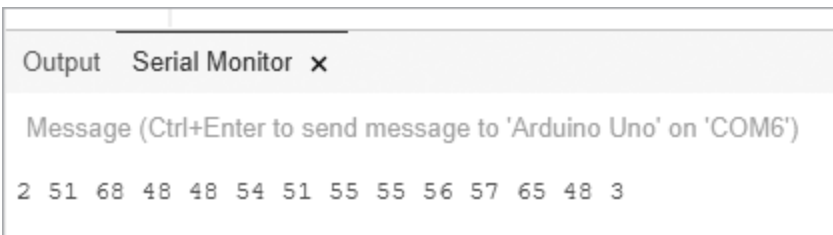


Figure 18-4: Example output from [Listing 18-1](#)

Notice that 14 numbers are displayed in the Serial Monitor window. Collectively, these are the RFID tag's unique ID number, which we'll use in future sketches to identify the tag being read. Scan all your RFID tags and record their ID numbers, because you'll need them for the next few projects.

Project #52: Creating a Simple RFID Control System

Now let's put the RFID system to use. In this project, you'll learn how to trigger an Arduino event when one of two correct RFID tags is read. The sketch stores two RFID tag numbers; when a card whose ID matches one of those numbers is read by the reader, it will display *Accepted* in the Serial Monitor. If a card whose ID does not match one of the stored IDs is presented, then the Serial Monitor will display *Rejected*. We'll use this as a base to add RFID controls to existing projects.

The Sketch

Enter and upload the following sketch. However, at 1 and 2, replace the x's in the array with the set of numbers you noted for two of your RFID tags in the previous section. (We discussed arrays in Chapter 6.) // Project 52 -

```

Creating a Simple RFID Control System #include
<SoftwareSerial.h> SoftwareSerial Serial2(2, 3); int data1 = 0;
int ok = -1; // use Listing 18-1 to find your tags' numbers
int tag1[14] = {x, x, x, x, x, x, x, x, x, x, x, x, x, x};
int tag2[14] = {x, x, x, x, x, x, x, x, x, x, x, x, x, x};
int newtag[14] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; //
used for read comparison
boolean comparetag(int aa[14], int
bb[14]) { boolean ff = false; int fg = 0; for (int cc = 0 ; cc
< 14 ; cc++) { if (aa[cc] == bb[cc]) { fg++; } } if (fg == 14)
{ ff = true; } return ff;
}
void checkmytags() // compares each
tag against the tag just read { ok = 0; // This variable
supports decision making. // If it is 1, we have a match; 0 is
a read but no match, // -1 is no read attempt made. if
(comparetag(newtag, tag1) == true) { ok++; } if
(comparetag(newtag, tag2) == true) { ok++; } }
void setup() {
Serial.begin(9600); Serial2.begin(9600); Serial2.flush(); //
need to flush serial buffer // otherwise first read may not be
correct }
void loop() { ok = -1; if (Serial2.available() > 0)
// if a read has been attempted { // read the incoming number
on serial RX delay(100); // needed to allow time for the data
// to come in from the serial buffer
for (int z = 0 ; z < 14 ;
z++) // read the rest of the tag { data1 = Serial2.read();
newtag[z] = data1; } Serial2.flush(); // stops multiple reads
// now to match tags
checkmytags();
// now do something
based on tag type if (ok > 0) // if we had a match {
Serial.println("Accepted"); ok = -1; } else if (ok == 0) // if
we didn't have a match { Serial.println("Rejected"); ok = -1; }
}

```

Understanding the Sketch

When a tag is presented to the RFID reader, it sends the tag's numbers, which collectively are its ID number, through the serial port. We capture all 14 of these numbers and place them in the array newtag[] at 7. Next, the

tag ID is compared against the two tag ID numbers stored at 1 and 2 using the function `checkmytags()` at 4 and 8, with the actual comparisons of the tag arrays performed by the function `comparetag()` at 3.

The `comparetag()` function accepts the two number arrays as parameters and returns (in Boolean) whether the arrays are identical (`true`) or different (`false`). If a match is made, the variable `ok` is set to 1 at 5 and 6. Finally, at 9, we have the actions to take once the tag read succeeds.

After uploading the sketch, open the Serial Monitor window and present some tags to the reader. The results should be similar to those in [Figure 18-5](#).

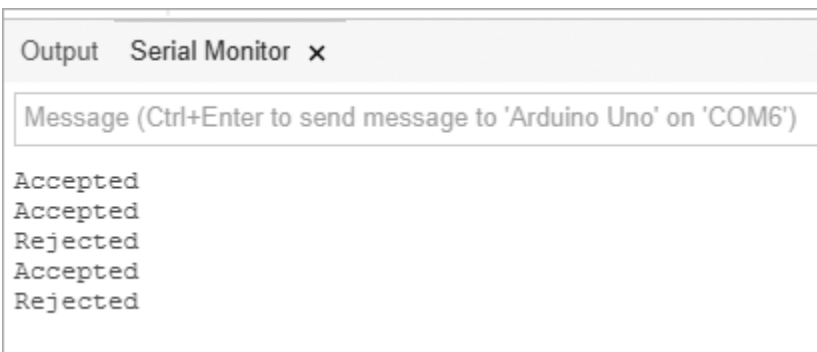


Figure 18-5: Results of Project 52

Storing Data in the Arduino's Built-in EEPROM

When you define and use a variable in your Arduino sketches, the stored data lasts only until the Arduino is reset or the power is turned off. But what if you want to keep the values for future use, as in the case of the user-changeable secret code for the numeric keypad in Chapter 11? That's where the *EEPROM* (*electrically erasable read-only memory*) comes in. The EEPROM stores variables in memory inside an ATmega328 microcontroller, and the values aren't lost when the power is turned off.

The EEPROM in the Arduino can store 1,024-byte variables in positions numbered from 0 to 1,023. Recall that a byte can store an integer with a value between 0 and 255, and you begin to see why it's perfect for storing RFID tag numbers. To use the EEPROM in our sketches, we first call the

EEPROM library (included with the Arduino IDE) using the following:

```
#include <EEPROM.h>
```

Then, to write a value to the EEPROM, we simply use this:

```
EEPROM.write(a, b);
```

Here, *a* is the position in the EEPROM memory where the information will be stored, and *b* is the variable holding the information we want to store in the EEPROM at position *a*.

To retrieve data from the EEPROM, we use this function:

```
value = EEPROM.read(position);
```

This takes the data stored in EEPROM position number *position* and stores it in the variable *value*.

NOTE

The EEPROM has a finite life, and it can eventually wear out! According to the manufacturer, Atmel, it can sustain 100,000 write/erase cycles for each position. Reads are unlimited.

Reading and Writing to the EEPROM

Here's an example of how to read and write to the EEPROM. Enter and upload [Listing 18-2](#).

```
// Listing 18-2
#include <EEPROM.h>
int zz;

void setup()
{
  Serial.begin(9600);
  randomSeed(analogRead(0));
}

void loop()
{
```

```

    Serial.println("Writing random numbers...");
    for (int i = 0; i < 1024; i++)
    {
        zz = random(255);
1      EEPROM.write(i, zz);
    }
    Serial.println();
    for (int a = 0; a < 1024; a++)
    {
2      zz = EEPROM.read(a);
        Serial.print("EEPROM position: ");
        Serial.print(a);
        Serial.print(" contains ");
3      Serial.println(zz);
        delay(25);
    }
}

```

Listing 18-2: EEPROM demonstration sketch In the loop at 1, a random number between 0 and 255 is stored in each EEPROM position. The stored values are retrieved in the second loop at 2, to be displayed in the Serial Monitor at 3.

Once the sketch has been uploaded, open the Serial Monitor. You should see something like [Figure 18-6](#).

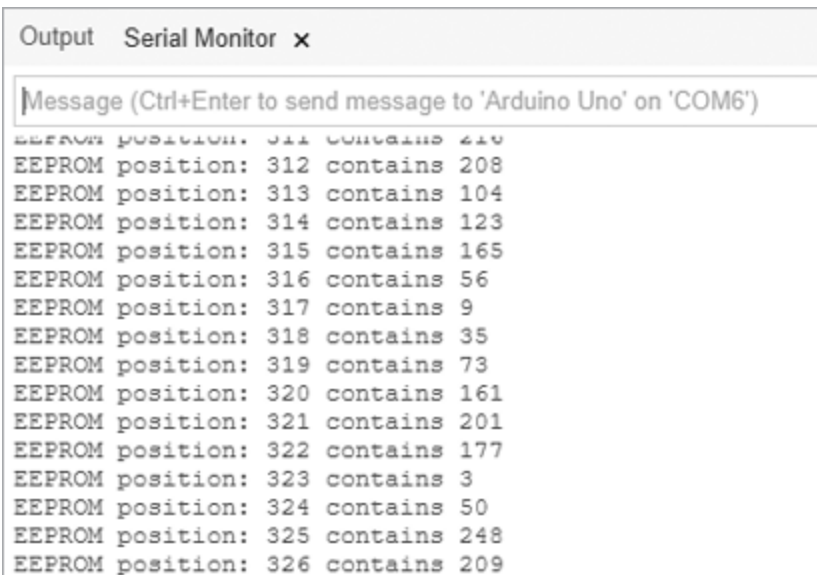


Figure 18-6: Example output from [Listing 18-2](#)

Now you're ready to create a project using the EEPROM.

Project #53: Creating an RFID Control with "Last Action" Memory

Although Project 52 showed how to use RFID to control something, such as a light or electric door lock, we had to assume that nothing would be remembered if the system were reset or the power went out. For example, if a light was on and the power went out, then the light would be off when the power returned. However, you may prefer the Arduino to remember what was happening before the power went out and return to that state. Let's solve that problem now.

In this project, the last action will be stored in the EEPROM (for example, "locked" or "unlocked"). When the sketch restarts after a power failure or an Arduino reset, the system will revert to the previous state stored in the EEPROM.

The Sketch

Enter and upload the following sketch. As you did for Project 52, replace each x in the arrays at 1 and 2 with the numbers for two of your RFID tags.

```
// Project 53 - Creating an RFID Control with "Last Action"
Memory
#include <SoftwareSerial.h>
SoftwareSerial Serial2(2, 3);
#include <EEPROM.h>
int data1 = 0;
int ok = -1;
int lockStatus = 0;
// use Listing 18-1 to find your tags' numbers
1 int tag1[14] = {
    x, x, x, x, x, x, x, x, x, x, x, x, x, x
};
2 int tag2[14] = {
    x, x, x, x, x, x, x, x, x, x, x, x, x, x
};
int newtag[14] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
}
```

```

}; // used for read comparisons

// comparetag compares two arrays and returns true if
identical
// this is good for comparing tags
boolean comparetag(int aa[14], int bb[14])
{
    boolean ff = false;
    int fg = 0;
    for (int cc = 0; cc < 14; cc++)
    {
        if (aa[cc] == bb[cc])
        {
            fg++;
        }
    }
    if (fg == 14)
    {
        ff = true;
    }
    return ff;
}

void checkmytags()
// compares each tag against the tag just read
{
    ok = 0;
    if (comparetag(newtag, tag1) == true)
    {
        ok++;
    }
    if (comparetag(newtag, tag2) == true)
    {
        ok++;
    }
}

3 void checkLock()
{
    Serial.print("System Status after restart ");
    lockStatus = EEPROM.read(0);
    if (lockStatus == 1)
    {
        Serial.println("- locked");
        digitalWrite(13, HIGH);
    }
    if (lockStatus == 0)
    {
        Serial.println("- unlocked");
    }
}

```

```

        digitalWrite(13, LOW);
    }
    if ((lockStatus != 1) && (lockStatus != 0))
    {
        Serial.println("EEPROM fault - Replace Arduino
hardware");
    }
}
void setup()
{
    Serial.begin(9600);
    Serial2.begin(9600);
    Serial2.flush(); // need to flush serial buffer
    pinMode(13, OUTPUT);
4   checkLock();
}
void loop()
{
    ok = -1;
    if (Serial2.available() > 0)    // if a read has been
attempted
    {
        // read the incoming number on serial RX
        delay(100);
        for (int z = 0; z < 14; z++) // read the rest of the tag
        {
            data1 = Serial2.read();
            newtag[z] = data1;
        }
        Serial2.flush();           // prevents multiple reads
        // now to match tags up
        checkmytags();
    }
5   if (ok > 0)                    // if we had a match
    {
        lockStatus = EEPROM.read(0);
        if (lockStatus == 1)       // if locked, unlock it
        {
6            Serial.println("Status - unlocked");
            digitalWrite(13, LOW);
            EEPROM.write(0, 0);
        }
        if (lockStatus == 0)
        {
7            Serial.println("Status - locked");
            digitalWrite(13, HIGH);

```



```

        EEPROM.write(0, 1);
    }
    if ((lockStatus != 1) && (lockStatus != 0))
    {
8      Serial.println("EEPROM fault - Replace Arduino
hardware");
    }
    else if (ok == 0)                // if we didn't have a match
    {
        Serial.println("Incorrect tag");
        ok = -1;
    }
    delay(500);
}

```

Understanding the Sketch

This sketch is a modification of Project 52. We use the onboard LED to simulate the status of something that we want to turn on or off every time an acceptable RFID ID tag is read. After a tag has been read and matched, the status of the lock is changed at 5. We store the status of the lock in the first position of the EEPROM. The status is represented by a number: 0 is unlocked and 1 is locked. This status will change (from locked to unlocked and back to locked) after every successful tag read at 6 or 7.

We've also introduced a fail-safe in case the EEPROM has worn out. If the value returned from reading the EEPROM is not 0 or 1, we should be notified at 8. Furthermore, the status is checked when the sketch restarts after a reset using the function `checkLock()` at 1, 2, 3, and 4, which reads the EEPROM value, determines the last status, and then sets the lock to that status (locked or unlocked).

Looking Ahead

Once again, we have used an Arduino board to re-create simply what could be a very complex project. You now have a base to add RFID control to your projects that will allow you to create professional-quality access systems and control digital outputs with the swipe of an RFID card. We'll demonstrate this again when we revisit RFID in Chapter 20.

19

DATA BUSES

In this chapter you will

Learn about the I²C bus

See how to use an EEPROM and a port expander on the I²C bus

Learn about the SPI bus

Find out how to use a digital rheostat on the SPI bus

An Arduino communicates with other devices via a *data bus*, a system of connections that allows two or more devices to exchange data in an orderly manner. A data bus can provide a connection between the Arduino and various sensors, I/O expansion devices, and other components.

The two major buses of most importance to the Arduino are the *Serial Peripheral Interface (SPI)* bus and the *Inter-Integrated Circuit (I²C)* bus. Many useful sensors and external devices communicate using these buses.

The I²C Bus

The I²C bus, also known as the *Two-Wire Interface (TWI)* bus, is a simple and easy-to-use data bus. Data is transferred between devices and the Arduino through two wires, known as *SDA* and *SCL* (the data line and clock line, respectively). In the case of the Arduino Uno, the SDA pin is A4 and the SCL pin is A5, as shown in [Figure 19-1](#).

Some newer R3 boards also have dedicated I²C pins at the upper-left corner for convenient access, as shown in [Figure 19-2](#). If you use these two pins,

you cannot use the A4 and A5 pins for other purposes.

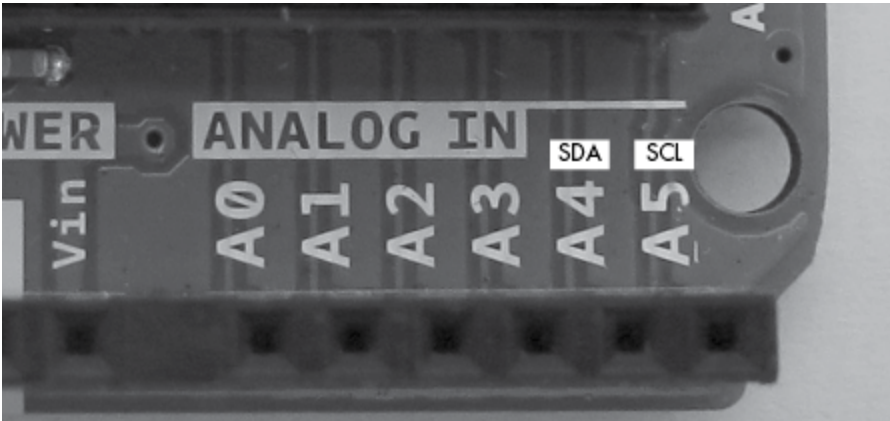


Figure 19-1: The I²C bus connectors on the Arduino Uno

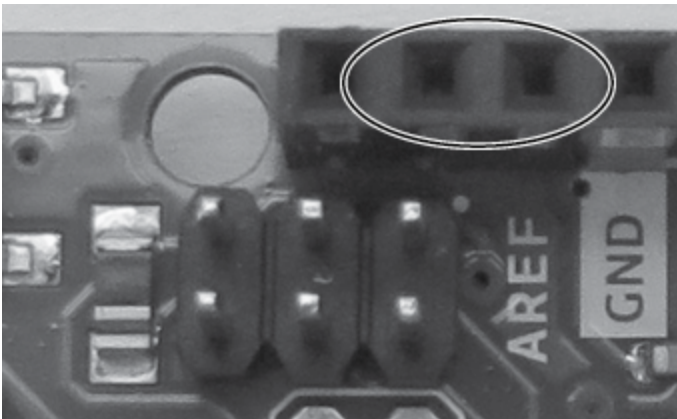


Figure 19-2: Additional dedicated I²C pins

As the six pins used for reprogramming the USB interface microcontroller take up the space normally used for pin labeling, you can see the labels on the rear of the Arduino, as shown in [Figure 19-3](#).

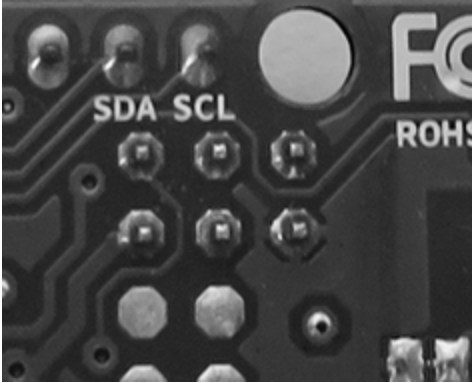


Figure 19-3: Labels for additional dedicated I²C pins

On the I²C bus, the Arduino is the *main device*, and each IC out on the bus is a *secondary*. Each secondary has its own address, a hexadecimal number that allows the Arduino to address and communicate with each device. Each device usually has a range of 7-bit I²C bus addresses to choose from, which is detailed in the manufacturer's data sheet. The particular addresses available are determined by wiring the IC pins a certain way.

NOTE

Because the Arduino runs on 5 V, your I²C device must also operate on 5 V or at least be able to tolerate it. Always confirm this by contacting the seller or manufacturer before use.

To use the I²C bus, you'll need to use the Wire library (included with the Arduino IDE):

```
#include <Wire.h>
```

Next, in `void setup()`, activate the bus with this:

```
Wire.begin();
```

Data is transmitted along the bus 1 byte at a time. To send a byte of data from the Arduino to a device on the bus, three functions are required:

The first function initiates communication with the following line of code (where *address* is the secondary device's bus address in hexadecimal—for example 0x50):

```
Wire.beginTransmission(address);
```

The second function sends 1 byte of data from the Arduino to the device addressed by the previous function (where *data* is a variable containing 1 byte of data; you can send more than 1 byte, but you'll need to use one `Wire.write()` call for each byte):

```
Wire.write(data);
```

Finally, once you have finished sending data to a particular device, use this to end the transmission:

```
Wire.endTransmission();
```

To request that data from an I²C device be sent to the Arduino, start with `Wire.beginTransmission(address)`, followed by the this code (where *x* is the number of bytes of data to request):

```
Wire.requestFrom(address, x);
```

Next, use the following function to store each incoming byte into a variable:

```
incoming = Wire.read(); // incoming is the variable receiving  
the byte of data
```

Then finalize the transaction with `Wire.endTransmission()`. We'll put these functions to use in the next project.

Project #54: Using an External EEPROM

In Chapter 18, we used the Arduino's internal EEPROM to prevent the erasure of variable data caused by a board reset or power failure. The

Arduino's internal EEPROM stores only 1,024 bytes of data. To store more data, you can use external EEPROMs, as you'll see in this project.



Figure 19-4: Microchip Technology's 24LC512 EEPROM

For our external EEPROM, we'll use the Microchip Technology 24LC512 EEPROM, which can store 64KB (65,536 bytes) of data ([Figure 19-4](#)). It's available from retailers such as Digi-Key (part number 24LC512-I/P-ND) and PMD Way (part number 24LC512A).

The Hardware

Here's what you'll need to create this project:

Arduino and USB cable

One Microchip Technology 24LC512 EEPROM

One breadboard

Two 4.7 k Ω resistors

One 100 nF ceramic capacitor

Various connecting wires

The Schematic

For the circuit, connect one 4.7 k Ω resistor between 5 V and SCL and the other between 5 V and SDA, as shown in [Figure 19-5](#).

