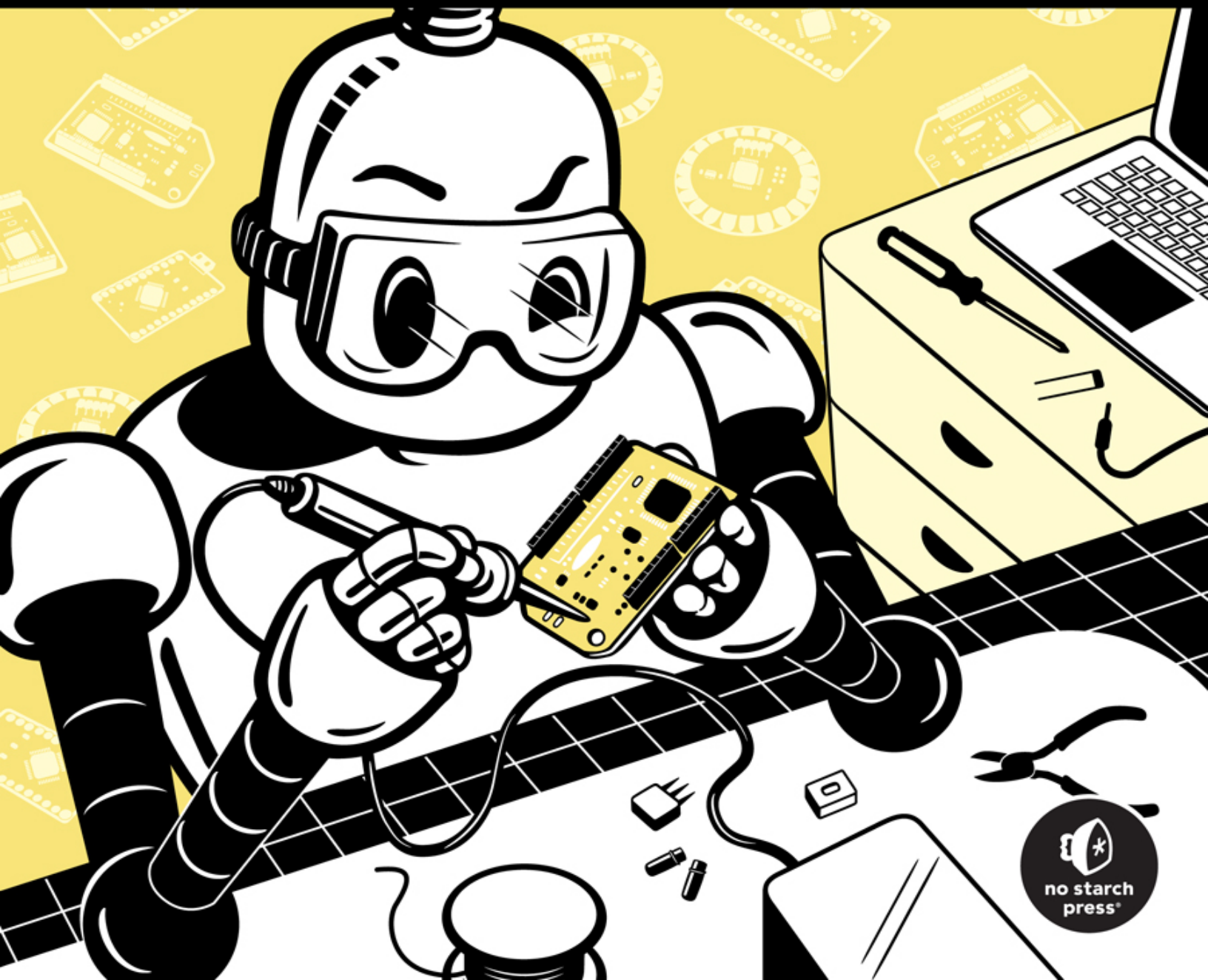
ARDUINO FOR ARDUINIANS

70 PROJECTS FOR THE EXPERIENCED PROGRAMMER

JOHN BOXALL



no starch press®

CONTENTS IN DETAIL

1. **TITLE PAGE**
2. **COPYRIGHT**
3. **ABOUT THE AUTHOR AND TECHNICAL REVIEWER**
4. **ACKNOWLEDGMENTS**
5. **INTRODUCTION**
6. Who Is This Book For?
7. What Can You Expect to Learn?
8. What I Expect You to Know
9. Materials You'll Need
10. Arduino IDE and Sketch Files
11. Parts and Accessories
12. Project PCBs
13. Moving On
14. **1**

    MULTIPLE BUTTONS WITH ONE ANALOG INPUT
15. Voltage Dividers
16. *Project #1: Three Buttons with One Analog Input*
17. *Project #2: Six Buttons with One Analog Input*
18. *Project #3: A 12-Button Keypad with One Analog Input*
19. Moving On

20. **2**

PORT MANIPULATION

21. rts, Pins, and Registers

22. *oject #4: Increasing Digital Output Pin Speed*

23. peed Advantage

24. *oject #5: Displaying Binary Numbers*

25. *oject #6: Creating a Blinking LED Wave*

26. *oject #7: Controlling Seven-Segment LED Displays*

27. *oject #8: Creating an Electronic Die*

28. *oject #9: Reading Multiple Digital Inputs Simultaneously*

29. nary-Coded Decimal Switches

30. *oject #10: Reading BCD Switches*

31. ving On

32. **3**

USING ATTINY MICROCONTROLLERS

33. e ATtiny85 Microcontroller

34. Ttiny Chips with the Arduino IDE

35. *oject #11: Building the “Hello, World” of Hardware*

36. e Arduino Uno vs. ATtiny85 Pin References

37. ding Reset Buttons to ATtiny85 Circuits

38. duino Functions Available for the ATtiny85

39. *oject #12: Creating a Quick-Read Thermometer*

40. creasing the ATtiny85’s Speed

41. scillator Speed

42. Oscillator Accuracy
43. *Project #13: Creating an ATtiny85 Programming Shield*
44. Moving On
45. **4**
BUILDING WATCHDOG TIMERS
46. Ensuring Constant and Reliable Operation
47. Watchdog Timer Theory
48. Watchdog Timer Circuit Configuration
49. *Project #14: Building a Watchdog Timer*
50. The 555 Circuit’s Inner Workings
51. Circuit Assembly
52. Moving On
53. **5**
CONTROLLING LEDS WITH CHARLIEPLEXING
54. Introduction to Charlieplexing
55. *Project #15: Charlieplexing with Six LEDs*
56. Larger Charlieplexing Displays
57. *Project #16: Charlieplexing with 12 LEDs*
58. *Project #17: Charlieplexing with the ATtiny85*
59. *Project #18: Building a 30-LED Matrix Display*
60. Making Custom Character Displays
61. Moving On
62. **6**
ADDING PROFESSIONAL POWER CONTROL

63. wer the Arduino without a Physical Switch
64. erating MOSFETs
65. ing the 555 Timer IC in Bistable Mode
66. *ject #19: Creating a Soft On/Off Switch*
67. *ject #20: Powering Off the Arduino Automatically*
68. tivating an Arduino from an External Device
69. rmally Open Contact Devices
70. tput Logic Devices
71. e DS3231 Real-Time Clock Library
72. *ject #21: Building an Event Logger*
73. ving On
74. **7**

    CONTROLLING AC MAINS POWER OUTLETS
75. tocouplers
76. mote-Control AC Outlets
77. cking the Outlet Transmitter
78. *ject #22: Controlling the Transmitter Board*
79. *ject #23: Controlling the Mains Outlet with a Timer*
80. *ject #24: Controlling the Mains Outlet with SMS*
81. ving On
82. **8**

    CONTROLLING HIGHER-POWERED DEVICES
83. e TPIC6B595
84. *ject #25: Creating a TPIC6B595 Binary Number Display*

Project #26: Building a PC-Controlled Relay Board

Using Multiple TPIC6B595s

Pyranha-Style LEDs

Project #27: Creating a Giant Seven-Segment LED Display

Single-Digit Display

Four-Digit Display

Moving On

9

BUILDING A DIGITAL MUSIC PLAYER AND SOUND BOARD

The YX6300 MP3 Module

Project #28: Building a Simple MP3 Player

Project #29: Building an MP3 Player Sound Board

Moving On

10

USING MULTIPLE I²C DEVICES WITH THE SAME ADDRESS

Project #30: Using an I²C Bus Scanner

The TCA9548A Breakout Board

Graphic OLED Displays

BMP180 Sensors

Project #31: Creating a Temperature and Air Pressure Display

Moving On

4. **11**

   EMULATING USB MICE AND KEYBOARDS WITH THE LEONARDO

5. USB Keyboards
6. Emulating Keyboard Modifiers and Special Keys
7. Pressing and Releasing One or More Keys
8. *Project #32: Logging Data with USB Keyboard Emulation*
9. *Project #33: Building a USB Shortcut Keyboard*
10. USB Mice
11. *Project #34: Keeping Your PC Awake Automatically*
12. USB Mouse Buttons
13. *Project #35: Creating a PC Random Painter*
14. Moving On
15. **12**