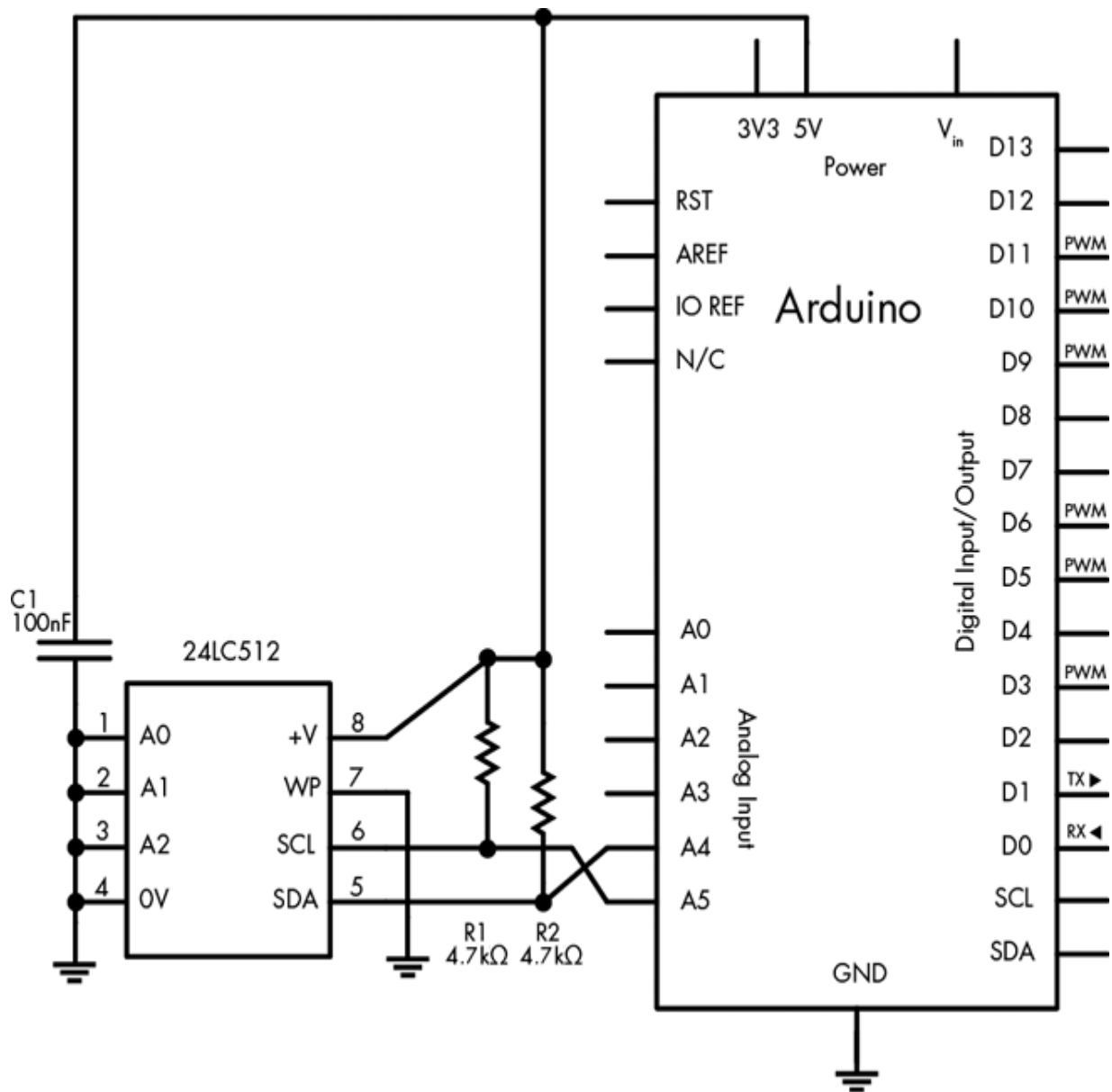


Figure 19-5: Schematic for Project 54

The bus address for the 24LC512 EEPROM IC is partially determined by the way it is wired into the circuit. The last 3 bits of the bus address are determined by the status of pins A2, A1, and A0. When these pins are connected to GND, their values are 0; when they are connected to 5 V, their values are 1.

The first 4 bits are preset as 1010. Therefore, in our circuit, since A0, A1, and A2 are connected directly to GND, the bus address is represented as

1010000 in binary, which is 0x50 in hexadecimal. This means that we can use 0x50 as the bus address in the sketch.

The Sketch

Although our external EEPROM can store up to 64KB of data, our sketch is intended to demonstrate just a bit of its use, so we'll store and retrieve bytes only in the EEPROM's first 20 memory positions.

Enter and upload the following sketch:

```
// Project 54 - Using an External EEPROM
1 #include <Wire.h>
   #define chip1 0x50

   byte d=0;

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

   void writeData(int device, unsigned int address, byte data)
   // writes a byte of data 'data' to the EEPROM at I2C address
   // 'device'
   // in memory location 'address'
   {
3     Wire.beginTransmission(device);
       Wire.write((byte)(address >> 8)); // left part of pointer
address
       Wire.write((byte)(address & 0xFF)); // and the right
       Wire.write(data);
       Wire.endTransmission();
       delay(10);
   }

4 byte readData(int device, unsigned int address)
   // reads a byte of data from memory location 'address'
   // in chip at I2C address 'device'
   {
       byte result; // returned value
       Wire.beginTransmission(device);
       Wire.write((byte)(address >> 8)); // left part of pointer
```



```

    address
    Wire.write((byte)(address & 0xFF)); // and the right
    Wire.endTransmission();
5  Wire.requestFrom(device,1);
    result = Wire.read();
    return result; // and return it as a result of the function
readData
}

void loop()
{
    Serial.println("Writing data...");
    for (int a=0; a<20; a++)
    {
        writeData(chip1,a,a);
    }
    Serial.println("Reading data...");
    for (int a=0; a<20; a++)
    {
        Serial.print("EEPROM position ");
        Serial.print(a);
        Serial.print(" holds ");
        d=readData(chip1,a);
        Serial.println(d, DEC);
    }
}

```

Let's walk through the sketch. At 1, we activate the library and define the I²C bus address for the EEPROM as chip1. At 2, we start the Serial Monitor and then the I²C bus. The two custom functions writeData() and readData() are included to save you time and give you some reusable code for future work with this EEPROM IC. We'll use them to write and read data, respectively, from the EEPROM.

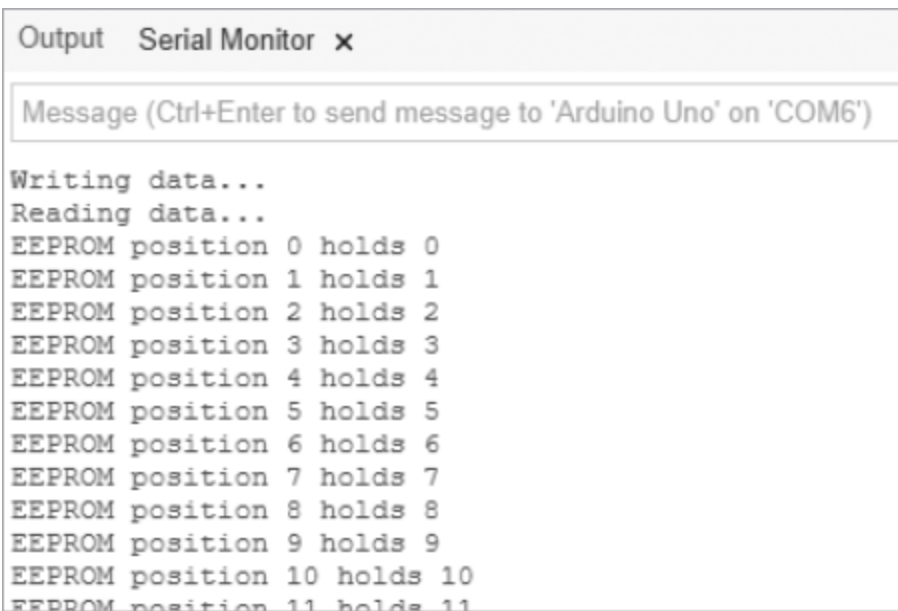
The function writeData() at 3 initiates transmission with the EEPROM, sends the address of where to store the byte of data in the EEPROM using the next two wire.write() function calls, sends a byte of data to be written, and then ends transmission.

The function readData() at 4 operates the I²C bus in the same manner as writeData(). First, however, it sets the address to read from, and then instead of sending a byte of data to the EEPROM, it uses wire.requestFrom() to read the data at 5. Finally, the byte of data sent

from the EEPROM is received into the variable `result` and becomes the return value for the function.

Running the Sketch

In `void loop()`, the sketch loops 20 times and writes a value to the EEPROM each time. Then it loops again, retrieving the values and displaying them in the Serial Monitor, as shown in [Figure 19-6](#).



```
Output  Serial Monitor x
Message (Ctrl+Enter to send message to 'Arduino Uno' on 'COM6')

Writing data...
Reading data...
EEPROM position 0 holds 0
EEPROM position 1 holds 1
EEPROM position 2 holds 2
EEPROM position 3 holds 3
EEPROM position 4 holds 4
EEPROM position 5 holds 5
EEPROM position 6 holds 6
EEPROM position 7 holds 7
EEPROM position 8 holds 8
EEPROM position 9 holds 9
EEPROM position 10 holds 10
EEPROM position 11 holds 11
```

[Figure 19-6](#): Results of Project 54

Project #55: Using a Port Expander IC

A *port expander* is another useful IC that is controlled via I²C. It's designed to offer more digital output pins. In this project, we'll use the Microchip Technology MCP23017 16-bit port expander IC ([Figure 19-7](#)), which has 16 digital outputs to add to your Arduino. It is available from retailers such as Digi-Key (part number MCP23017-E/SP-ND) and PMD Way (part number MCP23017A).

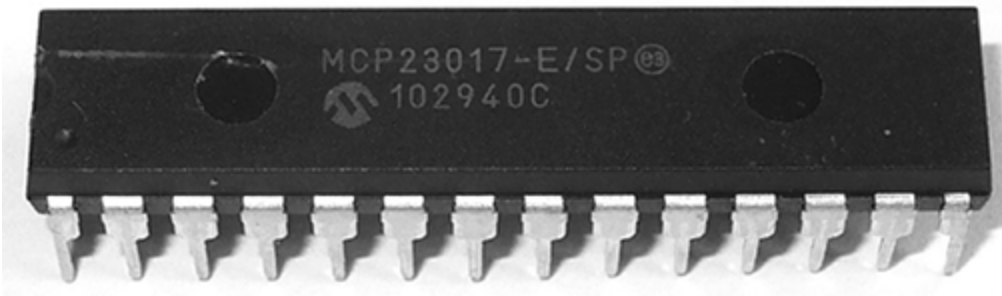


Figure 19-7: Microchip Technology's MCP23017 port expander IC

In this project, we'll connect the MCP23017 to an Arduino and demonstrate how to control the 16 port expander outputs with the Arduino. Each of the port expander's outputs can be treated like a regular Arduino digital output.

The Hardware

Here's what you'll need to create this project:

Arduino and USB cable

One breadboard

Various connecting wires

One Microchip Technology MCP20317 port expander IC

Two 4.7 k Ω resistors

(Optional) An equal number of 560 Ω resistors and LEDs

The Schematic

[Figure 19-8](#) shows the basic schematic for an MCP23017. As with the EEPROM from Project 54, we can set the I²C bus address by using a specific wiring order. With the MCP23017, we connected pins 15 through 17 to GND to set the address to 0x20.

When you're working with the MCP23017, it helps to have the pinout diagram from the IC's data sheet, as shown in [Figure 19-9](#). Note that the 16 outputs are divided into two banks: GPA7 through GPA0 on the right and GPB0 through GPB7 on the left. We'll connect LEDs via 560 Ω resistors

from some or all of the outputs to demonstrate when the outputs are being activated.

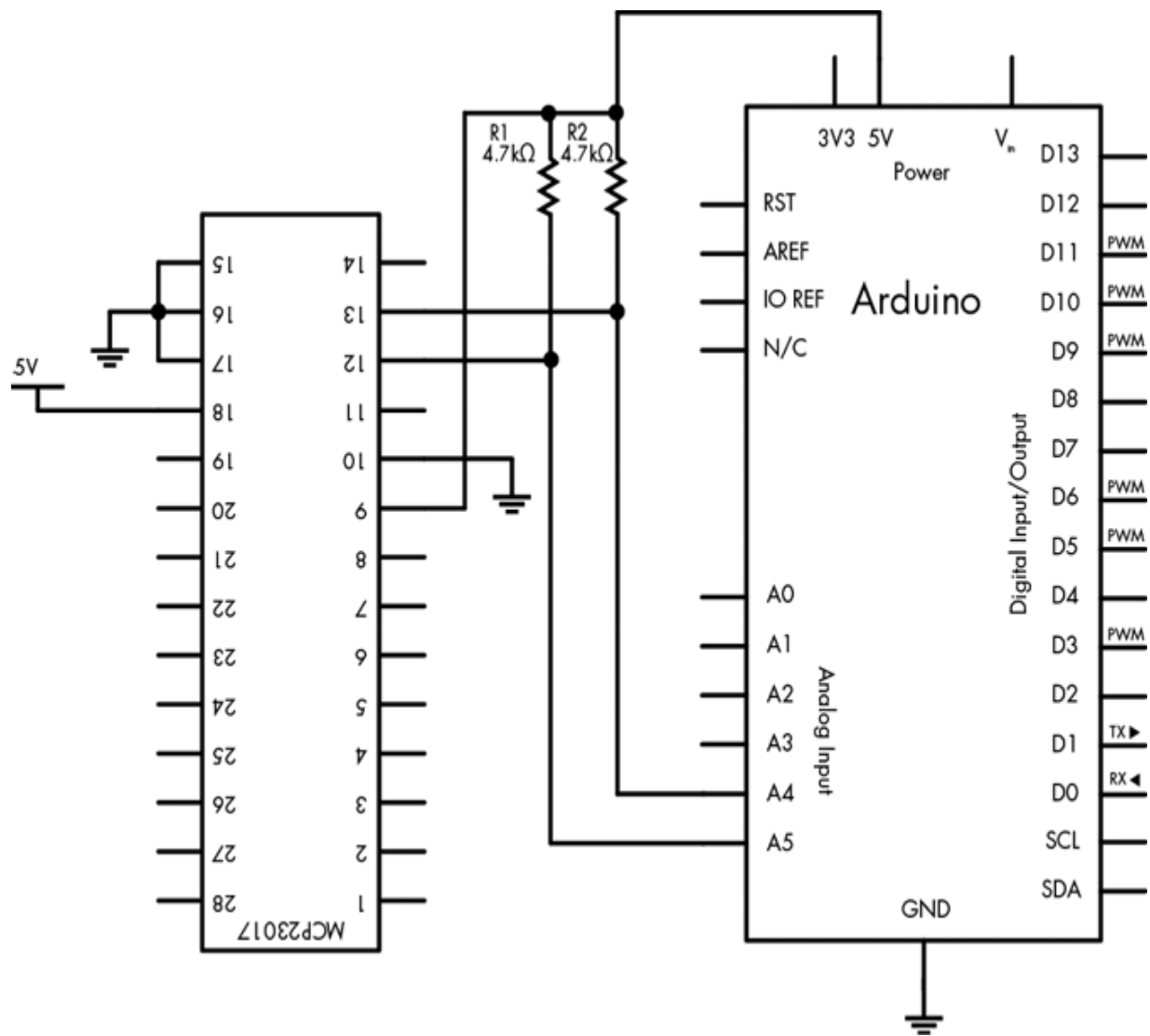


Figure 19-8: Schematic for Project 55

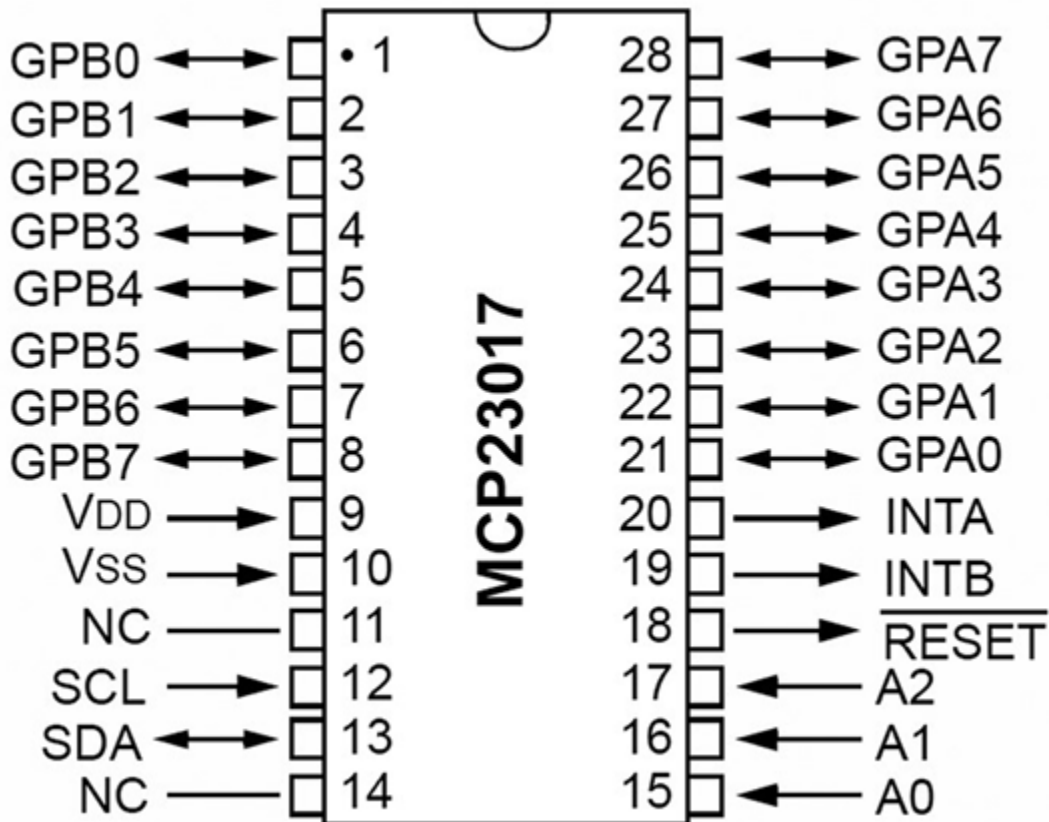


Figure 19-9: Pinout diagram for MCP23017

The Sketch

Enter and upload the following sketch:

```
// Project 55 - Using a Port Expander IC

#include "Wire.h"
#define mcp23017 0x20

void setup()
{
1  Wire.begin();      // activate I2C bus
   // set up MCP23017
   // set I/O pins to outputs
   Wire.beginTransmission(mcp23017);
   Wire.write(0x00); // IODIRA register
   Wire.write(0x00); // set all of bank A to outputs
   Wire.write(0x00); // set all of bank B to outputs
2  Wire.endTransmission();
```

```

}

void loop()
{
    Wire.beginTransmission(mcp23017);
    Wire.write(0x12);
3    Wire.write(255);    // bank A
4    Wire.write(255);    // bank B
    Wire.endTransmission();
    delay(1000);

    Wire.beginTransmission(mcp23017);
    Wire.write(0x12);
    Wire.write(0);    // bank A
    Wire.write(0);    // bank B
    Wire.endTransmission();
    delay(1000);
}

```

To use the MCP23017, we need the lines listed in `void setup()` from 1 through 2. To turn on and off the outputs on each bank, we send 1 byte representing each bank in order; that is, we send a value for bank GPA0 through GPA7 and then a value for GPB0 through GPB7.

When setting individual pins, you can think of each bank as a binary number (as explained in “A Quick Course in Binary” in Chapter 6 on page 104). Thus, to turn on pins 1 through 4, you would send the number 11110000 in binary (240 in decimal), inserted into the `wire.write()` function shown at 3 for bank GPA0 through GPA7 or 4 for bank GPB0 through GPB7.

Hundreds of devices use the I²C bus for communication. Now that you know the basics of how to use this bus, you can use any of these devices with an Arduino board.

The SPI Bus

The SPI bus differs from the I²C bus in that it can be used to send data to and receive data from a device simultaneously and at different speeds, depending on the microcontroller used. Communication, however, is also

main/secondary: the Arduino acts as the main and determines which secondary device it will communicate with at any one time.

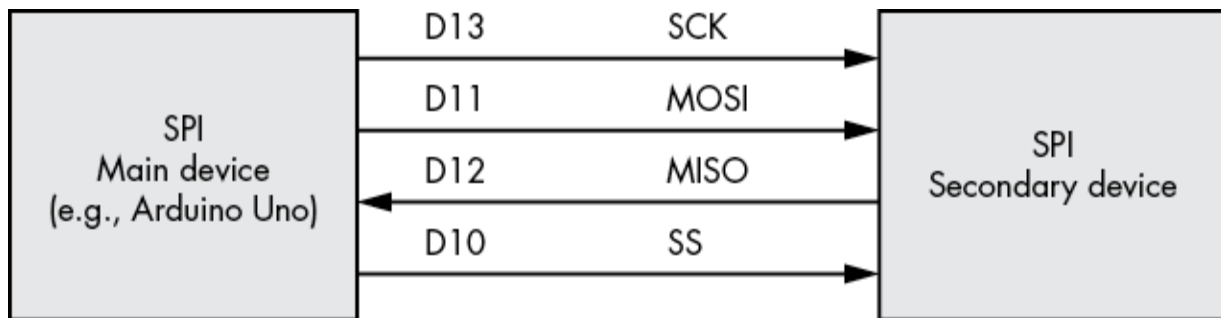
Pin Connections

Each SPI device uses four pins to communicate with a main: *MOSI* (main out, secondary in), *MISO* (main in, secondary out), *SCK* (serial clock), and *SS* or *CS* (secondary select or chip select). These SPI pins are connected to the Arduino as shown in [Figure 19-10](#).



[Figure 19-10](#): SPI pins on an Arduino Uno

A typical single Arduino-to-SPI device connection is shown in [Figure 19-11](#). Arduino pins D11 through D13 are reserved for SPI, but the SS pin can use any other digital pin (often D10 is used because it's next to the SPI pins).



[Figure 19-11](#): Typical Arduino-to-SPI device connection

NOTE

As with I²C devices, your SPI device must either operate on 5 V or tolerate it since the Arduino runs on 5 V. Be sure to check this out with the seller or manufacturer before use.

Implementing the SPI

Now let's examine how to implement the SPI bus in a sketch. Before doing this, however, we'll run through the functions used. First, include the SPI library (included with the Arduino IDE software):

```
#include "SPI.h"
```

Next, you need to choose a pin to be used for SS and set it as a digital output in `void setup()`. Because we're using only one SPI device in our example, we'll use D10 and set it to HIGH first, because most SPI devices have an “active low” SS pin (this means the pin is connected to GND to be set to HIGH, and vice versa):

```
pinMode(10, OUTPUT);  
digitalWrite(10, HIGH);
```

Here is the function to activate the SPI bus:

```
SPI.begin();
```

Finally, we need to tell the sketch which way to send and receive data. Some SPI devices require that their data be sent with the most significant bit first, and some want the MSB last. (Again, see “A Quick Course in Binary” in Chapter 6 for more on MSB.) Therefore, in `void setup()`, we use the following function after `SPI.begin()`:

```
SPI.setBitOrder(order);
```

Here, *order* is either `MSBFIRST` or `MSBLAST`.

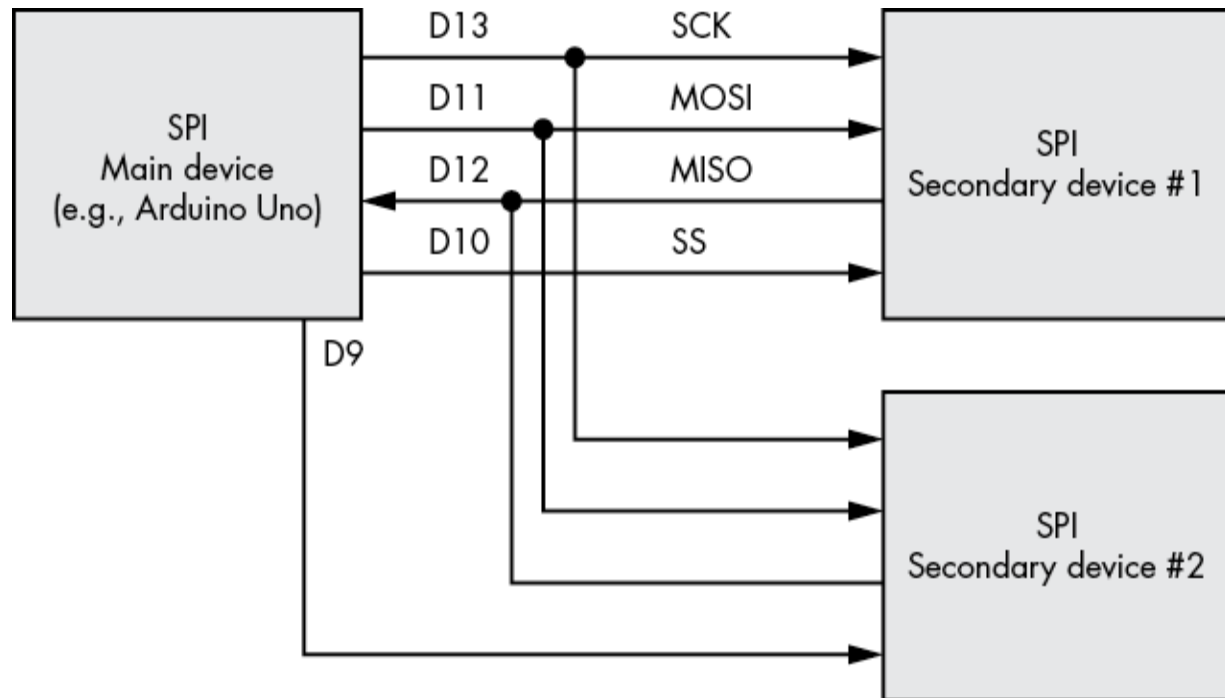
Sending Data to an SPI Device

To send data to an SPI device, we first set the SS pin to LOW, which tells the SPI device that the main (the Arduino) wants to communicate with it. Next, we send bytes of data to the device with the following line, as often as necessary—that is, you use this once for each byte you are sending:

```
SPI.transfer(byte);
```

After you've finished communicating with the device, set the SS pin to HIGH to tell the device that the Arduino has finished communicating with it.

Each SPI device requires a separate SS pin. For example, if you had two SPI devices, the second SPI device's SS pin could be D9 and connected to the Arduino as shown in [Figure 19-12](#).



[Figure 19-12](#): Two SPI devices connected to one Arduino

When communicating with secondary device #2, you would use the D9 (instead of the D10) SS pin before and after communication.

Project 56 demonstrates using the SPI bus with a digital rheostat.

Project #56: Using a Digital Rheostat

In simple terms, a *rheostat* device is similar to the potentiometers we examined in Chapter 4, except the rheostat has two pins: one for the wiper and one for the return current. In this project, you'll use a digital rheostat to set the resistance in the sketch instead of physically turning a potentiometer knob or shaft yourself. Rheostats are often the basis of volume controls in audio equipment that use buttons rather than dials, such as a car stereo. The

tolerance of a rheostat is much larger than that of a normal fixed-value resistor—in some cases, around 20 percent larger.

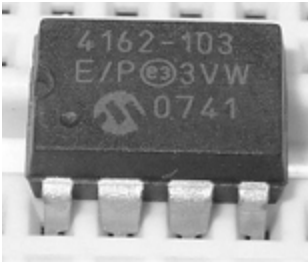


Figure 19-13: Microchip Technology's MCP4162 digital rheostat

For Project 56, we will use the Microchip Technology MCP4162 shown in [Figure 19-13](#). The MCP4162 is available in various resistance values; this example uses the 10 k Ω version. It is available from retailers such as Digi-Key (part number MCP4162-103E/P-ND) and element14 (part number 1840698). The resistance can be adjusted in 257 steps; each step has a resistance of around 40 Ω . To select a particular step, we send 2 bytes of data to a command byte (which is 0) and the value byte (which is between 0 and 256). The MCP4162 uses nonvolatile memory, so when the power is disconnected and then reconnected, the last value selected is still in effect.

We'll control the brightness of an LED using the rheostat.

The Hardware

Here's what you'll need to create this project:

Arduino and USB cable

One breadboard

Various connecting wires

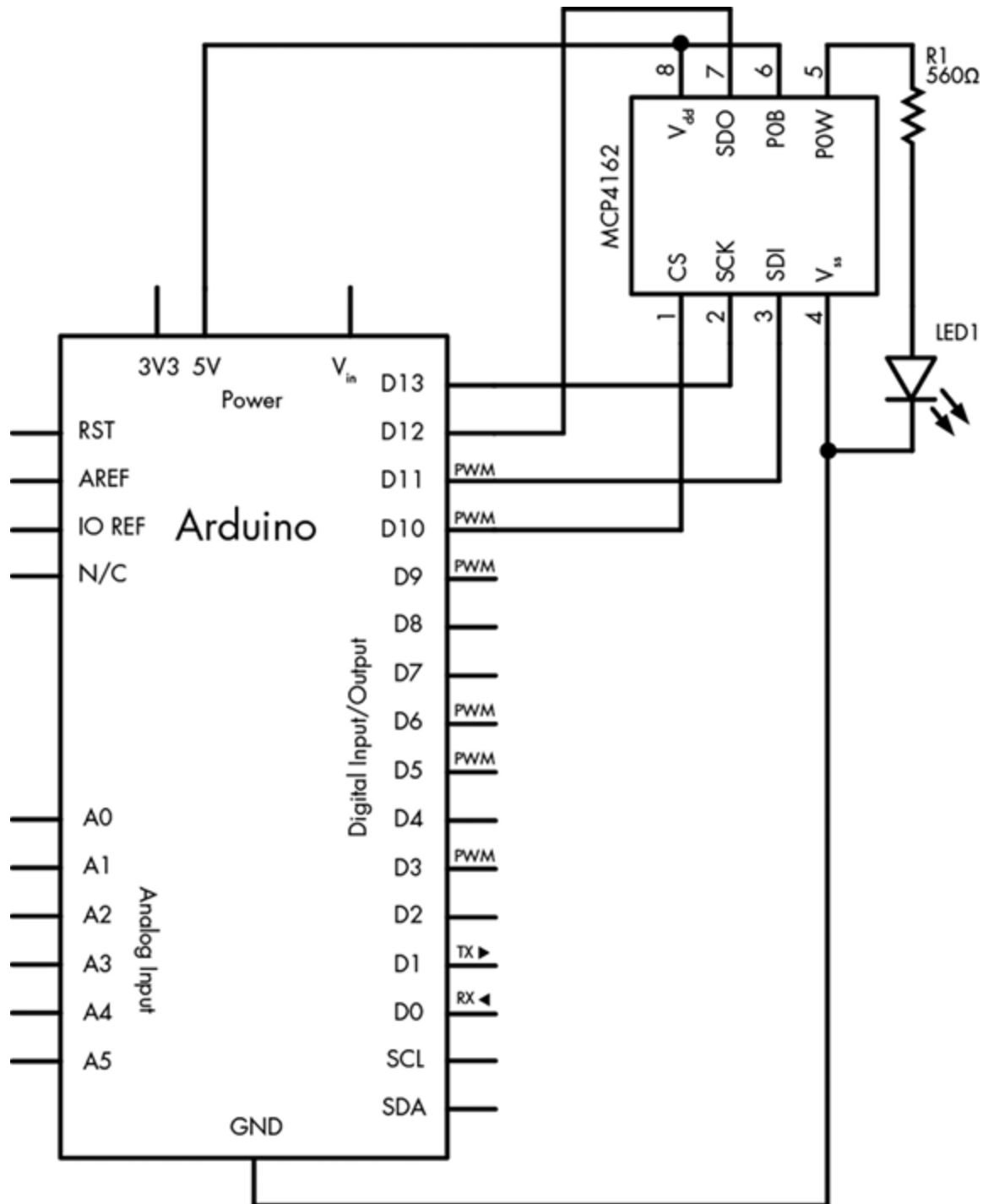
One Microchip Technology MCP4162 digital rheostat

One 560 Ω resistor

One LED

The Schematic

[Figure 19-14](#) shows the schematic. The pin numbering on the MCP4162 starts at the top left of the package. Pin 1 is indicated by the indented dot to the left of the Microchip logo on the IC (see [Figure 19-13](#)).



[Figure 19-14](#): Schematic for Project 56

The Sketch

Enter and upload the following sketch:

```
// Project 56 - Using a Digital Rheostat

1 #include "SPI.h" // necessary library
  int ss=10;        // using digital pin 10 for SPI secondary
  select
  int del=200;      // used for delaying the steps between LED
  brightness values

  void setup()
  {
2    SPI.begin();
      pinMode(ss, OUTPUT);    // we use this for the SS pin
      digitalWrite(ss, HIGH); // the SS pin is active low, so set
  it up high first
3    SPI.setBitOrder(MSBFIRST);
      // our MCP4162 requires data to be sent MSB (most
  significant byte) first
  }

4 void setValue(int value)
  {
      digitalWrite(ss, LOW);
      SPI.transfer(0);    // send the command byte
      SPI.transfer(value); // send the value (0 to 255)
      digitalWrite(ss, HIGH);
  }

  void loop()
  {
5    for (int a=0; a<256; a++)
      {
          setValue(a);
          delay(del);
      }
6    for (int a=255; a>=0; a--)
      {
          setValue(a);
          delay(del);
      }
  }
}
```

Let's walk through the code. First, we set up the SPI bus at 1 and 2. At 3, we set the byte direction to suit the MCP4162. To make setting the resistance easier, we use the custom function at 4, which accepts the resistance step (0 through 255) and passes it to the MCP4162. Finally, the sketch uses two loops to move the rheostat through all the stages, from 0 to the maximum at 5 and then back to 0 at 6. This last piece should make the LED increase and decrease in brightness, fading up and down for as long as the sketch is running.

Looking Ahead

In this chapter, you learned about and experimented with two important Arduino communication methods. Now you're ready to interface your Arduino with a huge variety of sensors, more advanced components, and other items as they become available on the market. One of the most popular components today is a real-time clock IC that allows your projects to keep and work with time—and that's the topic of Chapter 20. So let's go!

<code>// Project 57 - Adding and Displaying Time and Date with an RTC

1
#include "Wire.h"

#define DS3231_I2C_ADDRESS 0x68

// Convert normal decimal numbers to binary coded decimal <span
class="CodeAnnotationHang" aria-label="annotation2">2 byte
decToBcd(byte val) {

 return((val/10*16) + (val%10)); }

// Convert binary coded decimal to normal decimal numbers byte
bcdToDec(byte val)

{

 return((val/16*10) + (val%16)); }

3
void setDS3231time(byte second, byte minute, byte hour, byte dayOfWeek,
byte dayOfMonth, byte month, byte year)

{

 // sets time and date data in the DS3231

 Wire.beginTransmission(DS3231_I2C_ADDRESS); Wire.write(0); // set
next input to start at the seconds register Wire.write(decToBcd(second)); //

```

set seconds Wire.write(decToBcd(minute)); // set minutes
Wire.write(decToBcd(hour)); // set hours
Wire.write(decToBcd(dayOfWeek)); // set day of week (1=Sunday,
7=Saturday) Wire.write(decToBcd(dayOfMonth)); // set date (1 to 31)
Wire.write(decToBcd(month)); // set month Wire.write(decToBcd(year)); //
set year (0 to 99) Wire.endTransmission();

}

```

4
void readDS3231time(byte *second, byte *minute,

byte *hour,

byte *dayOfWeek,

byte *dayOfMonth,

byte *month,

byte *year)

{

Wire.beginTransmission(DS3231_I2C_ADDRESS); Wire.write(0); // set DS3231 register pointer to 00h Wire.endTransmission();

Wire.requestFrom(DS3231_I2C_ADDRESS, 7);

// request seven bytes of data from DS3231 starting from register 00h
*second = bcdToDec(Wire.read() & 0x7f); *minute =
bcdToDec(Wire.read()); <span epub:type="pagebreak" title="354"
id="Page_354"/> *hour = bcdToDec(Wire.read() & 0x3f); *dayOfWeek =
bcdToDec(Wire.read()); *dayOfMonth = bcdToDec(Wire.read()); *month =
bcdToDec(Wire.read()); *year = bcdToDec(Wire.read()); }

```

void displayTime()

{

    byte second, minute, hour, dayOfWeek, dayOfMonth, month, year;

    // retrieve data from DS3231

<span class="CodeAnnotationHang" aria-label="annotation5">5</span>
    readDS3231time(&second, &minute, &hour, &dayOfWeek, &dayOfMonth,
    &month, &year);


    // send it to the Serial Monitor Serial.print(hour, DEC);

    // convert the byte variable to a decimal number when displayed
    Serial.print(":");

    if (minute<10)

    {

        Serial.print("0");

    }

    Serial.print(minute, DEC); Serial.print(":");

    if (second<10)

    {

        Serial.print("0");

    }

    Serial.print(second, DEC); Serial.print(" ");

```



```
Serial.print(dayOfMonth, DEC); Serial.print("/");  
  
Serial.print(month, DEC); Serial.print("/");  
  
Serial.print(year, DEC);  
  
Serial.print(" Day of week: "); switch(dayOfWeek){  
case 1:  
  
Serial.println("Sunday"); break;  
  
case 2:  
  
Serial.println("Monday"); break;  
  
case 3:  
  
Serial.println("Tuesday"); break;  
  
case 4:  
  
Serial.println("Wednesday"); break;  
  
case 5:  
  
Serial.println("Thursday"); break;  
  
case 6:  
  
<span epub:type="pagebreak" title="355" id="Page_355"/>  
Serial.println("Friday"); break;  
  
case 7:  
  
Serial.println("Saturday"); break;  
  
}  
  
}
```

```

void setup()

{

    Wire.begin();

    Serial.begin(9600);


    // set the initial time here: // DS3231 seconds, minutes, hours, day, date,
    month, year <span class="CodeAnnotationHang" aria-
    label="annotation6">6</span> setDS3231time(0, 56, 23, 6, 30, 10, 21);

}

```

```

void loop()

{

    displayTime(); // display the real-time clock data in the Serial Monitor,
    delay(1000); // every second }</code>

```

```

<code>setDS3231time(<var>second</var>, <var>minute</var>,
<var>hour</var>, <var>dayOfWeek</var>, <var>dayOfMonth</var>,
<var>month</var>, <var>year</var>)</code>

```

```

<code>// Project 58 - Creating a Simple Digital Clock

```

```

#include "Wire.h"

```

```

<span class="CodeAnnotationHang" aria-label="annotation1">1</span>
#include <LiquidCrystal.h> #define DS3231_I2C_ADDRESS 0x68

```

```
LiquidCrystal lcd( 8, 9, 4, 5, 6, 7 );
```

```
// Convert normal decimal numbers to binary coded decimal byte  
decToBcd(byte val)
```

```
{
```

```
    return( (val/10*16) + (val%10) ); }
```

```
// Convert binary coded decimal to normal decimal numbers byte  
bcdToDec(byte val)
```

```
{
```

```
    return( (val/16*10) + (val%16) ); }
```

```
void setDS3231time(byte second, byte minute, byte hour, byte dayOfWeek,  
byte dayOfMonth, byte month, byte year) {
```

```
    // sets time and date data in the DS3231
```

```
    Wire.beginTransmission(DS3231_I2C_ADDRESS); Wire.write(0); // set  
next input to start at the seconds register Wire.write(decToBcd(second)); //  
set seconds Wire.write(decToBcd(minute)); // set minutes  
Wire.write(decToBcd(hour)); // set hours  
Wire.write(decToBcd(dayOfWeek)); // set day of week (1=Sunday,  
7=Saturday) Wire.write(decToBcd(dayOfMonth)); // set date (1 to 31)  
Wire.write(decToBcd(month)); // set month Wire.write(decToBcd(year)); //  
set year (0 to 99) Wire.endTransmission();
```

```
}
```

```

void readDS3231time(byte *second,

byte *minute,

byte *hour,

byte *dayOfWeek,

byte *dayOfMonth,

byte *month,

byte *year)

{

    Wire.beginTransmission(DS3231_I2C_ADDRESS); Wire.write(0); // set
DS3231 register pointer to 00h Wire.endTransmission();

<span epub:type="pagebreak" title="358" id="Page_358"/>
    Wire.requestFrom(DS3231_I2C_ADDRESS, 7);

    // request seven bytes of data from DS3231 starting from register 00h
    *second = bcdToDec(Wire.read() & 0x7f); *minute =
bcdToDec(Wire.read()); *hour = bcdToDec(Wire.read() & 0x3f);
    *dayOfWeek = bcdToDec(Wire.read()); *dayOfMonth =
bcdToDec(Wire.read()); *month = bcdToDec(Wire.read()); *year =
bcdToDec(Wire.read()); }

void displayTime()

{

    byte second, minute, hour, dayOfWeek, dayOfMonth, month, year;

    // retrieve data from DS3231

```

```
    readDS3231time(&second, &minute, &hour, &dayOfWeek,  
&dayOfMonth, &month, &year);
```

```
    // send the data to the LCD shield lcd.clear();
```

```
    lcd.setCursor(4,0);
```

```
    lcd.print(hour, DEC);
```

```
    lcd.print(":");
```

```
    if (minute<10)
```

```
    {
```

```
        lcd.print("0");
```

```
    }
```

```
    lcd.print(minute, DEC);
```

```
    lcd.print(":");
```

```
    if (second<10)
```

```
    {
```

```
        lcd.print("0");
```

```
    }
```

```
    lcd.print(second, DEC);
```

```
    lcd.setCursor(0,1);
```

```
    switch(dayOfWeek){
```

case 1:

```
lcd.print("Sun");
```

```
break;
```

case 2:

```
lcd.print("Mon");
```

```
break;
```

case 3:

```
lcd.print("Tue");
```

```
break;
```

case 4:

```
lcd.print("Wed");
```

```
break;
```

case 5:

```
lcd.print("Thu");
```

```
break;
```

case 6:

```
<span epub:type="pagebreak" title="359" id="Page_359"/> lcd.print("Fri");  
break;
```

case 7:

```
lcd.print("Sat");
```

```
break;
```

```

    }

    lcd.print(" ");

    lcd.print(dayOfMonth, DEC); lcd.print("/");

    lcd.print(month, DEC);

    lcd.print("/");

    lcd.print(year, DEC);

}

void setup()

{

    Wire.begin();

    <span class="CodeAnnotationHang" aria-label="annotation2">2</span>
    lcd.begin(16, 2); // set the initial time here: // DS3231 seconds, minutes,
    hours, day, date, month, year <span class="CodeAnnotationHang" aria-
    label="annotation3">3</span> // setDS3231time(0, 27, 0, 5, 15, 11, 20); }

void loop()

{

    displayTime(); // display the real-time clock time on the LCD,
    delay(1000); // every second }</code>

<code>// Project 59 - Creating an RFID Time-Clock System