   TRANSFERRING DATA TO AND FROM USB FLASH DRIVES

16. Selecting and Preparing a USB Flash Drive
17. The USB Interface Module
18. Writing Data
19. Appending Data to a File
20. *Project #36: Logging Sensor Data*
21. Reading Numerical Data
22. *Project #37: Using a USB Security Key*
23. *Project #38: Configuring a Project via USB Flash Drive*

24. Reading Text
25. Moving On
26. **13**
INTERFACING WITH PS/2 KEYBOARDS
27. The PS/2 Keyboard
28. PCF8574 LCD Modules
29. Testing the I²C LCD and PS/2 Keyboard
30. *Project #39: Creating an RGB LED Tester*
31. *Project #40: Building a Text Capture Device*
32. Moving On
33. **14**
CONTROLLING THE ARDUINO WITH BLUETOOTH
34. Bluetooth Modules
35. Level Converter Modules
36. Pairing with Windows Devices
37. Pairing with macOS Devices
38. Pairing with Android Devices
39. Sending Data via Bluetooth
40. Windows Devices
41. macOS Devices
42. Android Devices
43. *Project #41: Capturing Data for Real-Time Monitoring*
44. *Project #42: Using Bluetooth for Digital I/O Control*
45. *Project #43: Using an Android App for Bluetooth Control*

Updating the Module Name and PIN

Moving On

15

ENERGY EFFICIENCY FOR PORTABLE PROJECTS

Basic Principles of Electricity

Measuring Power Consumption

USB-Sourced Power

Wired Power

Arduino Power Consumption

Estimating Battery Life

Software-Based Methods to Reduce Power Consumption

Power Down/Wake Periodic Mode

The DS3231 Real-Time Clock Library

Project #44: Creating a Low-Power Periodic Data Logger

Wake on Interrupt Mode

Inexpensive Sound Sensors

Project #45: Logging Interrupt Triggers

Hardware-Based Methods to Reduce Power

Uploading Sketches with the ICSP Pins

Project #46: Building a Minimalist 5 V Arduino Circuit

Moving On

16

MONITORING AUTOMOTIVE ELECTRONICS WITH THE CAN BUS

The CAN Bus
Required CAN Bus Hardware
Project #47: Monitoring Engine Data
Project #48: Logging OBDII Car Data
Moving On
17
ARDUINO-TO-RS232 COMMUNICATION
The RS232 Bus
Connecting to RS232
Testing Arduino-to-PC Connections via USB
Project #49: Creating a PC-to-Arduino Remote Control
Project #50: Enabling Arduino-to-Arduino Communication
Other RS232 Data Configurations
Moving On
18
ARDUINO-TO-RS485 COMMUNICATION
The RS485 Bus
Connecting to RS485
Project #51: Creating an Arduino-to-PC Data Link
Project #52: Creating an Arduino-to-Arduino Data Link
Project #53: Remote Control Operation
Controlling Two or More Secondary RS485 Devices
Project #54: Controlling Two or More Secondary Arduino Boards

88. ving On
89. **19**
THE ESP32 MICROCONTROLLER PLATFORM AND IOT
90. e ESP32
91. nfiguring the Arduino IDE for the ESP32
92. sting the ESP32
93. e GPIO Pins
94. e Wi-Fi Connectivity
95. rt Forwarding
96. *ject #55: Remote-Controlling a Single GPIO Pin*
97. *ject #56: Remote-Controlling Four GPIO Pins*
98. lse-Width Modulation
99. *ject #57: Building a Hosted Web Page for a User Interface*
00. ving On
01. **20**
REMOTE CONTROL VIA TELEGRAM
02. nfiguring Your Telegram Account
03. nfiguring the Arduino IDE
04. *ject #58: Remote-Controlling Four LEDs*
05. *ject #59: Retrieving Remote Data*
06. *ject #60: Automating Data Transmission*
07. ving On
08. **21**
RETRIEVE THE CURRENT TIME FROM AN INTERNET

TIME SERVER

The Network Time Protocol

Project #61: Retrieving the Time and Date

Project #62: Displaying the Time and Date on an OLED

Project #63: Displaying Two Time Zones on an OLED

Project #64: Building a Giant Digital Clock

Moving On

22

CAPTURE AND LOG DATA TO GOOGLE SHEETS

Project #65: Logging Time and Temperature Data

Preparing the Google Sheets Document

The Google Apps Script

Preparing the Hardware

Tips for Future Google Sheets Projects

Moving On

23

BUILDING A MINI WEB SERVER

Project #66: Creating a Basic Text Web Server

Project #67: Creating an ESP32 I/O Port Monitor

Project #68: Building a Time and Weather Server

Moving On

24

THE ESP32 CAMERA BOARD

Selecting an ESP32 Camera

Setting Up the ESP32 Camera

Project #69: Streaming Video from a Basic Camera Server

External Wi-Fi Antenna

Project #70: Taking Photos and Saving Them to a Memory Card

The ESP32 Camera Pinouts

Moving On

AFTERWORD

APPENDIX COMPRESSING AND INSTALLING ARDUINO LIBRARIES

Compressing Files into a ZIP File

Windows 7 and Newer

macOS 10 or Newer

Ubuntu 20.04 LTS and Newer

Installing Your New Library

INDEX

ARDUINO FOR ARDUINIANS

70 Projects for the Experienced Programmer

by John Boxall



San Francisco

ARDUINO FOR ARDUINIANS. Copyright © 2024 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.

First printing

27 26 25 24 23 1 2 3 4 5

ISBN-13: 978-1-7185-0278-9 (print)
ISBN-13: 978-1-7185-0279-6 (ebook)

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Sydney Cromwell
Developmental Editor: Abigail Schott-Rosenfield
Cover Illustrator: Gina Redman
Interior Design: Octopod Studios
Technical Reviewer: Xander Soldaat
Copyeditor: Kim Wimpsett

Proofreader: Elizabeth Littrell
Indexer: BIM Creatives, LLC

For information on distribution, bulk sales, corporate sales, or translations, please contact No Starch Press® directly at info@nostarch.com or:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Name: Boxall, John author.

Title: Arduino for Arduinians / by John Boxall.

Description: San Francisco : No Starch Press, [2024] | Includes index.

Identifiers: LCCN 2023016487 (print) | LCCN 2023016488 (ebook) | ISBN 9781718502789 (print) | ISBN 9781718502796 (ebook)

Subjects: LCSH: Arduino (Programmable controller) | Microcontrollers—Programming.

Classification: LCC TJ223.P76 B689 2024 (print) | LCC TJ223.P76 (ebook) | DDC 006.2/2—dc23/eng/20230422

LC record available at https://lccn.loc.gov/2023016487

LC ebook record available at https://lccn.loc.gov/2023016488

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.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

John Boxall has been an electronics enthusiast for more than 30 years, spending much of that time in the electronics retail, wholesale, and consulting fields. He also spent several years writing Arduino tutorials, projects, and reviews of kits and accessories at *https://tronixstuff.com*. His other books include *Arduino Workshop*, 2nd edition (2021), and *AVR Workshop* (2022), both from No Starch Press.

About the Technical Reviewer

Xander Soldaat is a former MINDSTORMS Community Partner for LEGO® MINDSTORMS. He has a deep background in IT infrastructure architecture, as well as embedded systems, compiler, and STEM curriculum development. He is currently an OpenShift cloud success architect at Red Hat. In his spare time, he likes to tinker with robots, electronics, retro computers, and tabletop games.

ACKNOWLEDGMENTS

As always, a huge thank-you to the Arduino team: Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, David Mellis, and all those who maintain and update the Arduino ecosystem. Without your vision, thought, and hard work, none of this would have been possible. Once again, kudos and thanks to the KiCad team for their open source electronics design automation suite, which I've used throughout this book for circuit schematics.

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

Thanks also to the following people (in no particular order) from whom I've received encouragement, inspiration, and support: Elizabeth Pryce, Mr. Richard Smith AC, the late Sir Clive Sinclair, and my wife, Kathleen, for her endless patience and missing husband.