```

```
<span class="CodeAnnotationHang" aria-label="annotation1">1</span>  
#include "Wire.h" // for RTC
```

```
<span class="CodeAnnotationHang" aria-label="annotation2">2</span>  
#include "SD.h" // for SD card
```

```
#include <LiquidCrystal.h>
```

```
#define DS3231_I2C_ADDRESS 0x68
```

```
LiquidCrystal lcd( 8, 9, 4, 5, 6, 7 );
```

```
int data1 = 0;
```

```
<span class="CodeAnnotationHang" aria-label="annotation3">3</span> //  
Use Listing 18-1 to find your tag numbers int Mary[14] = {
```

```
2, 52, 48, 48, 48, 56, 54, 67, 54, 54, 66, 54, 66, 3}; int John[14] = {
```

```
2, 52, 48, 48, 48, 56, 54, 66, 49, 52, 70, 51, 56, 3}; int newtag[14] = {
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0}; // used for read comparisons
```

```
// Convert normal decimal numbers to binary coded decimal byte  
decToBcd(byte val)
```

```
{
```

```
    return( (val/10*16) + (val%10) ); }
```

```
// Convert binary coded decimal to normal decimal numbers byte  
bcdToDec(byte val)
```



```
{  
  
    return( (val/16*10) + (val%16) ); }
```

```
void setDS3231time(byte second, byte minute, byte hour, byte dayOfWeek,  
byte dayOfMonth, byte month, byte year) {
```

```
    // Sets time and date data in the DS3231
```

```
    Wire.beginTransmission(DS3231_I2C_ADDRESS); Wire.write(0); // set  
next input to start at the seconds register Wire.write(decToBcd(second)); //  
set seconds Wire.write(decToBcd(minute)); // set minutes  
Wire.write(decToBcd(hour)); // set hours  
Wire.write(decToBcd(dayOfWeek)); // set day of week (1=Sunday,  
7=Saturday) Wire.write(decToBcd(dayOfMonth)); // set date (1 to 31)  
Wire.write(decToBcd(month)); // set month Wire.write(decToBcd(year)); //  
set year (0 to 99) Wire.endTransmission();  
  
}
```

```
void readDS3231time(byte *second,
```

```
<span epub:type="pagebreak" title="362" id="Page_362"/>byte *minute,  
byte *hour,
```

```
byte *dayOfWeek,
```

```
byte *dayOfMonth,
```

```
byte *month,
```

```
byte *year)
```

```
{
```

```
Wire.beginTransmission(DS3231_I2C_ADDRESS); Wire.write(0); // set DS3231 register pointer to 00h Wire.endTransmission();
```

```
Wire.requestFrom(DS3231_I2C_ADDRESS, 7);
```

```
// Request seven bytes of data from DS3231 starting from register 00h
*second = bcdToDec(Wire.read() & 0x7f); *minute =
bcdToDec(Wire.read()); *hour = bcdToDec(Wire.read() & 0x3f);
*dayOfWeek = bcdToDec(Wire.read()); *dayOfMonth =
bcdToDec(Wire.read()); *month = bcdToDec(Wire.read()); *year =
bcdToDec(Wire.read()); }
```

```
// Compares two arrays and returns true if identical.
```

```
// This is good for comparing tags.
```

```
boolean comparetag(int aa[14], int bb[14])
```

```
{
```

```
    boolean ff=false;
```

```
    int fg=0;
```

```
    for (int cc=0; cc<14; cc++) {
```

```
        if (aa[cc]==bb[cc])
```

```
        {
```

```
            fg++;
```

```
        }
```

```
    }
```

```
    if (fg==14)
```

```
{  
  
ff=true; // all 14 elements in the array match each other }  
  
return ff;  
}
```

```
void wipeNewTag()
```

```
{  
  
    for (int i=0; i<=14; i++) {  
  
        newtag[i]=0;  
  
    }  
}
```

```
void setup()
```

```
{  
  
    Serial.flush(); // need to flush serial buffer Serial.begin(9600);  
  
<span epub:type="pagebreak" title="363" id="Page_363"/> Wire.begin();  
    lcd.begin(16, 2);  
  
    // set the initial time here: // DS3231 seconds, minutes, hours, day, date,  
    month, year // setDS3231time(0, 27, 0, 5, 15, 11, 12); // Check that the  
    microSD card exists and can be used <span class="CodeAnnotationHang"  
    aria-label="annotation4">4</span> if (!SD.begin(8)) {  
  
        lcd.print("uSD card failure"); // stop the sketch
```

```

    return;

}

lcd.print("uSD card OK"); delay(1000);

lcd.clear();

}

}

void loop()

{

    byte second, minute, hour, dayOfWeek, dayOfMonth, month, year;

    if (Serial.available() > 0) // if a read has been attempted {

        // read the incoming number on serial RX

        delay(100); // allow time for the data to come in from the serial buffer for
(int z=0; z<14; z++) // read the rest of the tag {

            data1=Serial.read();

            newtag[z]=data1;

        }

        Serial.flush(); // stops multiple reads // retrieve data from DS3231

        readDS3231time(&second, &minute, &hour, &dayOfWeek,
&dayOfMonth, &month, &year);

    }

```

```

// now do something based on the tag type <span
class="CodeAnnotationHang" aria-label="annotation5">5</span> if
(comparetag(newtag, Mary) == true) {

    lcd.print("Hello Mary "); File dataFile = SD.open("DATA.TXT",
FILE_WRITE); if (dataFile)

    {

        dataFile.print("Mary ");

        dataFile.print(hour);

        dataFile.print(":");

        if (minute<10) { dataFile.print("0"); }

        dataFile.print(minute);

        dataFile.print(":");

        if (second<10) { dataFile.print("0"); }

        dataFile.print(second);

        dataFile.print(" ");

        dataFile.print(dayOfMonth); <span epub:type="pagebreak" title="364"
id="Page_364"/> dataFile.print("/"); dataFile.print(month);

        dataFile.print("/");

        dataFile.print(year);

        dataFile.println();

        dataFile.close();

```

```
}
```

```
delay(1000);
```

```
lcd.clear();
```

```
wipeNewTag();
```

```
}
```

```
if (compareTag(newtag, John)==true) {
```

```
    lcd.print("Hello John "); File dataFile = SD.open("DATA.TXT",  
    FILE_WRITE); if (dataFile)
```

```
{
```

```
    dataFile.print("John ");
```

```
    dataFile.print(hour);
```

```
    dataFile.print(":");
```

```
    if (minute<10) { dataFile.print("0"); }
```

```
    dataFile.print(minute);
```

```
    dataFile.print(":");
```

```
    if (second<10) { dataFile.print("0"); }
```

```
    dataFile.print(second);
```

```
    dataFile.print(" ");
```

```
    dataFile.print(dayOfMonth); dataFile.print("/");
```

```
    dataFile.print(month);
```

```
dataFile.print("/");  
  
dataFile.print(year);  
  
dataFile.println();  
  
dataFile.close();  
  
}  
  
delay(1000);  
  
lcd.clear();  
  
wipeNewTag();  
  
}  
  
}</code>
```

Understanding the Sketch

In this sketch, the system first waits for an RFID card to be presented to the reader. If the RFID card is recognized, then the card owner's name, the time, and the date are appended to a text file stored on the microSD card.

At 1 are the functions required for the I²C bus and the real-time clock, and at 2 is the line required to set up the microSD card shield. At 4, we check and report on the status of the microSD card. At 5, the card just read is compared against the stored card numbers for two people—in this case, John and Mary. If there is a match, the data is written to the microSD card. With some modification, you could add more cards to the system simply by adding the cards' serial numbers below the existing numbers at 3 and then adding other comparison functions like those at 5.

When the time comes to review the logged data, copy the file *data.txt* from the microSD card. Then view the data with a text editor or import it into a

spreadsheet for further analysis. The data is laid out so that it's easy to read, as shown in [Figure 20-5](#).

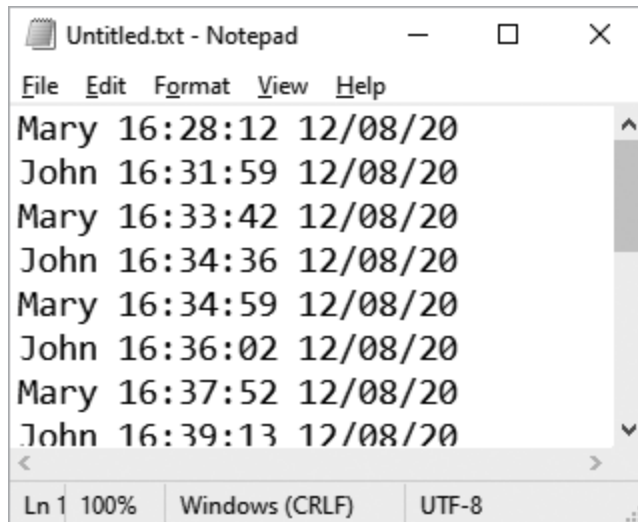


Figure 20-5: Example data generated by Project 59

Looking Ahead

In this chapter, you learned how to work with time and date data via the RTC IC. The RFID system described in Project 59 gives you the framework you need to create your own access systems or even track when, for example, your children arrive home. In the final two chapters, we'll create projects that will use the Arduino to communicate over the internet and a cellular phone network.

21

THE INTERNET

In this chapter you will

Build a web server to display data on a web page

Use your Arduino to send tweets on Twitter

Remotely control Arduino digital outputs from a web browser

This chapter will show you how to connect your Arduino to the outside world via the internet. This allows you to broadcast data from your Arduino and remotely control your Arduino from a web browser.

What You'll Need

To build these internet-related projects, you will need some common hardware, a cable, and some information.

Let's start with the hardware. You'll need an Ethernet shield with the W5100 controller chip. You have two options to consider: you can use the genuine Arduino-brand Ethernet shield, as shown in [Figure 21-1](#), or you can use an Arduino Uno-compatible board with integrated Ethernet hardware, such as PMD Way part 328497, as shown in [Figure 21-2](#). The latter is a good choice for new projects or when you want to save physical space and money. As you can see, the integrated board has the connectors for Arduino shields, a USB port, an Ethernet socket, and a microSD card socket.

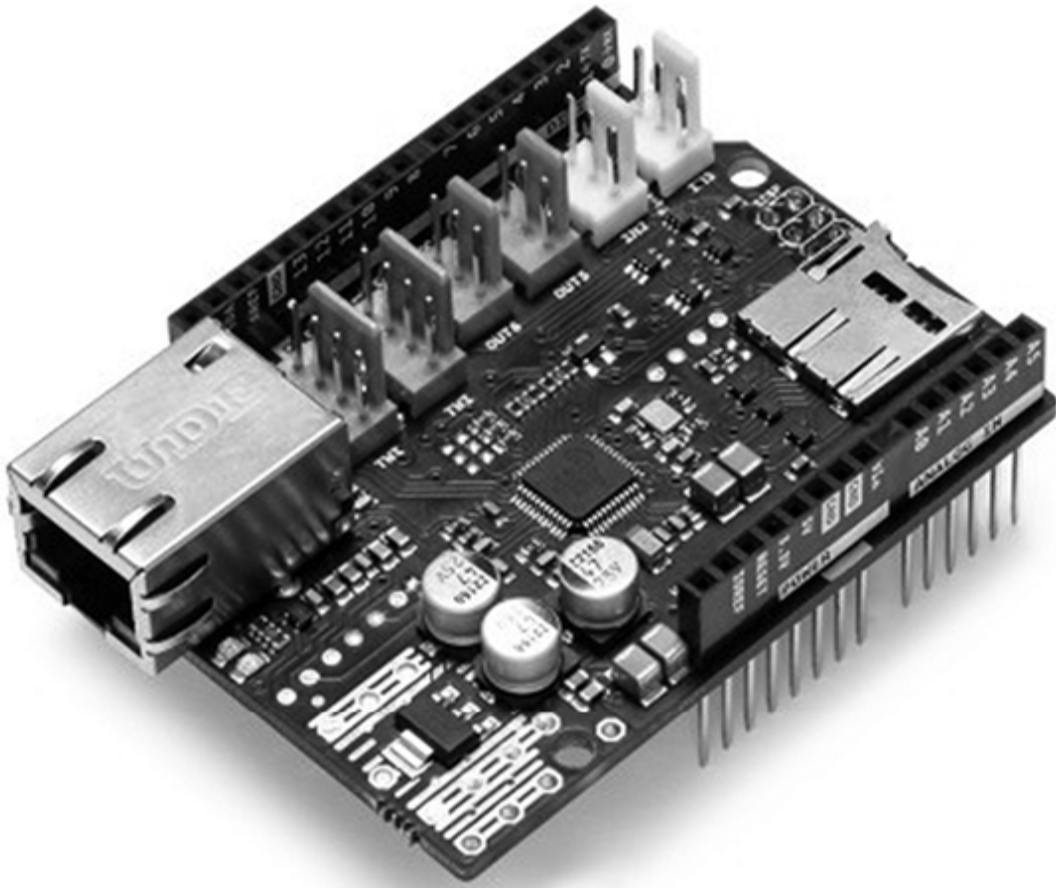


Figure 21-1: An Arduino Ethernet shield

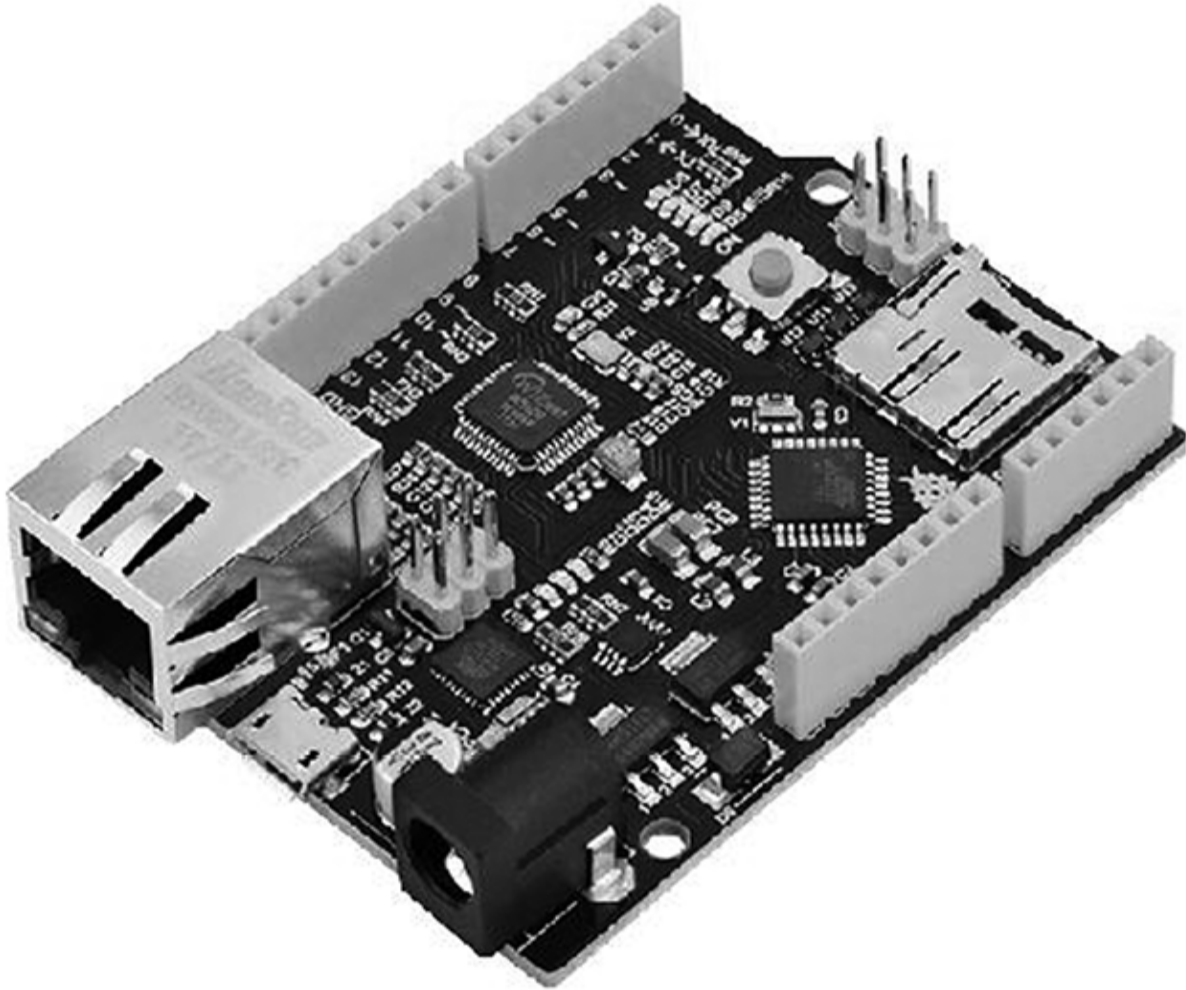


Figure 21-2: An Arduino Uno-compatible board with integrated Ethernet

Regardless of your choice of hardware, you'll also need a standard 10/100 CAT5, CAT5E, or CAT6 network cable to connect your Ethernet shield to your network router or internet modem.

In addition, you'll need the IP address of your network's router gateway or modem, which should look something like this: 192.168.0.1. You'll also need your computer's IP address in the same format as your router's IP address.

Finally, if you want to communicate with your Arduino from outside your home or local area network, you'll need a static, public IP address. A static IP address is a fixed address assigned to your physical internet connection by your internet service provider (ISP). Your internet connection may not

have a static IP address by default; contact your ISP to have this activated if necessary. If your ISP cannot offer a static IP or if it costs too much, you can get an automated redirection service that offers a hostname that can divert to your connection's IP address through a third-party company, such as No-IP (<http://www.noip.com/>) or Dyn (<https://account.dyn.com/>). Now let's put our hardware to the test with a simple project.

Project #60: Building a Remote Monitoring Station

In projects in previous chapters, we gathered data from sensors to measure temperature and light. In this project, you'll learn how to display those values on a simple web page that you can access from almost any web-enabled device. This project will display the values of the analog input pins and the status of digital inputs 0 to 9 on a simple web page, functionality that will serve as the basis for a remote monitoring station.

Using this framework, you can add sensors with analog and digital outputs, such as temperature, light, and switch sensors, and then display the sensors' status on a web page.

The Hardware

Here's what you'll need to create this project:

One USB cable

One network cable

One Arduino Uno and Ethernet shield, or one Arduino Uno-compatible board with integrated Ethernet

The Sketch

Enter the following sketch, but *don't upload it* yet:

```
/* Project 60 - Building a Remote Monitoring Station
   created 18 Dec 2009 by David A. Mellis, modified 9 Apr 2012
   by Tom Igoe
   modified August 2020 by John Boxall
```

```

    */

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

1 IPAddress ip(xxx,xxx,xxx,xxx); // Replace this with your
  project's IP address
2 byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
  EthernetServer server(80);

void setup()
{
  // Start the Ethernet connection and server
  Ethernet.begin(mac, ip);
  server.begin();
  for (int z=0; z<10; z++)
  {
    pinMode(z, INPUT);          // set digital pins 0 to 9 to
inputs
  }
}

void loop()
{
  // listen for incoming clients (incoming web page request
connections)
  EthernetClient client = server.available();
  if (client) {
    // an HTTP request ends with a blank line
    boolean currentLineIsBlank = true;
    while (client.connected()) {
      if (client.available()) {
        char c = client.read();
        if (c == '\n') && currentLineIsBlank) {
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println("Connection: close");
          client.println();
          client.println("<!DOCTYPE HTML>");
          client.println("<html>");
          // add a meta refresh tag, so the browser pulls
again every 5 sec:
3          client.println("<meta http-equiv=\"refresh\"
content=\"5\">");
          // output the value of each analog input pin onto
the web page
          for (int analogChannel = 0; analogChannel < 6;

```

```

    analogChannel++) {
        int sensorReading = analogRead(analogChannel);
4        client.print("analog input ");
        client.print(analogChannel);
        client.print(" is ");
        client.print(sensorReading);
        client.println("<br />");
    }
    // output the value of digital pins 0 to 9 onto the
web page
    for (int digitalChannel = 0; digitalChannel < 10;
digitalChannel++)
    {
        boolean pinStatus = digitalRead(digitalChannel);
        client.print("digital pin ");
        client.print(digitalChannel);
        client.print(" is ");
        client.print(pinStatus);
        client.println("<br />");
    }
    client.println("</html>");
    break;
}
if (c == '\n') {
    // you're starting a new line
    currentLineIsBlank = true;
}
else if (c != '\r') {
    // you've gotten a character on the current line
    currentLineIsBlank = false;
}
}
}
// give the web browser time to receive the data
delay(1);
// close the connection:
client.stop();
}
}

```

We'll discuss this sketch in more detail a bit later. First, before uploading the sketch, you'll need to enter an IP address for your Ethernet shield so that it can be found on your local network or modem. You can determine the first three parts of the address by checking your router's IP address. For example, if your router's address is 192.168.0.1, change the last digit to

something random and different from that of other devices on your network, using a number between 2 and 254 that isn't already in use on your network. Enter the altered IP address at 1 in the sketch, like so:

```
IPAddress ip(192, 168, 0, 69); // Ethernet shield's IP address
```

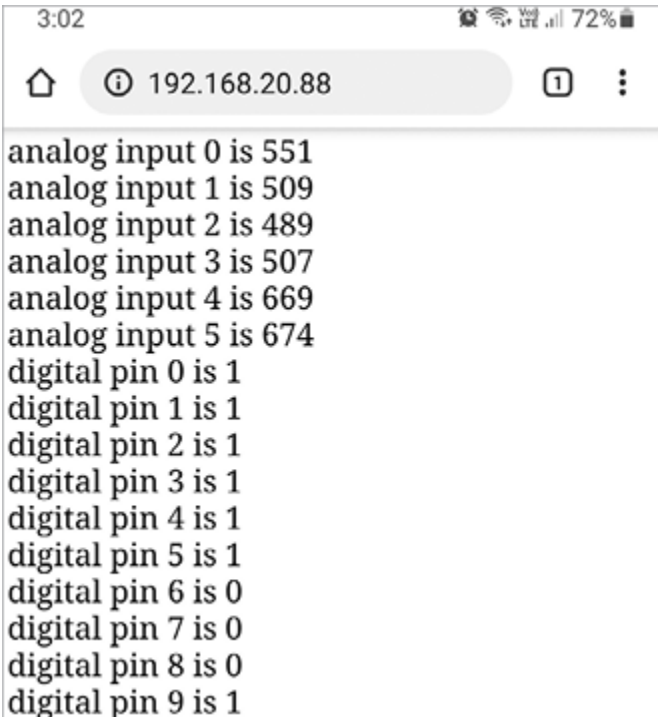


Figure 21-3: Values of the pins monitored by our station, viewable as a web page on any web-connected device with a web browser

Once you've made that change, save and upload your sketch. Next, insert the Ethernet shield into your Arduino if required, connect the network cable to your router or modem and the Ethernet connector, and power on your Arduino board.

Wait about 20 seconds. Then, using a web browser on any device or computer on your network, enter the IP address from 1. If you see something like [Figure 21-3](#), the framework of your monitoring station is working correctly.

Troubleshooting

If this project doesn't work for you, try the following:

Check that the IP address is set correctly in the sketch at 1.

Check that the sketch is correct and uploaded to your Arduino.

Double-check the local network. You might check whether a connected computer can access the internet. If so, check that the Arduino board has power and is connected to the router or modem.

If you're accessing the project web page from a smartphone, make sure your smartphone is accessing your local Wi-Fi network and not the cell phone company's cellular network.

If none of the Ethernet shield's LEDs are blinking when the Arduino has power and the Ethernet cable is connected to the shield and router or modem, try another patch lead.

Understanding the Sketch

Once your monitoring station is working, you can return to the most important parts of the sketch. The code from the beginning until 3 is required because it loads the necessary libraries and starts the Ethernet hardware in `void setup()`. Prior to 3, the `client.print()` statements are where the sketch sets up the web page to allow it to be read by the web browser. From 3 on, you can use the functions `client.print()` and `client.println()` to display information on the web page as you would with the Serial Monitor. For example, the following code is used to display the first six lines of the web page shown in Figure 19-3:

```
client.print("analog input ");  
client.print(analogChannel);  
client.print(" is ");  
client.print(sensorReading);
```

At 4, you see an example of writing text and the contents of a variable to the web page. Here you can use HTML to control the look of your displayed web page, as long as you don't overtax your Arduino's memory. In other words, you can use as much HTML code as you like until you reach the maximum sketch size, which is dictated by the amount of memory

in your Arduino board. (The sizes for each board type are described in Table 13-2 on page 234.)

One thing to notice is the MAC address that networks can use to detect individual pieces of hardware connected to the network. Each piece of hardware on a network has a unique MAC address, which can be changed by altering one of the hexadecimal values at 2. If two or more Arduino-based projects are using one network, you must enter a different MAC address for each device at 2. If your shield has a MAC address included with it, use that value.

Finally, if you want to view your web page from a device that is not connected to your local network, such as a tablet or phone using a cellular connection, then you'll need to use a technique called *port forwarding* in your network router or modem, provided by an organization such as the previously mentioned No-IP or Dyn. Port forwarding is often unique to the make and model of your router, so do an internet search for “router port forwarding” or visit a tutorial site such as <http://www.wikihow.com/Port-Forward> for more information.

Now that you know how to display text and variables on a web page, let's use the Arduino to tweet.

Project #61: Creating an Arduino Tweeter

In this project, you'll learn how to make your Arduino send tweets through Twitter. You can receive all sorts of information that can be generated by a sketch from any device that can access Twitter. If, for example, you want hourly temperature updates from home while you're abroad or even notifications when the kids come home, this can offer an inexpensive solution.

Your Arduino will need its own Twitter account, so do the following:

- Visit <http://twitter.com/> and create your Arduino's Twitter account. Make note of the username and password.

- Get a *token* from the third-party website <http://arduino-tweet.appspot.com/>. A token creates a bridge between your Arduino and the Twitter service.

You'll need to follow only step 1 on this site.

Copy and paste the token, along with your Arduino's new Twitter account details, into a text file on your computer.

Download and install the Twitter Arduino library from

<https://github.com/NeoCat/Arduino-Twitter-library/archive/master.zip>.

The Hardware

Here's what you'll need to create this project:

One USB cable

One network cable

One Arduino Uno and Ethernet shield, or one Arduino Uno-compatible board with integrated Ethernet

The Sketch

Enter the following sketch, but *don't upload it* yet:

```
// Project 61 - Creating an Arduino Tweeter
#include <SPI.h>
#include <Ethernet.h>
#include <Twitter.h>
// Ethernet shield settings
1 IPAddress ip(192,168,0,1); // Replace this with your project's
  IP address
2 byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
3 Twitter twitter("insertyourtokenhere");

  // Message to post
4 char msg[] = "I'm alive!";

void setup()
{
  delay(1000);
  Ethernet.begin(mac, ip);
  // or you can use DHCP for automatic IP address
  configuration
  // Ethernet.begin(mac);
  Serial.begin(9600);
```

```

    Serial.println("connecting ...");
}

void loop()
{
5  if (twitter.post(msg)) {
        int status = twitter.wait(&Serial);
        if (status == 200) {
            Serial.println("OK.");
        } else {
            Serial.print("failed : code ");
            Serial.println(status);
        }
    } else {
        Serial.println("connection failed.");
    }
    while (1);
}

```



Figure 21-4: Your Arduino's tweet

As with Project 60, insert your IP address at 1 and modify the MAC address if necessary at 2. Then insert the Twitter token between the double quotes at 3. Finally, insert the text that you want to tweet at 4. Now upload the sketch and connect your hardware to the network. (Don't forget to follow your Arduino's Twitter account with your own account!) After a minute or so, visit your Twitter page or load the app on a device, and the message should be displayed, as shown in [Figure 21-4](#).

When you're creating your Arduino tweeter, keep in mind that you can send no more than one tweet per minute and that each message must be unique. (These are Twitter's rules.) When sending tweets, Twitter also replies with a status code. The sketch will receive and display this in the Serial Monitor using the code at 5. [Figure 21-5](#) shows an example.

```
connecting ...
HTTP/1.1 403 Forbidden
Content-Type: text/html; charset=utf-8
Cache-Control: no-cache
X-Cloud-Trace-Context: 31e34e66928b59e94cda735e0944674d;o=1
Date: Thu, 13 Aug 2020 06:02:00 GMT
Server: Google Frontend
Content-Length: 73
Connection: close

Error 403 - {"errors":[{"code":187,"message":"Status is a duplicate."}]}
```

failed : code 403

Figure 21-5: Example error message from Twitter due to a duplicate post attempt

If you receive a 403 message like this, either your token is incorrect or you're sending tweets too quickly. (For a complete list of Twitter error codes, see <https://finderrorcode.com/twitter-error-codes.html>.)

Controlling Your Arduino from the Web

You can control your Arduino from a web browser in several ways. After doing some research, I've found a method that is reliable, secure, and free: Teleduino.

Teleduino is a free service created by New Zealand Arduino enthusiast Nathan Kennedy. It's a simple yet powerful tool for interacting with an Arduino over the internet. It doesn't require any special or customized Arduino sketches; instead, you simply enter a special URL into a web browser to control the Arduino. You can use Teleduino to control digital output pins and servos or to send I²C commands, and more features are being added all the time. In Project 62, you'll learn how to configure Teleduino and remotely control digital outputs from a web-enabled device.

Project #62: Setting Up a Remote Control for Your Arduino

Before starting your first Teleduino project, you must register with the Teleduino service and obtain a unique key to identify your Arduino. To do so, visit <https://www.teleduino.org/tools/request-key/> and enter the required information. You should receive an email with your key, which will look something like this: 187654321Z9AEFF952ABCDEF8534B2BBF.

Next, convert your key into an array variable by visiting <https://www.teleduino.org/tools/arduino-sketch-key/>. Enter your key, and the page should return an array similar to that shown in [Figure 21-6](#).

```
byte key[] = { 0x65, 0x9A, 0xCE, 0xB1,  
               0xA7, 0x5E, 0x57, 0x2B,  
               0x8F, 0xFE, 0xA4, 0x40,  
               0x9E, 0x7B, 0x7A, 0xBC };
```

Figure 21-6: A Teleduino key as an array

Each key is unique to a single Arduino, but you can get more keys if you want to run more than one Teleduino project at a time.

The Hardware

Here's what you'll need to create this project:

One USB cable

One network cable

One Arduino Uno and Ethernet shield, or one Arduino Uno-compatible board with integrated Ethernet

One 560 Ω resistor (R1)

One breadboard

One LED of any color

Assemble your hardware and connect an LED to digital pin 8, as shown in [Figure 21-7](#).

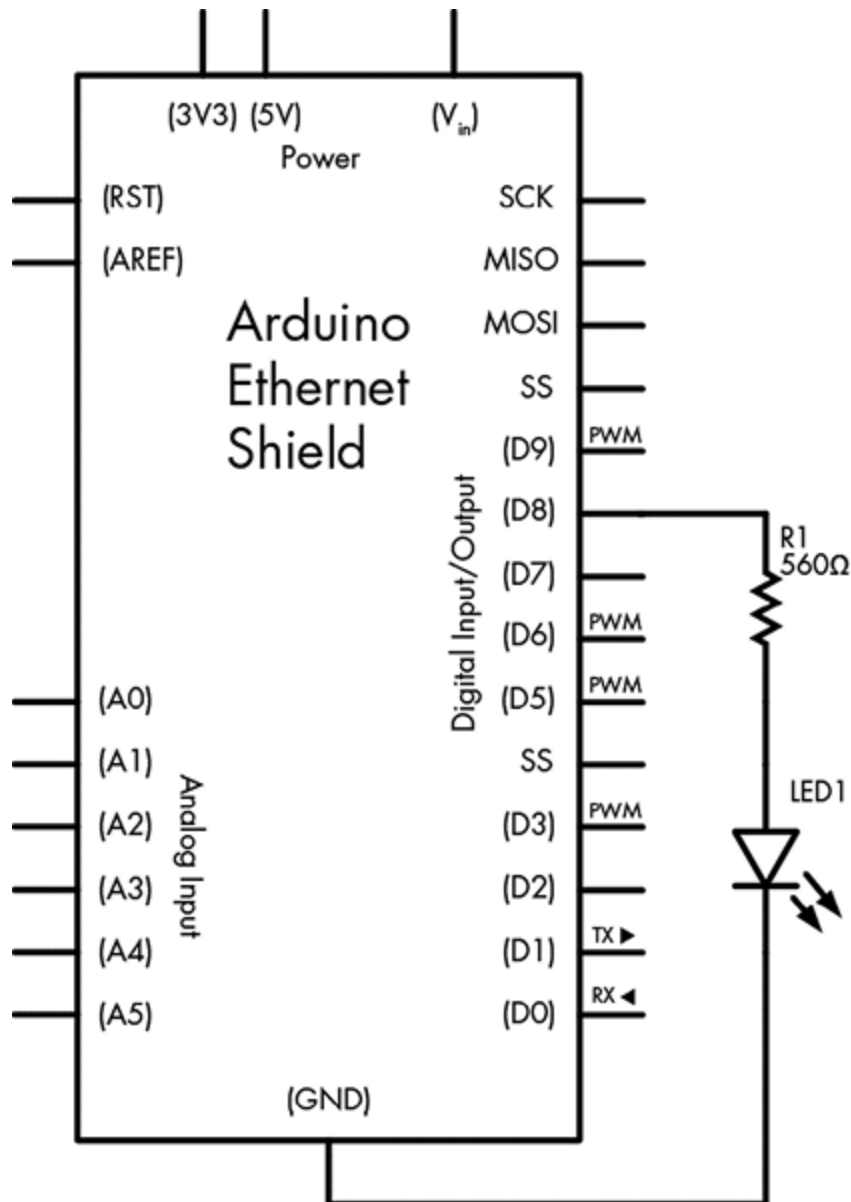


Figure 21-7: Schematic for Project 62

The Sketch

Projects in Teleduino use only one sketch, which is included with the Teleduino library. Here's how to access the sketch:

Download and install the Teleduino library from

<https://www.teleduino.org/downloads/>.

Restart the Arduino IDE and select

File►Examples►Teleduino328►TeleduinoEthernetClientProxy.

You should now see the Teleduino sketch. Before uploading it to your Arduino, replace the default key with your key array. The variable you need to replace should be on line 36 of the sketch. Once you've replaced it, save the sketch, and then upload it to your Arduino.

Now connect your hardware to the network and watch the LED. After a minute or so, it should blink a few times and then rest. The number of blinks represents the status of the Teleduino service, as shown in [Table 21-1](#).

Table 21-1: *Teleduino Status Blink Codes*

Number of blinks	Message
1	Initializing
2	Starting network connection
3	Connecting to the Teleduino server
4	Authentication successful
5	Session already exists
6	Invalid or unauthorized key
10	Connection dropped

If you see five blinks, then another Arduino is already programmed with your key and connected to the Teleduino server. At 10 blinks, you should check your hardware and internet connections. Once the Arduino has connected, it should blink once every 5 seconds or so. Because the status LED is controlled by digital pin 8, you can't use that pin for any other purpose while you're using Teleduino.

Controlling Your Arduino Remotely

To control your Teleduino remotely, you can use any device with a web browser. However, you first need to set the mode for each digital pin you wish to control. The command to control the Arduino is sent by entering a URL that you create:

```
http://us01.proxy.teleduino.org/api/1.0/328.php?k={YOURKEY}&r=definePinMode&pin=<X>&mode=<Y>
```

You'll need to change three parameters in the URL. First, replace {YOURKEY} with the long alphanumeric key you received from the Teleduino site. Next, replace <X> with the digital pin number you want to control. Third, change the <Y> to 1 to set up the digital pin as an output.

Now you can control the digital pin remotely. The command to do this is:

```
http://us01.proxy.teleduino.org/api/1.0/328.php?k={YOURKEY}&r=setDigitalOutput&pin=<X>&output=<S>
```

Again, you'll need to change three parameters in the URL. First, replace {YOURKEY} with the long alphanumeric key you received from the Teleduino site. Next, replace <X> with the digital pin number you want to control. Third, change the <S> to 0 for low or 1 for high to alter the digital output. For example, to turn digital pin 7 to high, you would enter:

```
http://us01.proxy.teleduino.org/api/1.0/328.php?k={YOURKEY}&r=setDigitalOutput&pin=7&output=1
```

After the command succeeds, you should see something like the following in your web browser:

```
{"status":200,"message":"OK","response"
{"result":0,"time":0.22814512252808,"values":[]}}
```

If the command fails, you should see an error message like this:

```
{"status":403,"message":"Key is offline or
invalid.,"response":[]}
```

You can send commands to change the digital pins to high or low by modifying the URL.

If a digital pin is capable of pulse-width modulation (PWM), as described in Chapter 3, you can also control the PWM output from a pin using:

```
http://us01.proxy.teleduino.org/api/1.0/328.php?k=
{YOURKEY}&r=setPwmOutput&pin=<X>&output=<Y>
```

where <x> is the digital output pin and <y> is the PWM level, between 0 and 255.

After you have created the URLs for your project, bookmark them in your browser or create a local web page with the required links as buttons. For example, you might have a URL bookmarked to set digital pin 7 to high and another bookmarked to set it back to low.

In some situations, the status of your Arduino outputs could be critical. As a fail-safe in case your Arduino resets itself due to a power outage or other interruption, set the default state for the digital pins. With your project connected to the Teleduino service, visit

<https://www.teleduino.org/tools/manage-presets/>. After entering your unique key, you should see a screen of options that allows you to select the mode and value for the digital pins, as shown in [Figure 21-8](#).

Pin	Mode	Value	Pin	Mode	Value
0	Unset	Unset	11	Unset	Unset
1	Unset	Unset	12	Unset	Unset
2	1 - output	0	13	Unset	Unset
3	1 - output	0	14	Unset	Unset
4	Unset	Unset	15	Unset	Unset
5	1 - output	0	16	Unset	Unset
6	1 - output	0	17	Unset	Unset
7	Unset	Unset	18	Unset	Unset
8	Unset	Unset	19	Unset	Unset
9	Unset	Unset	20	Unset	Unset
10	Unset	Unset	21	Unset	Unset

Figure 21-8: Default pin status setup page

Looking Ahead

Along with easily monitoring your Arduino over the internet and having it send tweets on Twitter, you can control your Arduino projects over the internet without creating any complex sketches, having much networking knowledge, or incurring monthly expenses. This enables you to control the Arduino from almost anywhere and extend the reach of its ability to send data. The three projects in this chapter provide a framework that you can build upon to design your own remote control projects.

The next chapter, which is the last one in the book, shows you how to make your Arduino send and receive commands over a cellular network connection.

22

CELLULAR COMMUNICATIONS

In this chapter you will

Have your Arduino dial a telephone number when an event occurs

Send a text message to a cell phone using the Arduino

Control devices connected to an Arduino via text message from a cell phone

You can connect your Arduino projects to a cell phone network to allow simple communication between your Arduino and a cellular or landline phone. With a little imagination, you can come up with many uses for this type of communication, including some of the projects included in this chapter.

Be sure to review this chapter before you purchase any hardware, because the success of the projects will depend on your cellular network. Your network must be able to do the following:

Operate at UMTS (3G) 850 MHz, 900 MHz, 1900 MHz, or 2100 MHz.

Allow the use of devices not supplied by the network provider.

To make use of these projects, you might consider selecting either a prepaid calling plan or a plan that offers a lot of included text messages, in case an error in your sketch causes the project to send out several SMS (Short Message Service) text messages. Also, make sure the requirement to enter a PIN to use the SIM card is turned off. (You should be able to do this easily by inserting the SIM card in a regular cell phone and changing the setting in the security menu.)

The Hardware

All the projects use a common hardware configuration, so we'll set that up first. You'll need specific hardware to complete the projects in this chapter, starting with a SIM5320-type 3G GSM shield and antenna, shown in [Figure 22-1](#). This shield is available from TinySine (<https://www.tinyosshop.com/>) and its distributors. There are two types of SIM5320 shield: the SIM5320A and SIM5320E.

The -E version uses the UMTS/HSDPA 900/2100 MHz frequency bands (mainly for European users), and the -A version uses the UMTS/HSDPA 850/1900 MHz frequency band (mainly for US-based users and Australians using the Telstra network).



Figure 22-1: 3G shield with antenna attached

You'll also need a power supply. In some situations, the 3G shield can draw up to 2 A of current (more than is available from the Arduino) and will

damage your Arduino if it's used without external power. Therefore, you will need an external power supply. This can be a DC plug pack or wall wart power supply brick (or a large 7.2 V rechargeable battery, solar panel/battery source, 12 V battery, or similar, as long as it doesn't exceed 12 V DC) that can offer up to 2 A of current.

Hardware Configuration and Testing

Now let's configure and test the hardware by making sure that the 3G shield can communicate with the cellular network and the Arduino. We first need to set up the serial communication jumpers, since the 3G shield communicates with the Arduino via a serial port in the same manner as the GPS modules used in Chapter 15. We can set which digital pins the shield will use to communicate with the Arduino using jumpers on the top right of the shield. All our projects will use digital pin 2 for shield transmit and digital pin 3 for shield receive. To configure this, connect jumpers over the TX2 and RX3 pins, as shown in [Figure 22-2](#).