Finally, thank you to everyone at No Starch Press for their efforts, including Abigail Schott-Rosenfield for their editorial input; Sydney Cromwell for guiding the book through the production process; Kim Wimpsett for copyediting; Elizabeth

Littrell for proofreading; and of course, Bill Pollock for his continuing support for authors who want to help people get into hardware and software design.

INTRODUCTION



I wrote this book for those who know the basics of working with the Arduino microcontroller environment but are hungry for more. Perhaps you've built the projects in *Arduino Workshop*, my first Arduino book for beginners, or some other introductory book. You know how to use data buses, control small motors, work with data from various sensors, and do much more. But there's much more to learn, and you're ready for more challenging projects.

In this book, you'll continue your journey into the world of Arduino-based learning and fun. Each chapter contains detailed tutorials on advanced techniques and tools you may have heard about but not yet tried, complete with instructions to build 70 complex and useful projects. I've chosen these topics based on the skills I've found most valuable during years of helping others create fun, useful, and innovative Arduino-based circuits and products.

Who Is This Book For?

This book is for those with intermediate-level Arduino experience who are ready to expand their skill set. You may have completed the introductory examples that come with Arduino starter kits, worked through a basic book on building Arduino projects, or taken a basic college-level computer engineering class that used an Arduino.

You might want to use this book as a reference to construct your own project ideas, or you might simply want to learn a range of new skills. If you're interested in improving power efficiency, interfacing with older devices using common data buses, logging data from your projects to online or offline storage, controlling projects over the internet, and more, this is the book for you.

To allow as many people as possible to complete this book, I've minimized the cost of building each project. You can easily purchase all required parts from a wide variety of retailers online.

What Can You Expect to Learn?

In each chapter of this book, I introduce a set of new concepts or tools and then teach you how to implement them in a total of 70 entertaining and educational projects. I hope these spur your creative imagination and become a reference for your own future work.

In general, you don't need to read sequentially, building the projects in one chapter before moving to the next one; instead, you can drop into any chapter you find interesting or useful. Chapters 19 and onward cover internet-connected projects, so it's simplest to read these sequentially, but it's not a requirement. When any chapter relies on content covered in a previous chapter, I provide the necessary references so you can go back and learn the required skills. Once you begin an individual chapter, read it from start to finish.

The following list describes the contents of each chapter in brief.

Chapter 1: Multiple Buttons with One Analog Input
Explains how you can read the status of one of many buttons using a single analog-to-digital input pin. This can be useful for creating user inputs with lots of buttons when you don't have many input pins available.

Chapter 2: Port Manipulation Explains how to use port manipulation with the digital output pins of your Arduino. Doing so reduces the amount of code required to control multiple outputs and allows you to control multiple digital outputs at once.

Chapter 3: Using ATtiny Microcontrollers Demonstrates how to use the smaller, cheaper ATtiny microcontrollers with the Arduino environment. ATtinys allow you to save money if your project doesn't need an entire Arduino board.

Chapter 4: Building Watchdog Timers Demonstrates how to build and use a watchdog timer, a small external circuit that can restart your project in case there's a hardware or software fault that causes it to stop working, which can be fixed with a reset.

Chapter 5: Controlling LEDs with Charlieplexing Introduces the concept of Charlieplexing, an efficient way of controlling one of many LEDs with fewer digital output pins.

Chapter 6: Adding Professional Power Control Shows you how to build an external circuit you can use to give your Arduino project soft-on and soft-off power control. You can use

this to make your Arduino turn itself off when a task has completed.

Chapter 7: Controlling AC Mains Power Outlets Covers a method of controlling mains power outlets safely by using wireless remote-control outlets and some easy soldering.

Chapter 8: Controlling Higher-Powered Devices Demonstrates how to control high-current shift register ICs, which allow your project to control higher currents than is possible with popular shift register ICs. The final project is a giant clock with a display comprised of super-bright LEDs.

Chapter 9: Building a Digital Music Player and Sound Board Introduces inexpensive MP3 player modules and shows you how to add audio to your Arduino projects for notifications and entertainment.

Chapter 10: Using Multiple I^2C Devices with the Same Address Demonstrates how to control multiple I^2C bus devices that share the same bus address using an external controller IC.

Chapter 11: Emulating USB Mice and Keyboards with the Leonardo Shows you how an Arduino Leonardo or compatible board can be used to emulate a USB mouse or

keyboard, enabling data logging directly to PC software or interesting and amusing projects.

Chapter 12: Transferring Data to and from USB Flash Drives Introduces how to save and retrieve data directly to or from a USB memory stick using an inexpensive module.

Chapter 13: Interfacing with PS/2 Keyboards Shows you how to interface your Arduino with a PS/2 style PC keyboard, which allows you to key data directly into your Arduino and gives you another option for a user interface.

Chapter 14: Controlling the Arduino with Bluetooth Demonstrates using inexpensive Bluetooth interface modules with your Arduino and building an Android app to remotely control your Arduino projects.

Chapter 15: Energy Efficiency for Portable Projects Shows you how to put your Arduino into various low-power and sleep modes, allowing for greater autonomy when being used on battery power.

Chapter 16: Monitoring Automotive Electronics with the CAN Bus Teaches you to interface the motor vehicle CAN data bus with your Arduino, so you can interrogate the vehicle's

engine management system for information about the car, like speed and RPM.

Chapter 17: Arduino-to-RS232 Communication
Demonstrates how to interface your Arduino with another Arduino or other devices over the popular RS232 data bus.

Chapter 18: Arduino-to-RS485 Communication
Demonstrates how to use the RS485 data bus for interfacing with RS485-compatible devices or just Arduino-to-Arduino communication over longer wired distances.

Chapter 19: The ESP32 Microcontroller Platform and IoT
The remaining chapters use an ESP32 Wi-Fi-enabled Arduino-compatible board to enable connection to the internet via a Wi-Fi access point. In Chapter 19, you'll use the ESP32 board to build a web-based remote control for your projects.

Chapter 20: Remote Control via Telegram Shows you how to remotely control and interact with your Arduino project from almost anywhere in the world using the Telegram social media app on a mobile device.

Chapter 21: Retrieve the Current Time from an Internet Time Server Teaches you to retrieve accurate time and date from network time servers on the internet using Wi-Fi.

Chapter 22: Capture and Log Data to Google Sheets Shows you how to log captured data from your ESP32 Arduino-compatible board directly to a Google Sheets spreadsheet for analysis and distribution.

Chapter 23: Building a Mini Web Server Explains how to build your own website to display data captured and analyzed by your ESP32 Arduino-compatible board.

Chapter 24: The ESP32 Camera Board Shows how to use and control a digital camera–enabled ESP32 Arduino-compatible board for video streaming or controlling the digital camera upon request.

Finally, if you need a reminder, you can refer to the appendix to learn how to create and install an Arduino library using the ZIP file installation method.

What I Expect You to Know

As this is an intermediate-level book, I expect you already know how to make a range of Arduino-based projects, from the simple to the more involved. You should also be familiar with writing your own Arduino sketches and uploading them to your board.

You should be able to understand circuit schematics to construct the circuits for each project. When describing connections to an Arduino or compatible board, instead of showing the entire board, a schematic uses labels to represent the Arduino pins to which connections should be made. For example, consider the schematic shown in Figure 1.

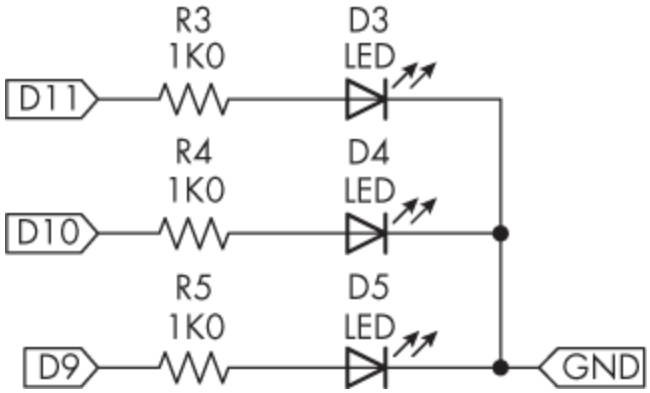


Figure 1: An example schematic

The Arduino's pin D11 is connected to R3, Arduino D10 to R4, and Arduino D9 to R5, with the LED's cathodes terminating at GND, which is connected to the Arduino's GND pin. Using schematics throughout this book will also improve your ability to read complex circuits.