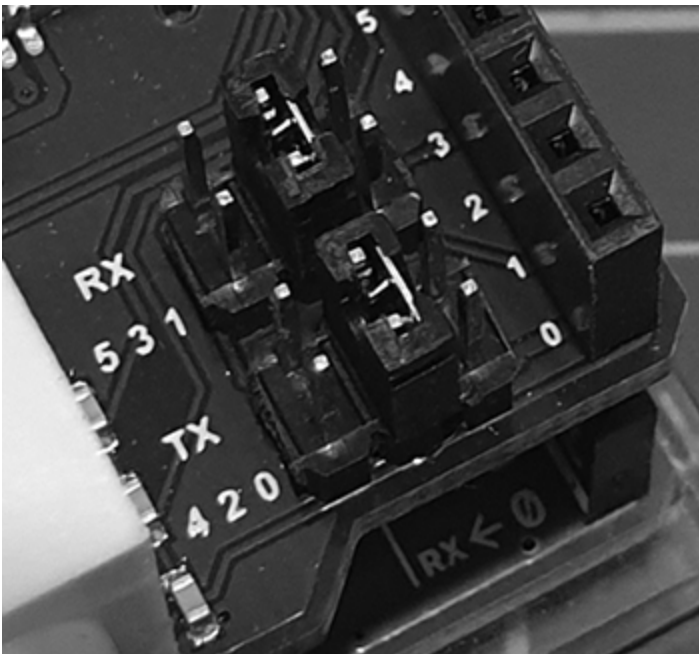


Figure 22-2: Shield serial configuration jumpers

Next, turn the shield over and insert your SIM card into the holder, as shown in [Figure 22-3](#).

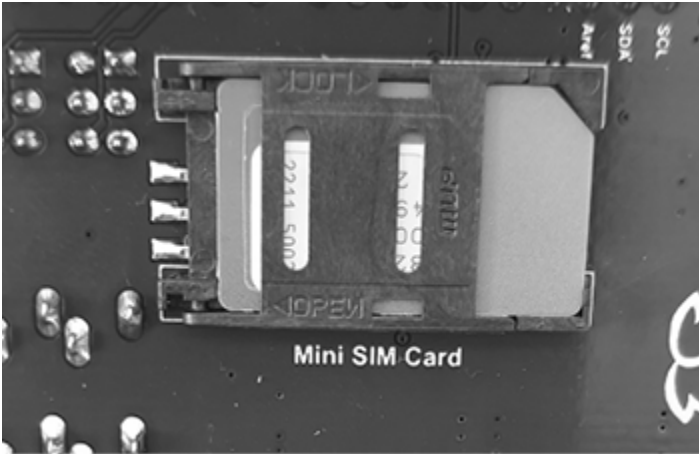


Figure 22-3: SIM card in place

Next, gently insert the 3G shield into the Arduino. Connect the external power and the USB cable between the Arduino and the PC. Finally, just as with a cellular phone, you need to turn the SIM module on (and off) using the power button on the top-left corner of the shield, as shown in [Figure 22-4](#). Press the button for 2 seconds and let go. After a moment, the P (for power) and S (for status) LEDs will come on, and the blue LED will start blinking once the 3G shield has registered with the cellular network.

For future reference, the shield's power button is connected to digital pin 8, so you can control the power from your sketch instead of manually turning the button on or off.

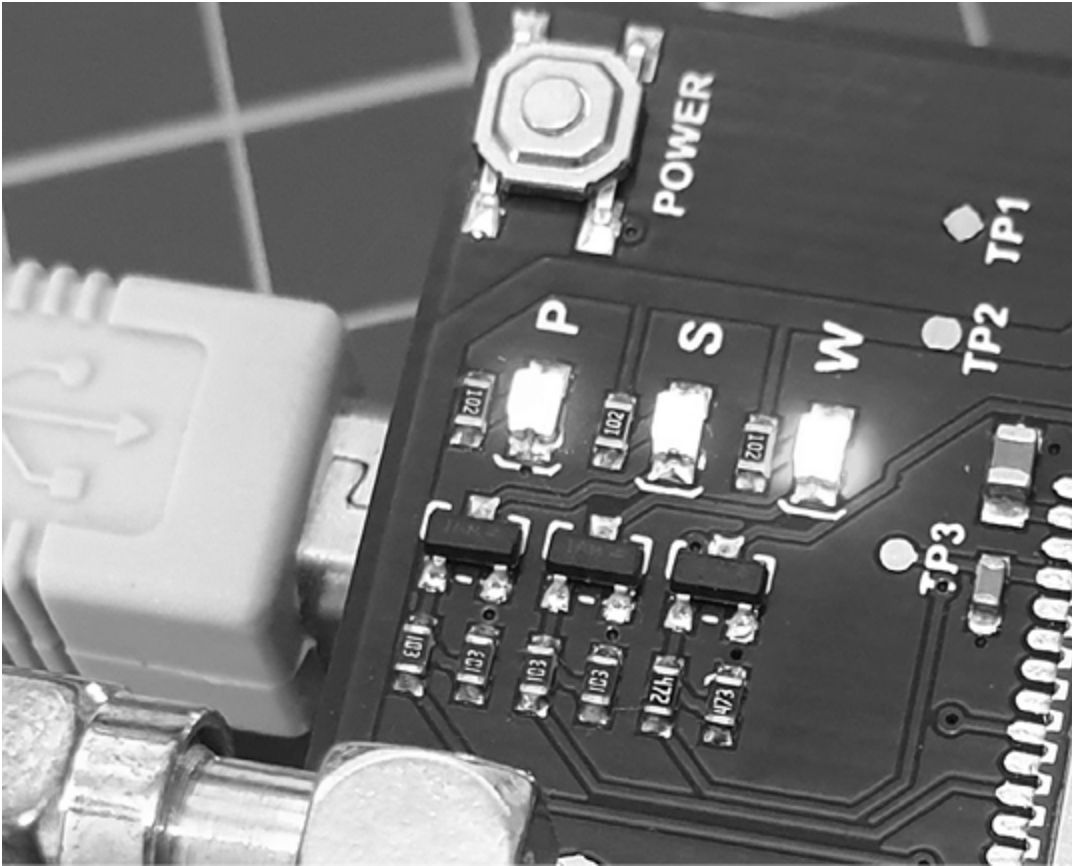


Figure 22-4: 3G shield power button and status LEDs

Now enter and upload the sketch shown in [Listing 22-1](#).

```
// Listing 22-1

1 #include <SoftwareSerial.h> // Virtual serial port
2 SoftwareSerial cell(2,3);
  char incoming_char = 0;

  void setup()
  {
    // Initialize serial ports for communication
    Serial.begin(9600);
3   cell.begin(4800);
    Serial.println("Starting SIM5320 communication...");
  }

  void loop()
  {
    // If a character comes in from 3G shield
```

```

if( cell.available() > 0 )
{
    // Get the character from the cellular serial port
    incoming_char = cell.read();
    // Print the incoming character to the Serial Monitor
    Serial.print(incoming_char);
}
// If a character is coming from the terminal to the
Arduino...
if( Serial.available() > 0 )
{
    incoming_char = Serial.read(); // Get the character from
the terminal
    cell.print(incoming_char); // Send the character to the
cellular module
}
}

```

Listing 22-1: 3G shield test sketch

This sketch simply relays all the information coming from the 3G shield to the Serial Monitor. The 3G shield has a software serial connection between it and Arduino digital pins 2 and 3 so that it won't interfere with the normal serial connection between the Arduino and the PC, which is on digital pins 0 and 1. We set up a virtual serial port for the 3G shield at 1, 2, and 3. By default, the 3G shield communicates over serial at 4,800 bps, and this is fine for our projects.

Once you've uploaded the sketch, open the Serial Monitor window and wait about 10 seconds. Then, using a different telephone, call the number for your 3G shield. You should see data similar to that shown in [Figure 22-5](#).


```
Starting SIM5320 communication...
```

```
RING
```

```
RING
```

```
RING
```

```
RING
```

```
RING
```

```
MISSED_CALL: 00:05AM 042[REDACTED]
```

Figure 22-5: Example output from [Listing 22-1](#)

The RING notifications come from the shield when you are calling it, and the missed call notification shows up when you end the call to the shield. If your cellular network supports caller ID, the originating phone number is shown after the time. (The number has been blacked out in [Figure 22-5](#) for the sake of privacy.) Now that the 3G shield is operating, we can make use of various functions for our projects.

Project #63: Building an Arduino Dialer

By the end of this project, your Arduino will dial a telephone number when an event occurs, as determined by your Arduino sketch. For example, if the temperature in your storage freezer rises above a certain level or your burglar alarm system activates, you could have the Arduino call you from a preset number, wait for 20 seconds, and then hang up. Your phone's caller ID will identify the phone number as the Arduino.

The Hardware

This project uses the hardware described at the beginning of the chapter as well as any extra circuitry you choose for your application. For demonstration purposes, we'll use a button to trigger the call.

In addition to the hardware already discussed, here's what you'll need to create this project:

One push button

One 10 k Ω resistor

One 100 nF capacitor

Various connecting wires

One breadboard

The Schematic

Connect the external circuitry, as shown in [Figure 22-6](#).

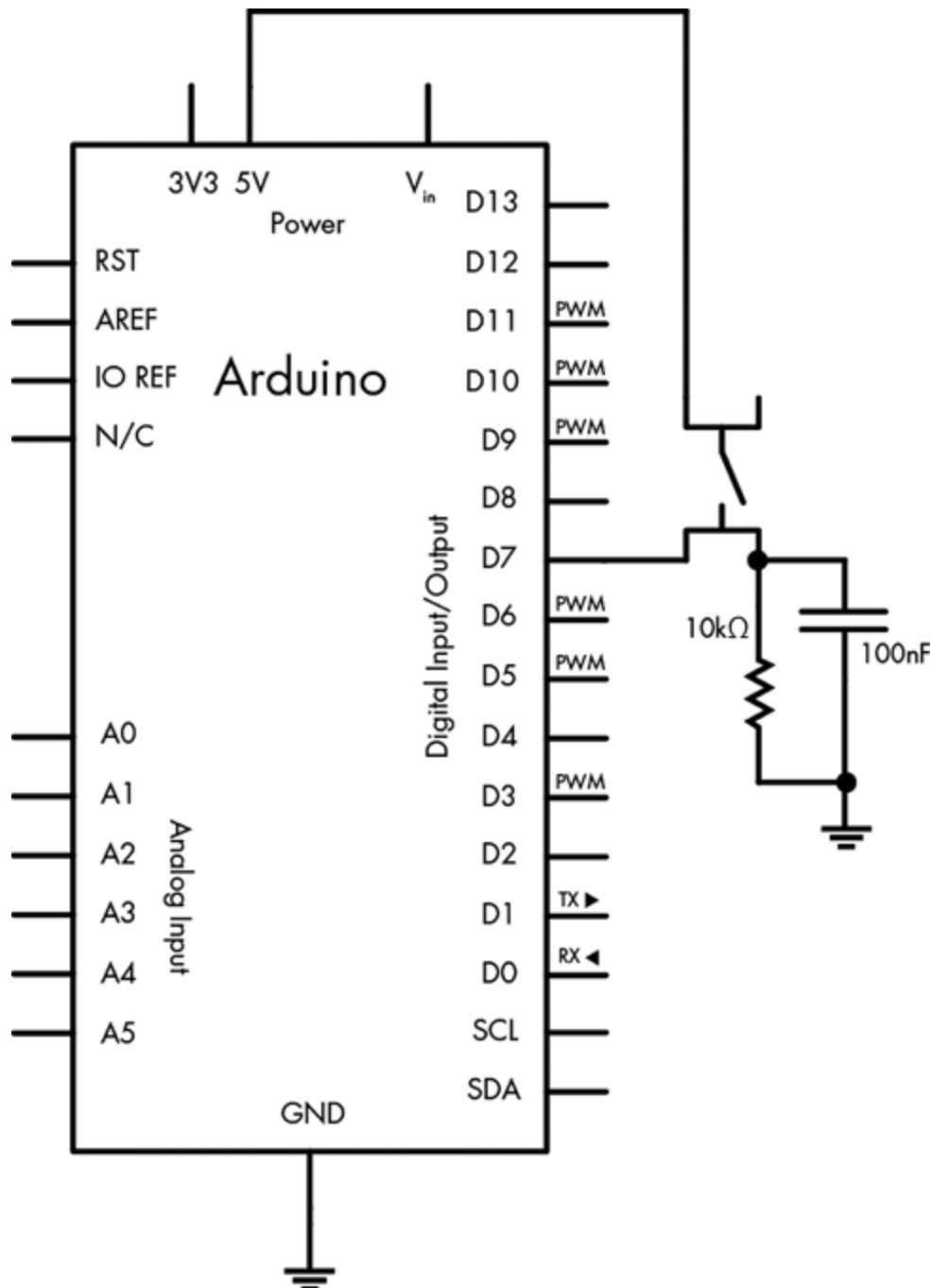


Figure 22-6: Schematic for Project 63

The Sketch

Enter *but don't upload* the following sketch:

```
// Project 63 - Building an Arduino Dialer
```

```

#include <SoftwareSerial.h> // Virtual serial port
SoftwareSerial cell(2,3);
char incoming_char = 0;

void setup()
{
  pinMode(7, INPUT);          // for button
  pinMode(8, OUTPUT);         // shield power control
  // initialize serial ports for communication
  Serial.begin(9600);
  cell.begin(4800);
}

void callSomeone()
{
  // turn shield on
1  Serial.println("Turning shield power on...");
   digitalWrite(8, HIGH);
   delay(2000);
   digitalWrite(8, LOW);
   delay(10000);
2  cell.println("ATDxxxxxxxxxx"); // dial the phone number
   xxxxxxxxxxxx
   // change xxxxxxxxxxxx to your desired phone number (with
   area code)
   Serial.println("Calling ...");
   delay(20000);                // wait 20 seconds
3  cell.println("ATH");          // end call
   Serial.println("Ending call, shield power off.");
   // turn shield off to conserve power
4  digitalWrite(8, HIGH);
   delay(2000);
   digitalWrite(8, LOW);
}
void loop()
{
5  if (digitalRead(7) == HIGH)
   {
6    callSomeone();
   }
}

```

Understanding the Sketch

After setting up the software serial and regular serial ports, the sketch waits for a press of the button connected to digital pin 7 at 5. Once it's pressed, the function `callSomeone()` is run at 6. At 1, digital pin 8 is toggled HIGH for 2 seconds, turning the shield on, and waits 10 seconds to give the shield time to register with the cellular network. Next, at 2, the sketch sends the command to dial a telephone number. Finally, after the call has been ended at 3, the shield is turned off to conserve power at 4.

You'll replace `xxxxxxxxxx` with the number you want your Arduino to call. Use the same dialing method that you'd use from your mobile phone. For example, if you wanted the Arduino to call 212.555.1212, you'd add this:

```
cell.println("ATD2125551212");
```

After you have entered the phone number, you can upload the sketch, wait a minute to allow time for the 3G module to connect to the network, and then test it by pressing the button. It's very easy to integrate the dialing function into an existing sketch, because it's simply called when required at 2. From here, it's up to you to find a reason—possibly triggered by a temperature sensor, a light sensor, or any other input reaching a certain level—for your Arduino to dial a phone number.

Now let's drag your Arduino into the 21st century by sending a text message.

Project #64: Building an Arduino Texter

In this project, the Arduino will send a text message to another cell phone when an event occurs. To simplify the code, we'll use the SerialGSM Arduino library, available from <https://github.com/meirm/SerialGSM/archive/master.zip>. After you've installed the library, restart the Arduino IDE.

The hardware you'll need for this project is identical to that for Project 63.

The Sketch

Enter the following sketch into the Arduino IDE, but *don't upload it yet*:

```
// Project 64 - Building an Arduino Texter
#include <SerialGSM.h>
#include <SoftwareSerial.h> // Virtual serial port
1 SerialGSM cell(2, 3);

void sendSMS()
{
2   cell.Message("The button has been pressed!");
   cell.SendSMS();
}

void setup()
{
   pinMode(7, INPUT); // for button
   pinMode(8, OUTPUT); // shield power control
   // turn shield on
   Serial.println("Turning shield power on...");
   digitalWrite(8, HIGH);
   delay(2000);
   digitalWrite(8, LOW);
   // initialize serial ports for communication
   Serial.begin(9600);
   cell.begin(4800);
   cell.Verbose(true);
   cell.Boot();
   cell.FwdSMS2Serial();
3   cell.Rcpt("xxxxxxxxxxxx");
   delay(10000);
}

void loop()
{
4   if (digitalRead(7) == HIGH)
       {
           sendSMS();
       }
}
```

Understanding the Sketch

The 3G shield is set up as normal at 1 and in void setup(). Button presses are detected at 4, and the function sendSMS() is called. This simple function sends a text message to the cell phone number stored at 3.

Before uploading the sketch, replace xxxxxxxxxxxx with the recipient's cell phone number; enter the area code plus number, without any spaces or brackets. For example, to send a text to 212.555.1212 in the United States, you would store 2125551212.

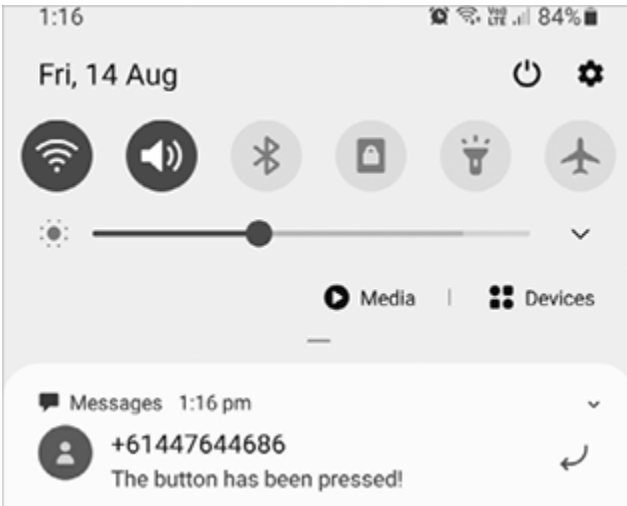


Figure 22-7: A sample text message being received

The text message to be sent is stored at 2. (Note that the maximum length for a message is 160 characters.)

After you have stored a sample text message and a destination number, upload the sketch, wait 30 seconds, and then press the button. In a moment, the message should arrive on the destination phone, as shown in [Figure 22-7](#).

Project 64 can be integrated quite easily into other sketches, and various text messages could be sent by comparing data against a parameter with a switch case statement.

NOTE

Remember that the cost of text messages can add up quickly, so when you're experimenting, be sure that you're using an unlimited or prepaid calling plan.

Project #65: Setting Up an SMS Remote Control

In this project, you'll control the digital output pins on your Arduino by sending a text message from your cell phone. You should be able to use your existing knowledge to add various devices to control. We'll allow for four separate digital outputs, but you can control more or fewer as required.

To turn on or off four digital outputs (pins 10 through 13 in this example), you'd send a text message to your Arduino in the following format: #axbxcxdx, replacing x with either a 0 for off or a 1 for on. For example, to turn on all four outputs, you'd send #a1b1c1d1.

The Hardware

This project uses the hardware described at the start of the chapter, plus any extra circuitry you choose. We'll use four LEDs to indicate the status of the digital outputs being controlled. Therefore, the following extra hardware is required for this example:

Four LEDs

Four 560 Ω resistors

Various connecting wires

One breadboard

The Schematic

Connect the external circuitry as shown in [Figure 22-8](#).