Finally, be prepared to take appropriate safety measures for your workstation. For the projects in this book, you'll work with basic hand tools, battery-powered electrical devices, sharp knives, cutters, soldering irons, and more. As with any hobby or

craft, it's up to you to take care of yourself and those around you.

WARNING

At no point in your projects should you work directly with the mains current. Leave that to a licensed electrician who is trained for such work. Contacting the mains current will kill you.

If you need help or have any compliments or brickbats about the content of this book, please contact me via the publisher's website at *https://nostarch.com/contactus*.

Materials You'll Need

To complete the projects in this book, you'll need up-to-date Arduino IDE software and various hardware parts and accessories, including a few printed circuit boards.

Arduino IDE and Sketch Files

If you haven't already done so, download and install the latest version of the Arduino IDE software. You can find versions for Windows 10 and newer, macOS 10.14 64-bit and newer, and Linux (Ubuntu or similar) from *https://www.arduino.cc/en*

/software. 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). If you're still using an older IDE, update to the latest version.

Next, visit the book's website at *https://nostarch.com/arduino-arduinians* to download the Arduino sketch files for every example and project in this book, as well as a list of parts used, PCB design files for the boards in some projects, and errata, if necessary. I'll explain how to order the PCB files shortly.

Parts and Accessories

As with any other electronic device, the Arduino is available from many retailers offering a range of products and accessories. When shopping for a microcontroller, please purchase an original Arduino or a quality derivative—otherwise you run the risk of receiving faulty or poorly performing goods. Why risk buying an inferior board that could end up costing you more in the long run? For a list of genuine Arduino suppliers, visit *http://arduino.cc/en/Main/Buy.*

All the parts required for projects are listed in their respective chapters. I recommend these suppliers (listed in alphabetical

order) for your purchases of Arduino-related parts and accessories:

Altronics (Australia) *https://altronics.com.au*

Arduino Store USA *https://store-usa.arduino.cc*

Newark/element14 *https://element14.com*

PMD Way *https://pmdway.com*

RS Components *https://www.rs-components.com*

SparkFun Electronics *https://sparkfun.com*

All the parts required for the projects in this book are common and easily available from the various resellers listed and others that you may already be familiar with. But don't go shopping yet. Take the time to read a few chapters to get an idea of what you'll need so that you won't waste money buying unnecessary things.

Project PCBs

It's possible to assemble all projects in this book using one or more solderless breadboards, which may also require some external soldering to other devices. For example, in Chapter 7,

you'll build a wireless AC outlet remote controller with the interface circuit on a solderless breadboard. However, for certain projects, such as the 30-LED matrix display in Chapter 5, ordering printed circuit boards (PCBs) will make your projects more reliable and easier to build. This also gives you a more permanent example of your efforts.

After you download and extract this book's files from the No Starch website, you'll find a group of ZIP files, each with a project number. Each project's ZIP file contains a group of files called Gerbers, which PCB manufacturers use to control the machines that shape and drill blank PCBs into the desired configuration.

You can view the PCBs by using a Gerber viewer tool such as *https://www.gerblook.org*. Simply upload your Gerber ZIP file, and the tool should display the front and back of the PCB along with the other layers of interest, as shown in Figure 2.

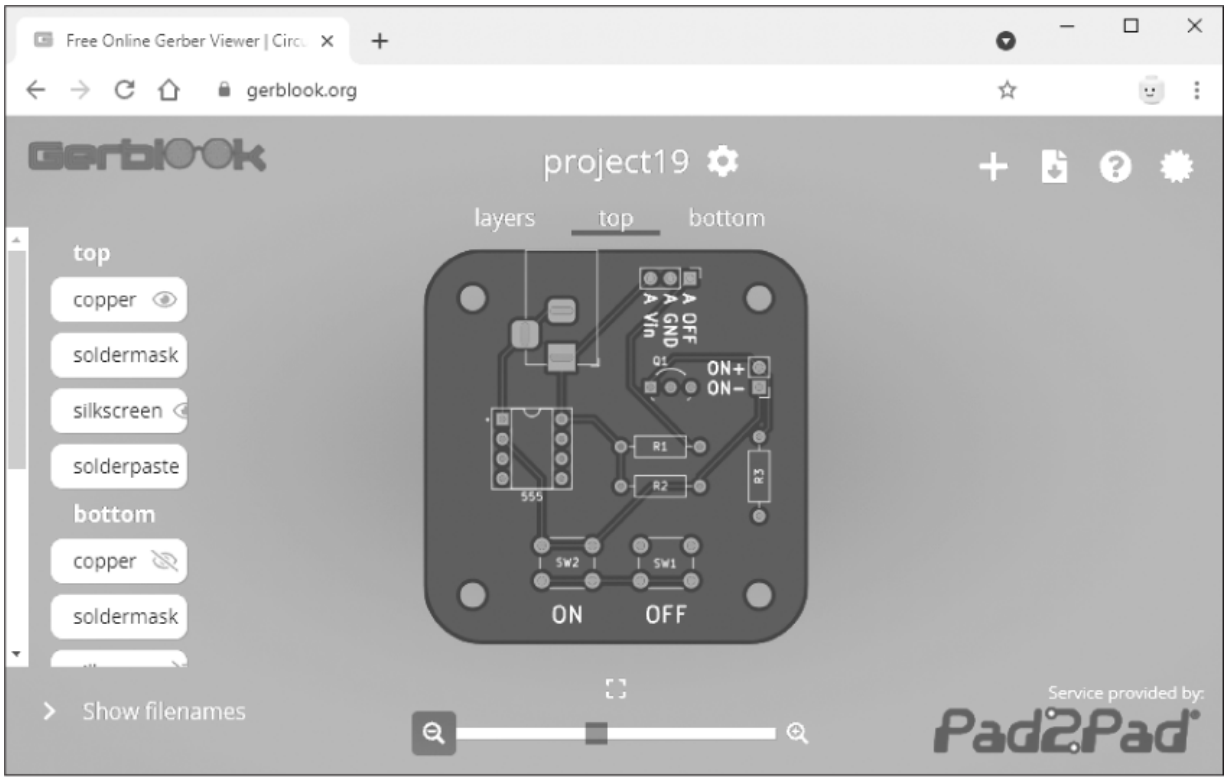


Figure 2: Using an online Gerber viewer to display the PCB for Project #19

Let's go through the process for ordering PCBs. You don't have to order all the PCBs for the book at once, but note that the turnaround time can be two to three weeks.

First, visit your preferred PCB manufacturer. I've found success with the following (listed in alphabetical order):

JLCPCB *https://jlcpcb.com*

OSH Park *https://oshpark.com*

PCB Way *https://pcbway.com*

You might use JLCPCB if you're looking to save money. I've found that OSH Park, based in the United States, has high-quality PCBs, and PCB Way sells good-quality items at reasonable prices. This is by no means an exhaustive list, just three companies that I've used with success over the years.

The manufacturer website you choose will generally have a Quote Now or Quick Order button on the front page. Clicking that should lead you to an Add Gerber File button. For example, Figure 3 shows a screenshot of that page on the PCB Way website.

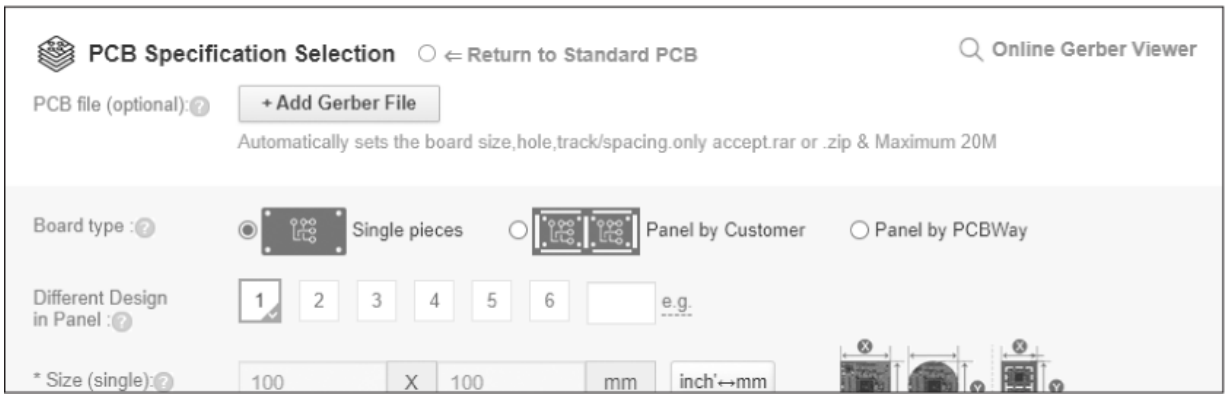


Figure 3: Uploading Gerbers to PCB factory

Click **Add Gerber File**, navigate to your required ZIP file, and click **Open** or **OK** to upload the ZIP file to the factory website. (You don't need to decompress the individual project ZIP files before uploading them, as the manufacturers can accept them without modification.) Once you upload your ZIP, the site

should show you your PCB's top and bottom sides, along with the measurements and potentially an initial price estimate, as shown in Figure 4. In general, the cost of a PCB can range from US$10 for smaller projects to US$40 or so for the larger ones.

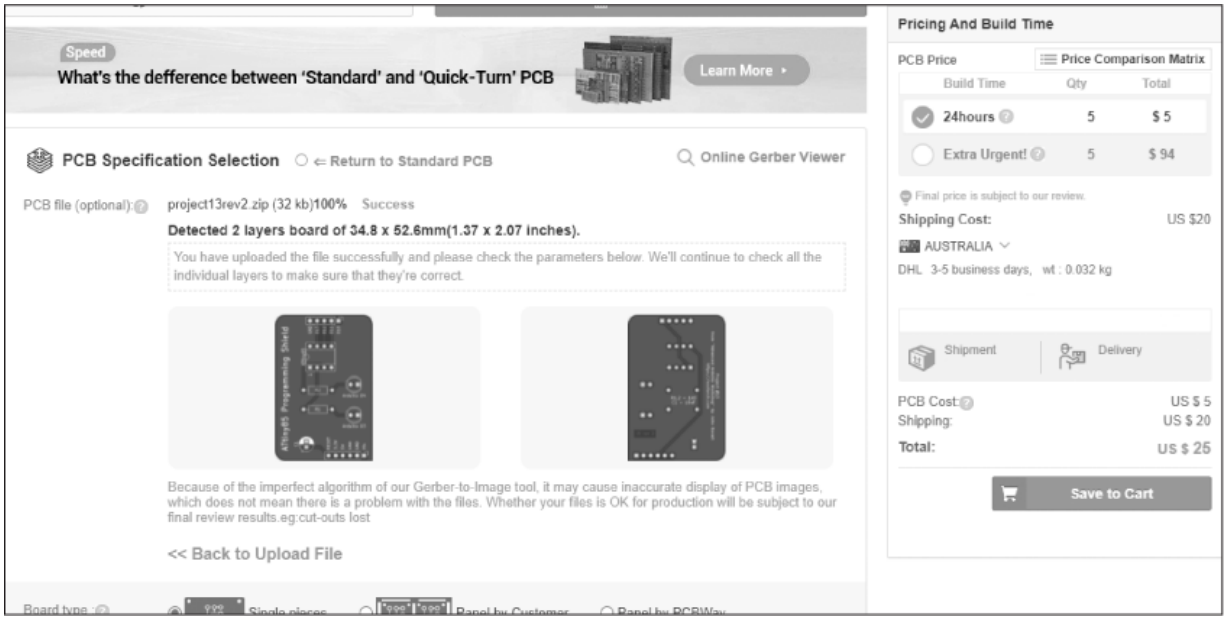


Figure 4: An example of a PCB price and freight quote displayed after uploading PCB ZIP files

Once you see the image of the PCB you're in the process of ordering, scroll down to find many design options to choose from, as shown in Figure 5.

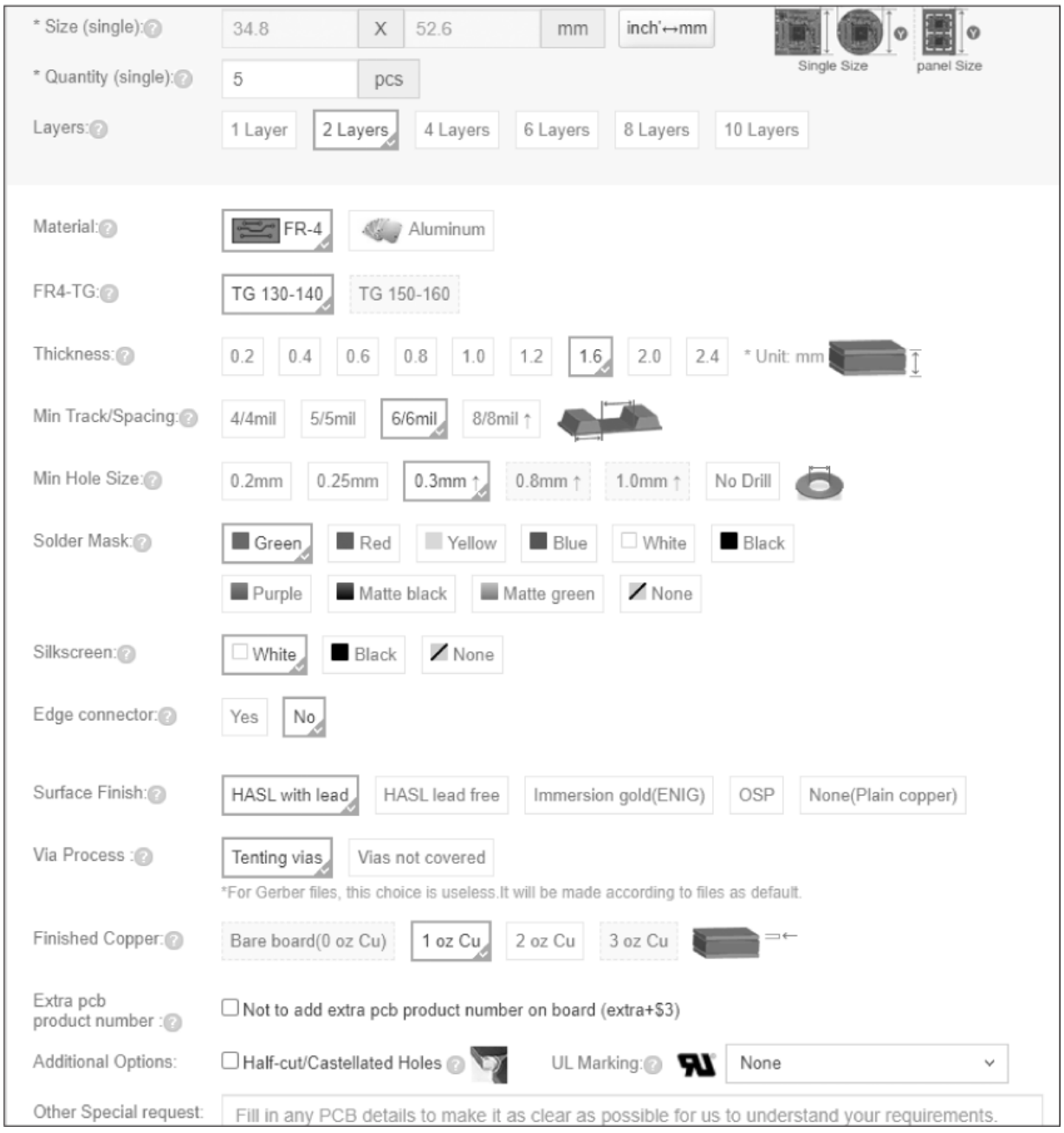


Figure 5: Selecting PCB options

Don’t panic—you only need to worry about a few of these options. You can change the following to suit your needs:

Quantity PCB factories will generally have a minimum quantity of three or five boards, but you can order as many as you like.

Solder mask This is the color of the protective coating on both sides of the PCB. Generally, the default color is green, but you can change this to suit your tastes or needs.

Silkscreen This is the color of the text printed on top of the solder mask, such as the board's labels or titles. Again, change this to suit your preferences.

Surface finish By default, the factory will generally use a lead-based process, which has been the standard for decades. If you prefer a lead-free environment or are having the boards delivered to a country that requires lead-free PCBs, you can change the surface finish to HASL lead-free or immersion gold.

Once you've made your selections or left them as the default, click **Next** or **Calculate** to see the final price. The website should also offer various delivery options, from cheap post to air courier, as shown in Figure 6. Save your PCB to your cart and then complete the last few steps for payment and delivery.