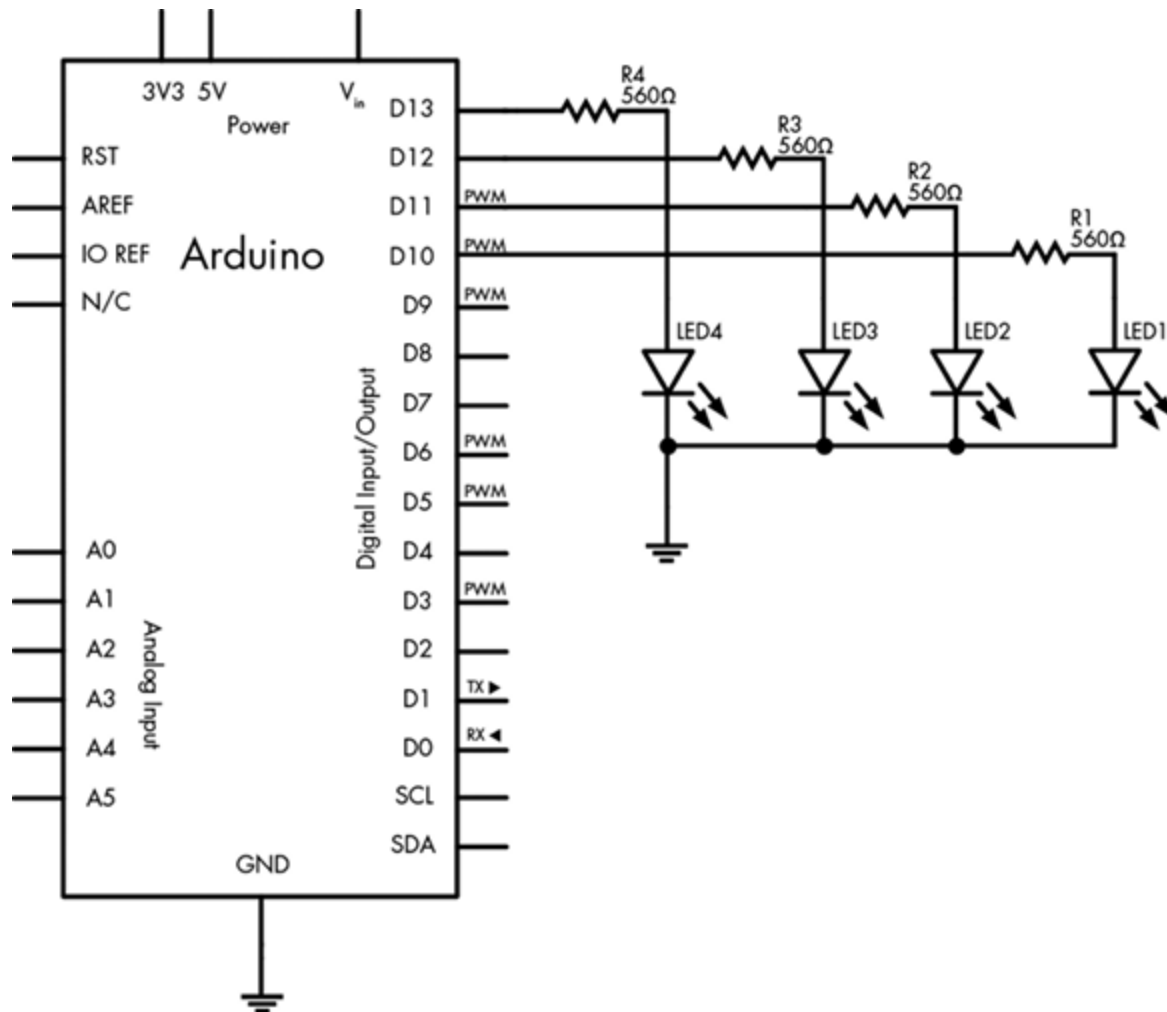


Figure 22-8: Schematic for Project 65

The Sketch

For this project, the 3G shield library is not used. Instead, we rely on the raw commands to control the module. Furthermore, we don't turn the shield on or off during the sketch, as we need it to be on in order to listen for incoming text messages. Enter and upload the following sketch:

```
// Project 65 - Setting Up an SMS Remote Control
#include <SoftwareSerial.h>
SoftwareSerial cell(2,3);
char inchar;

void setup()
{
    // set up digital output pins to control
```

```

    pinMode(10, OUTPUT);
    pinMode(11, OUTPUT);
    pinMode(12, OUTPUT);
    pinMode(13, OUTPUT);
    digitalWrite(10, LOW); // default state for I/O pins at
power-up or reset,
    digitalWrite(11, LOW); // change as you wish
    digitalWrite(12, LOW);
    digitalWrite(13, LOW);
    // initialize the 3G shield serial port for communication
    cell.begin(4800);
    delay(30000);
1  cell.println("AT+CMGF=1");
    delay(200);
2  cell.println("AT+CNMI=3,3,0,0");
    delay(200);
}

void loop()
{
    // if a character comes in from the cellular module...
3  if(cell.available() > 0)
    {
        inchar = cell.read();
4      if (inchar == '#') // the start of our command
        {
            delay(10);
            inchar = cell.read();
5          if (inchar == 'a')
            {
                delay(10);
                inchar = cell.read();
                if (inchar == '0')
                {
                    digitalWrite(10, LOW);
                }
                else if (inchar == '1')
                {
                    digitalWrite(10, HIGH);
                }
                delay(10);
                inchar = cell.read();
                if (inchar == 'b')
                {
                    inchar = cell.read();
                    if (inchar == '0')

```

```
{
    digitalWrite(11, LOW);
}
else if (inchar == '1')
{
    digitalWrite(11, HIGH);
}
delay(10);
inchar = cell.read();
if (inchar == 'c')
{
    inchar = cell.read();
    if (inchar == '0')
    {
        digitalWrite(12, LOW);
    }
    else if (inchar == '1')
    {
        digitalWrite(12, HIGH);
    }
    delay(10);
    inchar = cell.read();
    if (inchar == 'd')
    {
        delay(10);
        inchar = cell.read();
        if (inchar == '0')
        {
            digitalWrite(13, LOW);
        }
        else if (inchar == '1')
        {
            digitalWrite(13, HIGH);
        }
        delay(10);
    }
}
cell.println("AT+CMGD=1,4"); // delete all SMS
}
```

Understanding the Sketch

In this project, the Arduino monitors every text character sent from the 3G shield. Thus, at 1, we tell the shield to convert incoming SMS messages to text and send the contents to the virtual serial port at 2. Next, the Arduino waits for a text message to come from the shield at 3.

Because the commands sent from the cell phone and passed by the 3G module to control pins on the Arduino start with a #, the sketch waits for a hash mark (#) to appear in the text message at 4. At 5, the first output parameter *a* is checked—if it is followed by a 0 or 1, the pin is turned off or on, respectively. The process repeats for the next three outputs controlled by *b*, *c*, and *d*.

Fire up your imagination to think of how easy it would be to use this project to create a remote control for all manner of things—lights, pumps, alarms, and more.

Looking Ahead

With the three projects in this chapter, you’ve created a great framework on which to build your own projects that can communicate over a cell network. You’re limited only by your imagination—for example, you could receive a text message if your basement floods or turn on your air conditioner from your cell phone. Once again, remember to take heed of network charges before setting your projects free.

At this point, after having read about (and hopefully built) the 65 projects in this book, you should have the understanding, knowledge, and confidence you need to create your own Arduino-based projects. You know the basic building blocks used to create many projects, and I’m sure you will be able to apply the technology to solve all sorts of problems and have fun at the same time.

I’m always happy to receive feedback about this book, which can be left via the contact details at the book’s web page: <https://nostarch.com/arduino-workshop-2nd-edition/>.

But remember—this is only the beginning. You can find many more forms of hardware to work with, and with some thought and planning, you can work with them all. You’ll find a huge community of Arduino users on the

internet (in such places as the Arduino forum at <http://forum.arduino.cc/>), and even at a local hackerspace or club.

So don't just sit there—make something!

Index

Please note that index links to approximate location of each term.

Symbols

;
// (comment), [17](#)
|| (or) operator, [63](#)
/* */ (comment), [17](#)
== (equal) operator, [61](#)
> (greater than) operator, [73](#)
>= (greater than or equal to) operator, [73](#)
< (less than) operator, [73](#)
<= (less than or equal to) operator, [73](#)
= (make equal to) operator, [61](#)
!= (not equal) operator, [61](#)
! (not) operator, [62](#)
~ (tilde) pins, [12](#), [38](#)
&& (and) operator, [62–63](#)
μF (microfarads), [51](#)
Ω (ohms), [26–27](#)

A

A (amperes), [25](#)

AC (alternating current), [25](#)

Adafruit Industries

ordering from, [5](#), [227](#), [229](#)

Pro Trinket, [236](#)

touchscreens, [211–212](#)

Adafruit Motor Shield library

installing, [259](#)

in sketches, [259–261](#), [264–266](#), [270–271](#), [273–275](#)

Adding and Displaying Time and Date with an RTC (project), [352–356](#)

Addressing Areas on the Touchscreen (project), [213–215](#)

algorithms, [24](#)

alternating current (AC), [25](#)

amperes or amps (A), [25](#)

Analog Devices, TMP36 temperature sensor, [79](#)

analog inputs, [12](#), [70](#)

analog signals, [69–70](#)

analog thermometer building project, [244–246](#)

`analogRead()` function, [70](#), [74](#)

`analogReference()` function, [75](#)

`analogWrite()` function, [38](#)

and (&&) operator, [62–63](#)

anodes, [29](#)

Arduino

about, [9–10](#)

community, [1–4](#), [393](#)

suppliers, [4–5](#)

Arduino boards and alternatives. *See also* Arduino Uno; Creating Your Own Breadboard Arduino (project), [233–239](#)

Arduino IDE (integrated development environment). *See also* Serial Monitor window board type selection, [233](#)

error messages, [20](#)

installation and configuration, [5–7](#)

libraries, [133](#), [134–135](#), [193–194](#)

screen layout, [14–16](#), [20](#), [90](#)

Arduino Store USA, [5](#)

Arduino Uno

about, [2](#), [235](#)

analog inputs, [70](#)

AREF pin, [74–75](#)

connecting to, [17](#)

hardware, [10–14](#)

I²C bus pins, [338](#)

interrupt monitoring, [149–150](#)

pulse-width modulation (~) pins, [38](#)

remote control of, [375–379](#)

in schematic diagrams, [46–47](#)

serial buffer, [94](#)

SPI pins, [346](#)

AREF (analog reference) pin, [74–75](#)

arithmetic operations, [73](#)

array elements, [113](#)

arrays, [112–114](#)

ASCII chart, [165](#)

Atmel ATmega328 microcontroller IC

EEPROM memory, [331–332](#)

pins, [229–230](#)

in schematic, [226–227](#)

uploading sketches to, [231–232](#)

attachInterrupt() function, [150](#)

audio amplifier circuits, [75–76](#)

Autoscroll box, [90](#)

B

B (base) pins, [40](#), [48](#)

backing sheets, [246](#)

batteries, [227–228](#), [248](#)

battery holders, [248–249](#), [256](#)

battery snaps, [227–228](#), [256](#)

battery testing project, [70–72](#)

baud, [90](#)

BC548 transistor, [39–40](#)

BCD (binary coded decimal) conversion, [355](#)

binary numbers

display project, [107–109](#)

for pixel presentation, [172](#)

quiz game project, [110–112](#)

working with, [104–106](#), [355](#)

bits, [104](#)

`blinkLED()` custom function, [84](#), [85–86](#), [88–89](#)

Board menu item, [233](#)

boards. *See also* Arduino Uno; breadboards; ProtoShields

 Arduino Uno alternatives, [234–239](#)

 choosing, [233–234](#)

 IDE type selection, [233](#)

boolean variable, [62](#)

Boolean variables, [62](#), [68](#)

bootloaders, [227](#)

bounce phenomenon, [54](#)

breadboards. *See also* ProtoShields, [31–32](#)

breakout boards, [212](#)

Building an Analog Thermometer (project), [244–246](#)

Building an Arduino Dialer (project), [385–388](#)

Building an Arduino Texter (project), [388–389](#)

Building a Remote Monitoring Station (project), [369–373](#)

Building and Controlling a Robot Vehicle (project), [254–261](#)

buttons. *See* push buttons

buzzers, [77–78](#)

byte variables, [105–106](#)

bytes, [104](#)

C

C (collector) pins, [40](#), [48](#)

capacitors

 ceramic, [51](#)

- electrolytic, [52](#)
- using, [50–51](#), [75](#)
- card readers. *See* memory card modules; RFID readers
- cathodes, [29](#)
- cellular communications
 - project hardware, [382–385](#)
 - projects, [385–388](#), [388–389](#), [390–393](#)
 - using, [381–382](#)
- ceramic capacitors, [51](#)
- CHANGE interrupt mode, [150](#)
- char statement, [175](#)
- character displays
 - with LED Matrix modules, [160–166](#)
 - with LedControl library, [157](#)
- character LCD modules
 - defining customer characters, [172–173](#)
 - demonstration sketch, [169–171](#)
 - hardware, [167–169](#)
- chassis models, [255](#)
- CheapStepper library
 - download, [252](#)
 - in sketches, [253–254](#)
- chip resistors, [27](#)
- circuit diagrams. *See* schematic diagrams
- circuits
 - building with schematics, [56–59](#)

- graph paper layouts, [130–131](#)
- higher-voltage, [41–42](#)
- properties, [24–25](#)
- with sketch example, [33–35](#)
- classes in sketches, [188](#)
- clock pin, [108](#)
- clock projects. *See also* real-time clock projects
 - GPS-based, [284–286](#)
- CNC plotter project (Michalis Vasilakis), [3–4](#)
- code systems
 - capacitor values, [51](#)
 - resistance values, [26–27](#)
 - schematic diagrams, [46–50](#)
 - Sony infrared signals, [316](#), [318](#)
 - Teleduino status, [377](#)
- coil schematic symbol, [48](#)
- collector (C) pins, [40](#), [48](#)
- collision sensing techniques, [262–266](#)
- colour tables, [175](#)
- COM (common) schematic symbol, [48](#)
- comments in sketches, [17](#)
- common-cathode modules, [115](#)
- comparison operators, [61](#), [62–63](#), [73](#)
- conditions in loops, [37](#)
- constructors, [188](#), [189](#)
- Controlling the Motor (project), [248–250](#)

Controlling Traffic (project), [64–68](#)

Controlling Two Seven-Segment LED Display Modules (project), [119–122](#)

.cpp (source) files, [187](#), [188–189](#)

Creating an Accurate GPS-Based Clock (project), [281–284](#), [284–286](#)

Creating an Arduino Tweeter (project), [373–375](#)

Creating a Blinking LED Wave (project), [33–35](#)

Creating a Custom Shield (project), [129–133](#)

Creating a Digital Thermometer (project), [122–123](#)

Creating an Electronic Die (project), [101–104](#)

Creating a Function to Repeat an Action (project), [84](#)

Creating a Function to Set the Number of Blinks (project), [85–86](#)

Creating an IR Remote Control Arduino (project), [318–321](#)

Creating an IR Remote Control Robot Vehicle (project), [321–324](#)

Creating a Keypad-Controlled Lock (project), [207–209](#)

Creating an LED Binary Number Display (project), [107–109](#)

Creating a Quick-Read Thermometer (project), [79–82](#)

Creating a Quick-Read Thermometer That Blinks the Temperature (project), [86–89](#)

Creating an RFID Control with “Last Action” Memory (project), [333–336](#)

Creating an RFID Time-Clock System (project), [360–365](#)

Creating a Simple Digital Clock (project), [356–359](#)

Creating a Simple RFID Control System (project), [328–331](#)

Creating a Single-Cell Battery Tester (project), [70–72](#)

Creating a Single-Digit Display (project), [117–119](#)

Creating a Stopwatch (project), [146–149](#)

Creating a Temperature History Monitor (project), [181–184](#)
Creating a Temperature-Logging Device (project), [142–144](#)
Creating a Three-Zone Touch Switch (project), [218–221](#)
Creating a Two-Zone On/Off Touch Switch (project), [215–218](#)
Creating a Wireless Remote Control (project), [293–298](#)
Creating Your Own Breadboard Arduino (project), [224–233](#)
crystal oscillators (“crystals”), [225–226](#)
CS (chip select) pin, [346](#)
current
 Arduino board limits, [39](#)
 with electric motors, [247](#), [250](#)
 in Ohm’s law (I), [30](#)
 properties, [24–25](#)

D

Darlington transistors. *See also* TIP120 Darlington transistor, [247](#)
data
 logging and log files, [143–144](#), [365](#)
 serial buffer, [93–95](#)
 writing to memory cards, [140–142](#)
data buses. *See also* I²C (Inter-Integrated Circuit) bus; SPI (Serial Peripheral Interface) bus, [337](#)
data display projects. *See also* numeric data displays
 LCD graphics, [181–184](#)
 web pages, [369–373](#)
data out pin, [108](#)
DC (direct current), [25](#)

- DC electric motors. *See* electric motors
- DC socket terminal blocks, [252](#)
- debounce circuits, [55](#)
- debugging, [92](#)
- DEC (decimal) parameter, [141](#)
- default: section, [207](#)
- #define statement, [187](#)
- Defining Custom Characters (project), [172–173](#)
- delay() function, [19](#), [150](#)
- Demonstrating a Digital Input (project), [55–61](#)
- Demonstrating PWM (project), [38–39](#)
- detachInterrupt() function, [150](#)
- Detecting Robot Vehicle Collisions (projects)
 - with infrared distance sensor, [269–271](#)
 - with microswitch, [262–266](#)
 - with ultrasonic distance sensor, [273–275](#)
- dialer-building project, [385–388](#)
- Digi-Key
 - digital rheostats, [348](#)
 - EEPROM, [339](#)
 - port expanders, [343](#)
- digital input/output pins
 - Arduino board, [12](#), [38](#), [39](#)
 - port expanders, [343](#)
 - timing state change, [145–146](#)
- digital inputs

- about, [53](#)
- demonstration project, [55–61](#)
- digital rheostats
 - connecting, [348–349](#)
 - testing, [349–350](#)
 - using, [348](#)
- digital signals, [69](#)
- Digital Stopwatch (project), [158–160](#)
- digital storage oscilloscopes, [54](#)
- digitalRead() function, [60](#), [69](#)
- digitalWrite() function, [19](#), [69](#)
- diodes, [40](#), [250](#)
- direct current (DC), [25](#)
- Displaying the Temperature in the Serial Monitor (project), [91–92](#)
- do-while statements, [93](#)
- Due (Arduino) board, [238–239](#)
- duty cycles, [37–38](#)
- Dyn (redirection service), [369](#), [373](#)

E

- E (emitter) pins, [40](#), [48](#)
- Edit menu, [15](#)
- EEPROM (electrically erasable read-only memory)
 - in comparison chart, [234](#)
 - external, [339–342](#)
 - internal, [331–333](#)

- in projects, [333–336](#)
- EEPROM library sketches, [331](#), [333–336](#)
- electric motors. *See also* stepper motors
 - controlling project, [248–250](#)
 - using, [247–248](#)
- electrical isolation, [41](#)
- electricity
 - Arduino board limits, [39](#)
 - properties, [24–25](#)
 - wall-power, [43](#)
- electrolytic capacitors, [52](#)
- electronic components. *See also specific components*
 - about, [25](#)
 - fundamental, [25–30](#), [39–41](#)
 - in schematic diagrams, [46–50](#)
- else. *See* if-else statements
- emitter (E) pins, [40](#), [48](#)
- equal (==) operator, [61](#)
- error messages, [20](#)
- Ethernet library sketches, [370](#), [373–374](#)
- Ethernet shields
 - hardware, [13](#), [126](#)
 - in projects, [238](#), [367–368](#), [371](#)

F

- FALLING interrupt mode, [150](#)

farads, [51](#)

FastLED library installation, [135–135](#)

feature creep, [24](#)

File menu, [15](#), [17](#)

files

- Arduino library requisites, [187–190](#)

- logs, [143–144](#), [286–289](#)

- writing to memory cards, [141–142](#)

fixed values, [60](#)

flash memory, [234](#)

float variables, [72](#), [73](#), [142](#)

for loops, [36–37](#)

Freetronics

- 433 MHz receiver shield, [295](#)

- Eleven board, [235](#)

- EtherMega board, [238](#)

- LCD & Keypad Shield, [281](#)

- pin labels, [229](#)

frequency bands, [299](#)

Fritzing application, [50](#)

FTDI cables, [232–233](#)

function creation

- accepting values, [85–86](#)

- example sketch, [84](#)

- overview, [83](#)

- returning values, [86](#)

function libraries. *See* libraries

G

GND (ground)

- and current, [25](#)

- in schematic diagrams, [49](#)

Google Maps, [283–284](#), [290](#)

GPS (Global Positioning System), [278](#), [283–284](#)

GPS data

- logging positions, [286–288](#)

- mapping with, [289–290](#)

- receiving, [282–283](#)

- sentence conversion, [281](#)

- time data, [284–285](#)

GPS receiver modules, [278](#)

GPS receivers

- building project, [281–284](#)

- using, [278](#), [280](#)

GPS sentences, [281](#)

GPS shields

- connecting, [278](#)

- in projects, [282–283](#), [284–285](#)

- testing, [280–281](#)

- using, [126](#), [127](#), [278](#), [279](#)

GPS Visualizer, [290](#)

graph paper printing program, [130](#)

graphic LCD modules

- background color, [174–175](#)

- connecting, [173–174](#)

- graphic functions, [177–180](#)

- projects, [181–184](#)

- text functions, [175–177](#)

greater than (>) operator, [73](#)

greater than or equal to (>=) operator, [73](#)

ground. *See* GND (ground)

H

.h (header) files, [187–188](#)

hardware suppliers, [4–5](#), [239](#)

HC-SR04 ultrasonic distance sensor, [271–272](#)

header (.h) files, [187–188](#)

heat sinks, [225](#)

Help menu, [15](#)

hexadecimal numbers, [321](#)

horns, [241–242](#)

I

I (current), [30](#)

I²C (Inter-Integrated Circuit) bus, [337](#), [338–339](#), [352](#)

IC (Integrated Circuit) extractors, [230–231](#)

IDE. *See* Arduino IDE (integrated development environment)

if-else statements, [61](#)

if-then statements, [60–61](#)

- #ifndef statement, [187](#)
- #include statement, [189](#), [190](#)
- instance creation, [188](#), [190](#)
- int variables, [35–36](#)
- interrupt handlers, [149](#)
- interrupts
 - about, [149–150](#)
 - demonstration project, [151–152](#)
 - modes and functions, [150](#)
 - in robot vehicle projects, [264](#)
- interrupts() function, [150](#)
- IP addresses, [369](#), [371](#), [372](#)
- IR (infrared) distance sensors
 - in robot vehicle collision detection project, [269–271](#)
 - testing, [267–269](#)
 - uses, [266](#)
 - wiring, [266–267](#)
- IR (infrared) remote controls
 - building project, [318–321](#)
 - operations, [315–316](#)
 - Sony TV remotes, [316–317](#), [318](#), [321](#)
 - test sketch, [317–318](#)
- IR receiver modules, [316](#)
- IR receivers, [316](#)
- IRremote library
 - download, [316](#)

in sketches, [317](#), [320–321](#), [321–324](#)
ISPs (internet service providers) and IP addressing, [369](#)