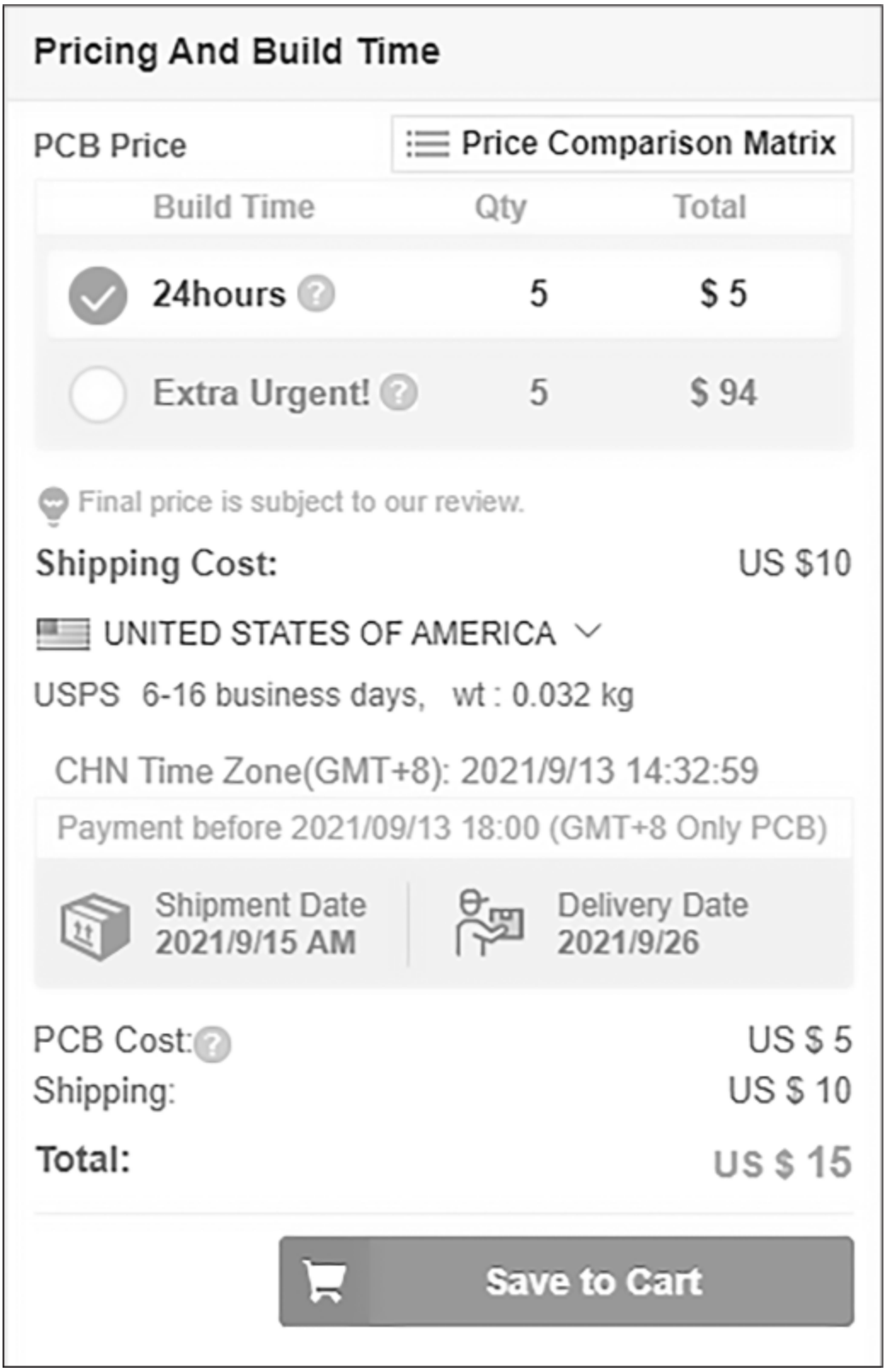


Figure 6: Final PCB quote

You should receive a set of nicely machined PCBs like those shown in Figure 7.

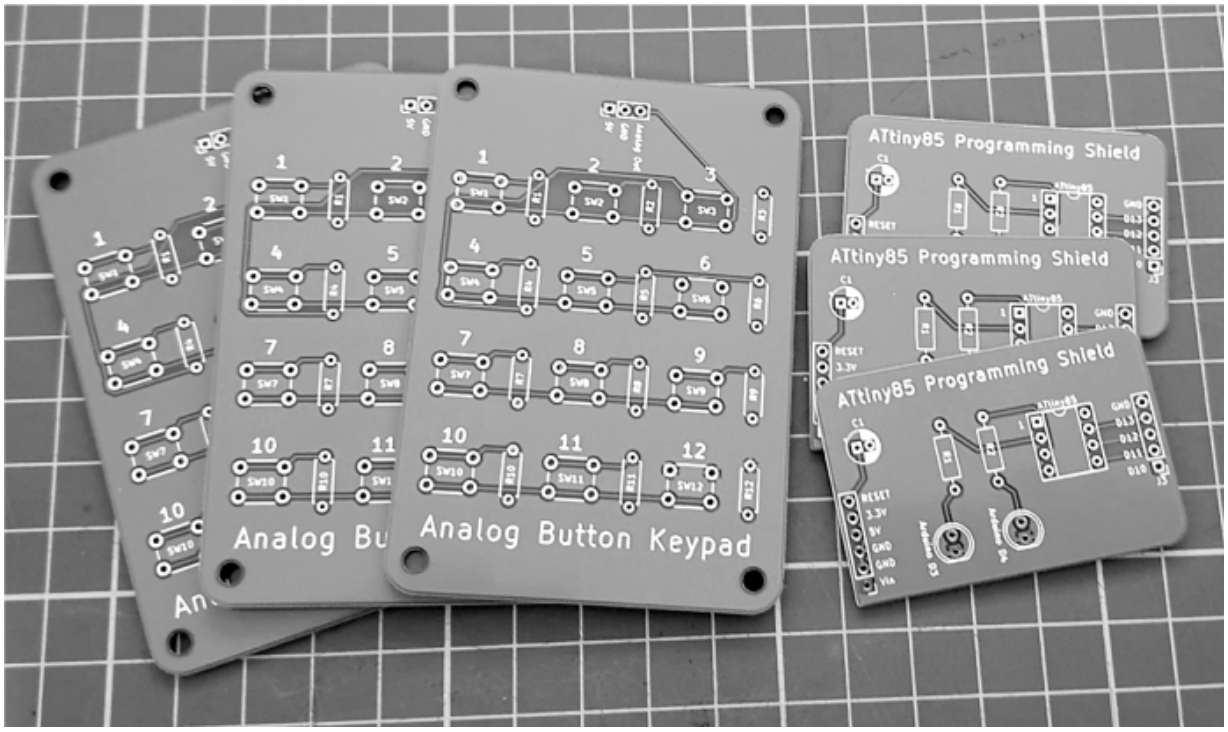


Figure 7: PCBs for various projects

These boards will form the basis for reliable and long-lasting projects.

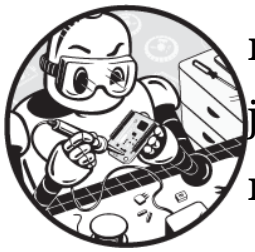
Moving On

In this introduction, you learned about the structure of this book, where to get the parts, and how to order your own PCBs used in some of the book's projects. You're ready to continue your Arduino journey, creating things you may never have

thought possible, from the simple yet convenient to the relatively complex. You'll also learn new methods to apply to your own designs, bringing original projects to life. Let's get started!

1

MULTIPLE BUTTONS WITH ONE ANALOG INPUT



Rather than using multiple digital inputs for multiple buttons, you can read many buttons with just one analog input pin. This not only reduces the number of input/output (I/O) pins used on your Arduino but can also save you money since you won't need any external integrated circuits (ICs), such as shift registers or port expanders.

In this chapter, you'll learn to:

- Use voltage dividers and buttons to create different voltages, differentiating buttons connected to an analog-to-digital converter (ADC) pin

- Read three or six buttons with one analog input
- Build a 12-button keypad that can be read with one analog input

Voltage Dividers

The multiple-button method used in this chapter is based on your Arduino's analog-to-digital converter (ADC) pins, which return a value relative to the voltage being connected to the ADC pin. Recall that your Arduino Uno or compatible board has six ADC pins, as shown in Figure 1-1.

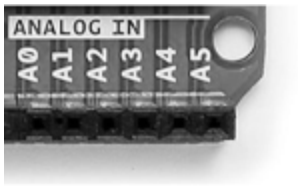


Figure 1-1: Analog inputs on the Arduino Uno

To build circuits that allow for reading up to 12 buttons at once via a single ADC pin, you'll need to use *voltage division*: the process of using two or more resistors to reduce a larger voltage to multiple smaller voltages, which the Arduino can read one at a time via an analog input pin. Figure 1-2 shows how you'd use two resistors for this purpose.

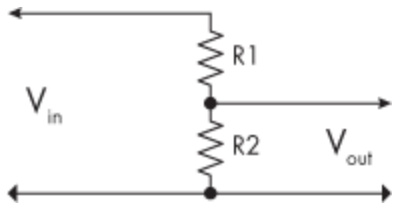


Figure 1-2: The schematic for a basic voltage divider

You can calculate the output voltage (V_{out}) with the following formula:

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

Substituting resistors of the same value and any voltage will result in an output voltage V_{out} that is half the input voltage V_{in}. For example, if both R1 and R2 are 1 kΩ and you have a 5 V input voltage, the calculation is V_{out} = 5 (1,000 / (1,000 + 1,000)), giving you a V_{out} of 2.5 V.

Resistors don't have an exact value; the more accurate the resistor, the more difficult it is to manufacture. You can choose between resistors of various tolerances, such as 5 percent or 1 percent. For example, a 1 kΩ, 1 percent resistor has a value between 990 and 1,010 Ω. When building voltage dividers or circuits in general, always try to use resistors with a 1 percent tolerance—they aren't much more expensive than 5 percent resistors, and your results will be more accurate.

When building a voltage divider, ensure that your resistors can handle the power you plan to run through them. For example, if you're using 0.25 W resistors and need to divide 24 V DC into 12 V, you can draw only 20 mA through the divider at most. Use the formula $W = V \times A$ (Watts = Volts (DC) × Amps (current)) to help you calculate how much power is used.

Combining the example voltage divider in Figure 1-2 with two buttons results in the circuit shown in Figure 1-3.

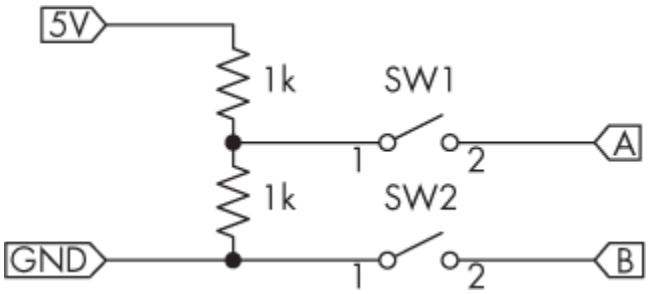


Figure 1-3: A voltage divider with two buttons

When you press the first button, the voltage at point A should be 2.5 V DC, as the current flows from the 5 V source and is divided by the two resistors. When you press the second button, the voltage measured at point B should be 0 V DC, as B is connected directly to GND. If you connect points A and B to an analog input, you should be able to detect which button has been pressed, as the values returned by the ADC will be different for each button (in theory, the values should be 512 for

button 1 and 0 for button 2). You're reading two buttons with only one input!

If you press two or more buttons at the same time, another voltage—determined by the combination of resistor dividers relative to the buttons pressed—should be returned to the ADC. The simplest way to determine the value is to build the circuit and test for yourself. With voltage division, you can use a series of buttons and resistors of the same value to generate different output voltages that your Arduino's analog input can read, enabling you to differentiate between each button.

Project #1: Three Buttons with One Analog Input

Let's start with a simple example of multiple button use: reading three buttons with one analog input. You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- Three 1 kΩ, 0.25 W, 1 percent resistors
- One 10 kΩ, 0.25 W, 1 percent resistor
- Three tactile buttons
- A solderless breadboard
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 1-4 on the solderless breadboard and connect the points labeled 5V, GND, and A0 to the Arduino's 5V, GND, and A0 pins, respectively.

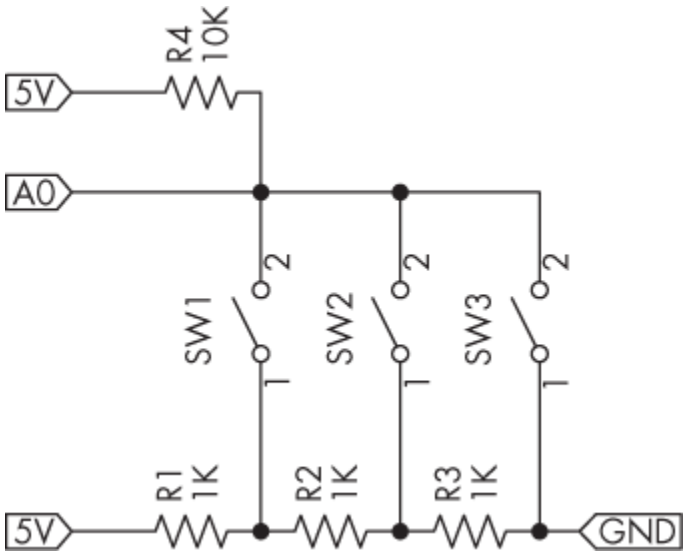


Figure 1-4: The circuit for Project #1

Now enter and upload the code for Project #1 to your Arduino. Wait a moment; then open the Serial Monitor in the Arduino IDE and start pressing the buttons one at a time. When you aren't pressing any buttons, the value returned by the ADC should hover around 1,010. Experiment with pressing buttons 1, 2, and 3 to return different values, as shown in Figure 1-5.



Figure 1-5: Example output from Project #1

Let's take a look at the code to see how this works:

```
// Project #1 - Reading three buttons from one a

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

void loop()
{
 ❶ int sensorValue = analogRead(A0);
 ❷ Serial.println(sensorValue);
   delay(100);
}
```

The code in Project #1 takes the value measured by analog pin A0 ❶ and displays it on the Serial Monitor ❷. This is to demonstrate the different analog values generated by the three-button circuit. After you upload the code, open the Serial Monitor and start pressing buttons. The example returned a value between 700 and 704 when button 1 was pressed, between 383 and 385 when button 2 was pressed, and 0 when button 3 was pressed. Your values may vary slightly because of the tolerance of the resistors.

As you press buttons, the current flows through at least one resistor as your R1 and, depending on the button pressed, through none or some of the resistors, which act as voltage dividers.

Using the resistor divider formula, you can calculate that pressing button 1 gives an R1 value of 1 kΩ and an R2 value of 2 kΩ (the sum of the resistors below the junction with the button). Using V_{out} = 5 (2,000 / (1,000 + 2,000)), the theoretical voltage presented to the ADC is 3.33 V. If you press button 2, V_{out} = 5 (1,000 / (1,000 + 2,000)) results in 1.66 V. If you press button 3, V_{out} = 5 (0 / (3,000 + 0)) results in 0 V, as the ADC is simply connecting straight to GND.

Finally, if you press no buttons, the value the ADC returns should hover above 1,010. The project enables this using the 10 kΩ pullup resistor in the circuit, which bridges the 5 V supply to the analog input. Without this resistor, the ADC would "float," reporting random values that may fall into the ranges of the other buttons. This is important, as we need a constant value range to be returned when no buttons are pressed.

In practice, the values returned by the ADC will vary a little. For example, the 5V pin on the Arduino (if powered via USB) measures slightly less, as some voltage is lost through the microcontroller, which affects the calculation in the ADC code. Always test your code with the Arduino powered by your intended power supply (USB or external DC power) to ensure accuracy with button-reading code.

Now that you've experimented with three buttons, let's try a more complex design.

Project #2: Six Buttons with One Analog Input

This project reads six buttons from one analog pin, with an improved sketch for button status reporting. You'll need the following parts:

- An Arduino Uno or compatible board and USB cable

- Six 1 kΩ, 0.25 W, 1 percent resistors
- One 1 kΩ, 0.25 W, 1 percent resistor
- Six tactile buttons
- A solderless breadboard
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 1-6 on the solderless breadboard and connect the points labeled 5V, GND, and A0 to the Arduino's 5V, GND, and A0 pins, respectively.