J

junction dots, [49](#)
justradios.com, [51](#)

K

k Ω (kiloohms), [26](#)
Kennedy, Nathan, [375](#)
Keypad library
 download, [204](#)
 in sketches, [205–206](#), [207–209](#)
keys, array conversion, [375](#)
KEYWORDS.TXT definition files, [187](#), [189–190](#)
kiloohms (k Ω), [26](#)
KIM-1 emulator (Oscar Vermeulen), [3](#)

L

L LED, [12](#)
L293D Motor Drive Shield, [257–258](#)
latch pin, [108](#), [109](#)
lc.clearDisplay function, [157](#)
lc.setChar() function, [157](#)
lc.setDigit() function, [157](#)
lc.setIntensity() function, [157](#)
lc.shutdown() function, [157](#)

LCDs (liquid crystal displays). *See also* character LCD modules; graphic LCD modules; LiquidCrystal library about, [167](#)

- number display, [171](#)

- text display, [170–171](#)

`lcd.begin()` function, [170](#)

`lcd.clear()` function, [170](#)

`lcd.createChar()` function, [172](#)

`lcd.print()` function, [171](#)

`lcd.setCursor()` function, [170](#)

`lcd.write()` function, [172](#)

least significant bit (LSB), [104](#)

LEDs (light-emitting diodes). *See also* LED projects; MAX2179 LED Driver IC; seven-segment LED display modules on Arduino board, [12](#), [16](#)

- brightness control effects, [37–38](#)

- connecting, [29–30](#)

- and resistors, [25](#)

- in schematic diagrams, [48](#)

- in sketch example, [18–21](#)

LED matrix modules

- connecting, [160–161](#)

- using, [162–166](#)

LED projects

- with Arduino built-in LED, [84](#), [85–86](#)

- binary number display, [107–109](#)

- Blinking LED Wave, [33–35](#), [36–37](#), [38–39](#), [49–50](#)

- circuit building demonstration, [55–61](#)

- controlling traffic, [64–68](#)
- electronic die-throwing, [101–104](#)
- LedControl() function, [157](#)
- LedControl library
 - download, [155](#)
 - sketches, [156–157](#), [158–159](#)
- LEDMatrixDriver library
 - download, [161](#)
 - sketches, [162–164](#)
- less than (<) operator, [73](#)
- less than or equal to (<=) operator, [73](#)
- libraries. *See also specific libraries*
 - about creating, [185–186](#)
 - custom demonstrations, [195–197](#), [197–201](#)
 - downloading and installing, [134–136](#)
 - installing custom, [190–194](#)
 - requisite files, [187–190](#)
 - using, [133](#)
- Library Manager, [136–135](#)
- Lilypad, [237](#)
- linear variable resistors, [75–76](#)
- linear voltage regulators, [224–225](#)
- Linux, Arduino IDE installation, [7](#)
- liquid crystal displays. *See* LCDs (liquid crystal displays)
- LiquidCrystal library sketches, [169–170](#), [282–283](#), [284–285](#), [357–359](#), [361–364](#)

- `lmd.setEnabled()` function, [164](#)
- `lmd.setIntensity()` function, [164](#)
- logarithmic variable resistors, [75–76](#)
- logging and log files, [143–144](#), [286–289](#)
- long variables
 - defined, [95](#)
 - using, [95–97](#)
- `loop()` function, [18](#)
- LoRa library
 - download, [299](#)
 - in sketches, [302–304](#), [306–309](#), [310–313](#)
- LoRa shields
 - in projects, [304–305](#), [309–314](#)
 - using, [298–299](#), [300](#)
- LOW interrupt mode, [150](#)
- LSB (least significant bit), [104](#)
- LSBFIRST parameter, [109](#), [116](#)

M

- MAC addresses, [372](#)
- macOS
 - Arduino IDE installation, [6](#)
 - ZIP file creation, [192–193](#)
- Making a Binary Quiz Game (project), [110–112](#)
- `map()` function, [218](#), [221](#)
- main-secondary devices

- I²C addressing, [338](#)
- SPI device connections, [346](#)
- MAX7219 LED driver IC. *See also* LedControl library
 - in Digital Stopwatch project, [158–160](#)
 - and LED numeric display modules, [154–155](#), [160](#)
 - package types, [153–154](#)
- Maxim DS3231 RTC module, [351–352](#)
- Mega 2560 (Arduino) board, [237–238](#)
- memory. *See also* EEPROM
- memory card modules. *See also* SD card library
 - connecting, [138–139](#)
 - testing, [139–140](#)
- memory cards
 - about, [137–138](#)
 - formatting, [137](#)
 - in GPS coordinates project, [286–288](#)
 - testing, [139–140](#)
 - writing data to (projects), [140–142](#), [142–144](#), [286–290](#), [360–365](#)
- message window area, [16](#)
- Microchip Technology
 - 24LC512 EEPROM, [339](#), [340](#)
 - MCP4162 digital rheostat, [348–350](#)
 - MCP23017 port expanders, [343–345](#)
- microcontrollers
 - Arduino, [11](#)
 - ATmega328p-PU, [226–227](#), [229–230](#)

- comparison chart, [234](#)
- removing and inserting, [230–231](#)
- microfarads (μF), [51](#)
- micros() function, [145](#)
- microSD card shields, [126](#), [127](#)
- microSD cards. *See* memory cards
- microswitches, [262–263](#)
- milliamps (mA), [25](#)
- millis() function, [145](#)
- Mini CNC Plotter (Michalis Vasilakis), [3–4](#)
- MISO (main in, secondary out) pin, [346](#)
- modulo functions, [120](#)
- MOSI (main out, secondary in) pin, [346](#)
- most significant bit (MSB), [104](#)
- motor shields, [257–258](#)
- MSB (most significant bit), [104](#)
- MSBFIRST parameter, [109](#)
- multimeters, [28](#)
- Multiplying a Number by Two (project), [94–95](#)

N

- Nano (Arduino) board, [236–237](#)
- NC (normally closed) schematic symbol, [48](#)
- network cables, [369](#)
- New icon, [16](#)
- No-IP (redirection service), [369](#), [373](#)

No Line Ending menu item, [94](#)

NO (normally open) schematic symbol, [48](#)

noInterrupts() function, [150](#)

not (!) operator, [62](#)

not equal (!=) operator, [61](#)

NPN-type transistors, [48](#)

numeric data display. *See also* MAX7219 LED driver IC; seven-segment LED display modules on LCD screens, [171](#)

 LED binary number project, [107–109](#)

numeric keypads

 connecting, [204–205](#)

 in keypad-controlled lock project, [207–209](#)

 using, [203–204](#)

numeric keypads. *See* keypads

O

ohms (Ω), [26–27](#), [47](#)

Ohm's law, [30](#)

Open icon, [16](#)

open source hardware, [239](#)

or (||) operator, [63](#)

oscilloscopes, [54](#)

output enable pin, [108](#)

P

picofarads (pF), [51](#)

piezoelectric (piezo) elements

- about, [77–78](#)
- demonstration project, [78–79](#)
- pin labels, [229–230](#)
- pinMode() function, [18](#), [60](#)
- pinout, [40](#)
- pins
 - Arduino Uno, [12](#)
 - ATmega328P-PU microcontroller IC, [229](#)
 - graphic LCD modules, [174](#)
 - I²C bus connectors, [338](#)
 - keypads, [205](#)
 - LCD modules, [168–169](#)
 - LED matrix modules, [160–161](#)
 - LED numeric displays, [155](#)
 - memory card modules, [139](#)
 - seven-segment display modules, [115–116](#)
 - shift registers, [107–108](#)
 - Teleduino digital, [378–379](#)
 - touchscreens, [212](#)
- pixels, [172](#)
- PMD Way
 - card readers, [327](#)
 - EEPROM, [339](#)
 - Ethernet shields, [367](#)
 - IR modules, [316](#)
 - LoRa shields, [299](#)

- ordering from, [5](#), [227](#)
- port expanders, [343](#)
- RF Link modules, [291](#)
- RTC ICs, [351](#)
- PNP-type transistors, [48](#)
- polarization, [29](#)
- port expanders, [343–345](#)
- port forwarding, [373](#)
- port type, [17](#)
- potentiometers, [75–77](#)
- power
 - defined, [25](#)
 - resistor ratings, [28](#)
- power connector, [11](#)
- power sockets, [12](#)
- private: section, [196](#)
- projects
 - ideas and examples, [1–4](#), [10](#)
 - parts list download, [5](#)
 - planning, [24](#)
 - safety, [8](#), [43](#)
- Proto-ScrewShields, [356–357](#), [360](#)
- ProtoShields
 - about, [125](#)
 - testing, [133](#)
 - using, [128](#), [129–132](#), [352](#)

public: section, [188](#)

pull-down resistors, [55](#)

pulse-width modulation. *See* PWM (pulse-width modulation)

push buttons

- in controlling traffic project, [64–68](#)

- demonstration project, [55–61](#)

- using, [53](#), [54](#)

- in wireless remote control project, [293–297](#)

PWM (pulse-width modulation), [37–39](#), [250](#)

Q

Q (transistor) schematic symbol, [48](#)

R

R (resistance), [30](#)

radio frequency (RF) modules. *See* RF Link modules

random() function, [100](#)

random numbers

- generating, [100–101](#)

- in projects, [101–104](#), [179–181](#)

real-time clock projects, [352–356](#), [356–359](#), [360–365](#)

Recording the Position of a Moving Object over Time (project), [286–290](#)

rectifier diodes

- about, [40–41](#)

- in circuit example, [41–42](#)

- in schematic diagrams, [47](#)

reference voltages, [73–75](#)

relays

- about, [41](#)

- in circuit example, [41–42](#)

- in schematic diagrams, [48](#)

Remote Control projects

- with infrared, [321–324](#)

- over internet, [375–379](#)

- over LoRa wireless, [299–304](#), [304–309](#)

- with radio frequency transmitters, [293–298](#)

- with text messaging, [390–393](#)

remote monitoring projects, [369–373](#)

Repeating with for Loops (project), [36–37](#)

RESET button, [13](#)

reset power sockets, [12](#)

resistance

- measurement and values, [26–28](#)

- in Ohm's law (R), [30](#)

resistors

- about, [25–28](#)

- pull-down, [55](#)

- in schematic diagrams, [47](#)

- variable, [75–77](#)

- in voltage dividers, [74–75](#)

RF Link modules

- using, [291–293](#)

- in wireless remote control projects, [293–298](#)

RFID (radio-frequency identification)

devices, [326–328](#)

operations, [325](#)

RFID readers

connecting, [327](#)

in projects, [328–330](#), [333–336](#), [360–365](#)

testing, [327–328](#)

using, [326–327](#)

RFID tags, [326](#), [328](#)

RGB color tables, [175](#)

rheostats. *See* digital rheostats

RISING interrupt mode, [150](#)

robot vehicle projects building and controlling, [254–261](#), [321–324](#)

detecting collisions, [262–266](#), [266–269](#), [269–271](#), [271–273](#), [273–275](#)

rotational range, [242](#)

RTC (real-time clock) IC modules. *See also* real-time clock projects

connecting, [352](#)

using, [351](#)

RX LED, [12](#)

S

Save as menu item, [17](#)

Save icon, [16](#)

schematic diagrams

building circuits from, [56–59](#)

drawing application, [50](#)

and ProtoShields, [128](#)

- using, [46–49](#)
- SCK (serial clock) pin, [346](#)
- SCL (clock line), [338](#)
- screw shields, [282](#), [286](#)
- SD card library sketches, [140–142](#), [142–144](#), [286–288](#), [361–364](#)
- SD card modules. *See also* memory card modules, [138](#)
- SD memory cards. *See* memory cards
- SDA (data line), [338](#)
- seeds, [100](#)
- Seeing the Graphic Functions in Action (project), [179–181](#)
- Seeing the Text Functions in Action (project), [176–177](#)
- semicolon (;), [18](#)
- Sending Remote Sensor Data Using LoRa Wireless (project), [309–314](#)
- serial buffer, [93–95](#)
- Serial Monitor icon, [16](#)
- Serial Monitor window
 - debugging with, [92](#)
 - using, [16](#), [89–90](#)
- serial ports
 - Arduino Uno pins, [12](#)
 - software, [279](#)
- `Serial.available()` function, [94](#), [150](#)
- `Serial.begin()` function, [90](#)
- `Serial.flush()` function, [95](#)
- `Serial.print()` function, [90](#)
- `Serial.println()` function, [90](#)

SerialGSM library

- download, [388](#)

- in sketches, [388–389](#)

Servo library sketches, [243–244](#)

servos

- in analog thermometer project, [244–246](#)

- connecting, [243](#)

- demonstration sketch, [243–244](#)

- using, [241–242](#)

Setting Up a Remote Control for Your Arduino (project), [375–379](#)

Setting Up an SMS Remote Control (project), [390–393](#)

setup() function, [18](#)

seven-segment LED display modules

- in projects, [117–119](#), [119–122](#), [122–123](#)

- using, [114–116](#)

74HC595 shift register IC, [106–109](#)

7805 linear voltage regulator, [224–225](#)

Sharp infrared analog sensor, [266](#)

shields. *See also specific shields*

- custom building project, [129–133](#)

- stacking, [127](#), [128](#)

- using, [13–14](#), [125](#), [126–127](#)

shift registers

- in LED binary display sketch, [109](#)

- pins, [108](#)

- schematic, [107](#)

- with seven-segment LED display modules, [115–116](#), [116–119](#), [119–122](#)
- using, [106–107](#)
- `shiftOut()` function, [109](#), [116](#)
- signals, digital vs. analog, [69](#)
- SIM cards, [382](#), [383](#)
- SIM5320 shield, [382](#)
- Sketch menu item, [15](#)
- sketches. *See also* functions; libraries
 - comments in, [17](#)
 - debugging, [92](#)
 - IDE window, [14–16](#)
 - modifying, [21](#)
 - uploading and running, [20](#), [230–233](#)
 - verifying, [20](#)
 - writing, [16–19](#)
- SMS (short message service) text messaging, [382](#)
- software. *See* Arduino IDE (integrated development environment); libraries; sketches software serial ports, [279](#)
- SoftwareSerial library
 - using, [279](#)
 - in sketches, [282–283](#), [284–285](#), [286–288](#), [327–328](#), [329–330](#), [384–385](#), [387–388](#), [391–393](#)
- soldering, [131–132](#)
- solderless breadboards. *See* breadboards
- Sony TV remotes, [316–317](#), [318](#), [321](#)
- source (.cpp) files, [187](#), [188–189](#)

SparkFun Electronics

ordering from, [5](#), [227](#)

RF Link modules, [291](#)

SPI (Serial Peripheral Interface) bus, [337](#), [346–347](#)

SPI data bus library sketches, [302](#), [304](#), [346–347](#), [349–350](#)

`SPI.begin()`, [347](#)

`SPI.setBitOrder()`, [347](#)

`SPI.transfer()`, [347](#)

spreadsheets, [144](#)

SRAM, [234](#)

SS (secondary select) pin, [346](#)

ST7735 TFT LCD module, [173–174](#), [181](#)

stacking shields, [126](#), [127](#), [128](#)

stall current, [247](#)

stepper motor controller boards

connecting, [251–252](#)

demonstration sketch, [253–254](#)

stepper motors, [251](#)

Stern, Becky, Wi-Fi Weather Display, [2–3](#)

stopwatch projects, [146–149](#), [151–152](#), [158–160](#)

`String()` function, [176](#)

`strlen()` function, [297](#)

surface-mount resistors, [27](#)

switch bounce, [54](#), [55](#)

switch case statement, [206–207](#)

T

Teleduino library download, [377](#)

Teleduino service

- in projects, [375–379](#)

- using, [375](#)

temperature-sensing and display projects

- analog display, [244–246](#)

- in custom library demonstration, [197–201](#)

- digital display, [122–123](#)

- historical display, [181–184](#)

- logging, [142–144](#)

- quick-read thermometer, [79–82](#), [86–89](#)

- sending remote data, [309–314](#)

- Serial Monitor display, [91–92](#)

temperature sensors. *See* TMP36 temperature sensor

terminal blocks, [252](#)

terminal shields, [262](#)

text displays, [170–171](#), [174–177](#)

text messaging

- building a texter, [388–389](#)

- remote control with, [390–393](#)

- using SMS, [382](#)

TFT graphics LCD library sketches, [174–176](#), [176–178](#), [179–180](#)

TFTscreen.background() function, [174](#)

TFTscreen.begin() function, [174](#)

TFTscreen.circle() function, [178](#)

`TFTscreen.fill()` function, [178](#)

`TFTscreen.line()` function, [178](#)

`TFTscreen.noFill()` function, [178](#)

`TFTscreen.point()` function, [178](#)

`TFTscreen.rect()` function, [178](#)

`TFTscreen.setTextSize()` function, [175](#)

`TFTscreen.stroke()` function, [175](#)

`TFTscreen.text()` function, [175](#), [176](#)

thermometer projects. *See* temperature-sensing and display projects 3G GSM shields

connecting, [383–384](#)

testing, [384–385](#)

using, [382](#)

time data. *See also* real-time clock projects; stopwatch projects creating a GPS-based clock, [284–286](#)

elapsed time recording, [144–146](#)

TinyGPS library

download, [281](#)

in sketches, [282–283](#), [284–285](#), [286–288](#)

TinySine 3G GSM shields, [382](#)

TIP120 Darlington transistor

about, [247–248](#)

in projects, [248–249](#)

TMP36 temperature sensor. *See also* temperature-sensing and display projects, [79–81](#), [82](#)

`toCharArray()` function, [176](#)

tokens (Twitter), [373](#)

Tools menu, [15](#), [233](#)

torque, [242](#)

touchscreens

- addressing and mapping, [213–215](#), [218](#)

- connecting, [212](#)

- in touch switch projects, [215–218](#), [218–221](#)

- using, [211](#)

transceivers, [298](#)

transistors

- about, [39–40](#)

- in circuit example, [41–42](#)

- Darlington, [247](#)

- in schematic diagrams, [48](#)

transmitters and receivers (TX/RX)

- in Freetronics Eleven board, [235](#)

- RF Link sets, [291–293](#)

trimpots (aka trimmers), [76–77](#)

true/false. *See* Boolean variables

Trying Out a Piezo Buzzer (project), [78–79](#)

Twitter and tweets, [373–375](#)

Twitter Arduino library

- download, [373](#)

- in sketches, [373–374](#)

Two-Wire Interface (TWI) bus. *See* I²C bus

TX LED, [12](#)

U

ultrasonic distance sensors

in collision detection project, [273–275](#)

connecting, [272](#)

testing, [272–273](#)

using, [271–272](#)

units of measure conversion charts, [51](#)

Uno. *See* Arduino Uno

unsigned long variable, [145](#)

Upload icon, [16](#), [20](#)

USB programming cables. *See* FTDI cables

USB (Universal Serial Bus) connector, [11](#), [12](#)

USB (Universal Serial Bus) interface

sockets, [12](#), [235](#)

uploading sketches with, [231–232](#)

Using a Digital Rheostat (project), [348–350](#)

Using an External EEPROM (project), [339–342](#)

Using Interrupts (project), [151–152](#)

Using LED Matrix Modules (project), [160–166](#)

Using long Variables (project), [95–97](#)

Using a Port Expander IC (project), [343–345](#)

V

V (volts), [25](#), [30](#)

variable resistors, [75–77](#)

variables

- displaying contents of, [91](#)
- private, [196](#)
- public, [188](#)
- using, [35–36](#)
- Vasilakis, Michalis, Mini CNC Plotter, [3–4](#)
- Verify
 - in IDE toolbar, [16](#)
 - using, [20](#)
- Vermeulen, Oscar, KIM-1 emulator, [3](#)
- VirtualWire library
 - download, [293](#)
 - in sketches, [296–298](#)
- Vishay TSOP4138 IR receiver, [316](#)
- void function type, [86](#)
- voltage
 - Arduino Uno limitation, [29–30](#)
 - and capacitors, [51](#)
 - measurement, [25](#)
 - in Ohm's law (V), [30](#)
 - reference, [73–75](#)
- voltage dividers, [74–75](#)

W

- W5100 controller chip, [367](#)
- weather display project, [2–3](#)
- web browsers, controlling Arduino from, [375–379](#)

web pages

creating, [369–373](#)

viewing, [373](#)

while statements, [93](#)

Wi-Fi Weather Display (Becky Stern), [2–3](#)

Windows

Arduino IDE installation, [7](#)

ZIP file creation, [190–191](#)

Wire library sketches, [338–339](#), [341–342](#), [345](#), [353–355](#), [357–359](#), [361–364](#)

`wire.begin()` function, [338](#)

`wire.beginTransmission()` function, [339](#)

`wire.endTransmission()` function, [339](#)

`wire.read()` function, [339](#)

`wire.requestFrom()` function, [339](#)

`wire.write()` function, [339](#)

wireless modules. *See* LoRa shields; RF Link modules

wires

breadboard, [31](#), [32](#)

in schematic diagrams, [48–49](#)

Writing Data to the Memory Card (project), [140–142](#)

Z

ZIP file creation

Mac OS X, [192–193](#)

Windows, [190–191](#)