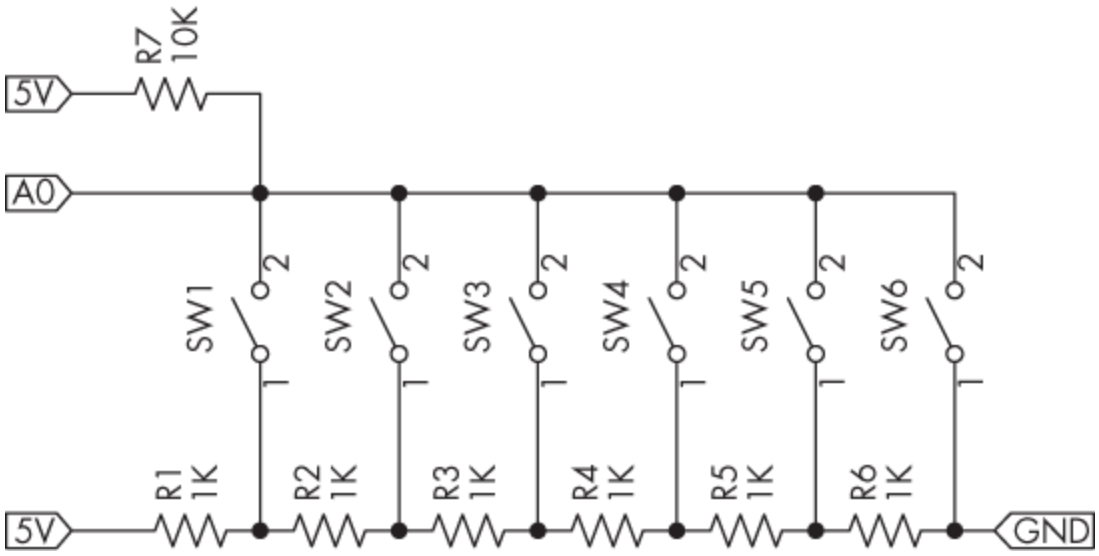


Figure 1-6: The circuit for Project #2

Now enter and upload the following sketch to your Arduino:

```
// Project #2 - Reading six buttons from one ana

void setup()
```

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

❶ int analogButton()
  // Returns number of button pressed by comparing
  {
      int adcValue;
      int button;
   ❷ adcValue = analogRead(A0);
      if (adcValue>1000) {button = 0;}
      else if (adcValue>857 && adcValue<861) {butt
      else if (adcValue>696 && adcValue<700) {butt
      else if (adcValue>535 && adcValue<539) {butt
      else if (adcValue>371 && adcValue<375) {butt
      else if (adcValue>199 && adcValue<203) {butt
   ❸ else if (adcValue>=0 && adcValue<16)   {butto
   ❹ return button;
  }
  void loop()
  {
   ❺ switch(analogButton()) // Read button status
   ❻ {
          case 0 : Serial.println("No button presse
          case 1 : Serial.println("Button 1 presse
          case 2 : Serial.println("Button 2 presse
          case 3 : Serial.println("Button 3 presse
          case 4 : Serial.println("Button 4 presse
```

```
            case 5 : Serial.println("Button 5 presse
            case 6 : Serial.println("Button 6 presse
        }
        delay(250);
    }
```

To simplify using the button circuit, the custom `analogButton()` function ❶ returns the number of the button being pressed, or 0 if nothing is pressed. The function takes the reading from analog pin A0 ❷ and compares it against the expected ranges returned for each button press ❸. The sketch returns the button number as the result of the `analogButton()` function ❹. The custom function in the main loop checks for a button press ❺, while a `switch…case` function acts on different button presses ❻.

Your required ranges from the analog input will vary; use the sketch from Project #1 to determine the exact ranges required for use in this project.

Project #3: A 12-Button Keypad with One Analog Input

This project expands the circuit in Project #2 to 12 buttons to create a larger final design that uses its own Arduino library. Using this library reduces the code required in the main sketch

used to read the buttons. It's possible to build this circuit on a solderless breadboard, but I recommend you download the PCB files for this project and make your own permanent version.

If you're using the breadboard, you'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- Twelve 1 kΩ, 0.25 W, 1 percent resistors
- One 10 kΩ, 0.25 W, 1 percent resistor
- Twelve tactile buttons
- A solderless breadboard
- Male-to-male jumper wires

If you're using the PCB, you'll need the following:

- An Arduino Uno or compatible board and USB cable
- Twelve 1 kΩ, 0.25 W, 1 percent resistors
- One 1 kΩ, 0.25 W, 1 percent resistor
- Twelve 6 × 6 × 4.3 mm tactile buttons
- 1 × 3 inline 2.54 mm header pins
- Male-to-female jumper wires

Figure 1-7 shows the schematic for this project.

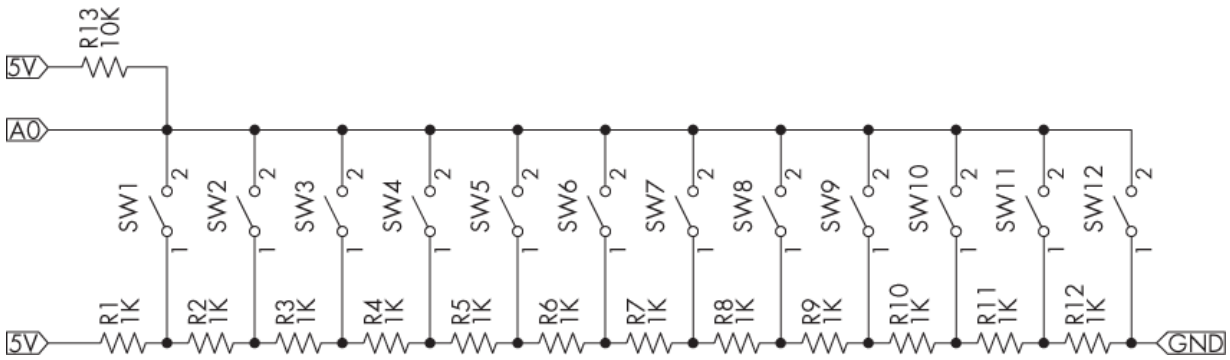


Figure 1-7: The schematic for Project #3

If you're building the PCB circuit, the parts are labeled so you can easily position the components on the PCB, as shown in Figure 1-8.

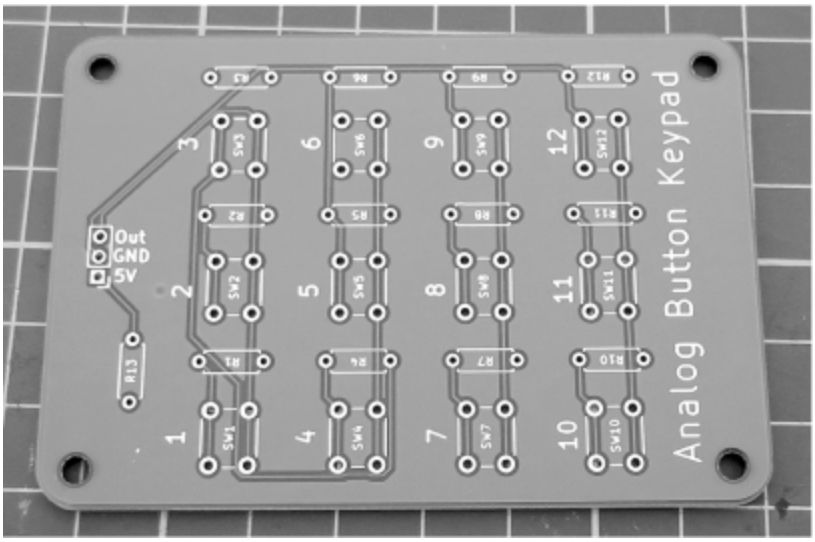


Figure 1-8: The top of the PCB for Project #3

Fit and solder the resistors and then the buttons. In general, when assembling circuits on PCBs, always start with the lowest-height parts first, and then work toward the largest or bulkiest

items. The inline header pins are usually sold in strips of 40 pins, so you'll need to trim off a 3-pin section to use with this project.

Once assembled, your project should resemble the board shown in Figure 1-9.

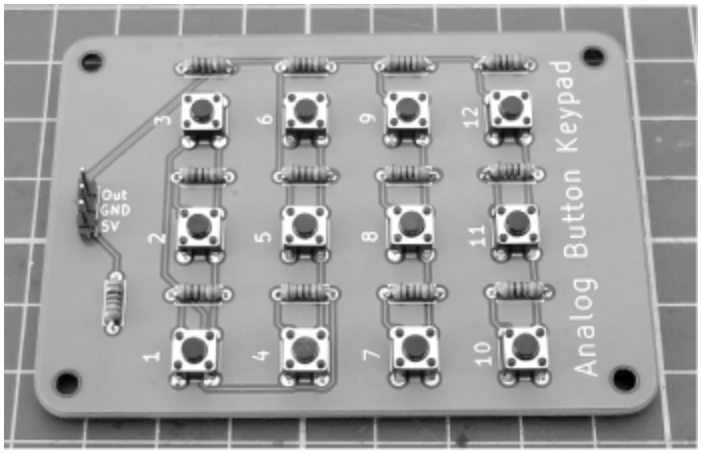


Figure 1-9: The completed Project #3 PCB

Now connect your circuit to the Arduino via the 5V, GND, and A0 pins, as in Figure 1-10.

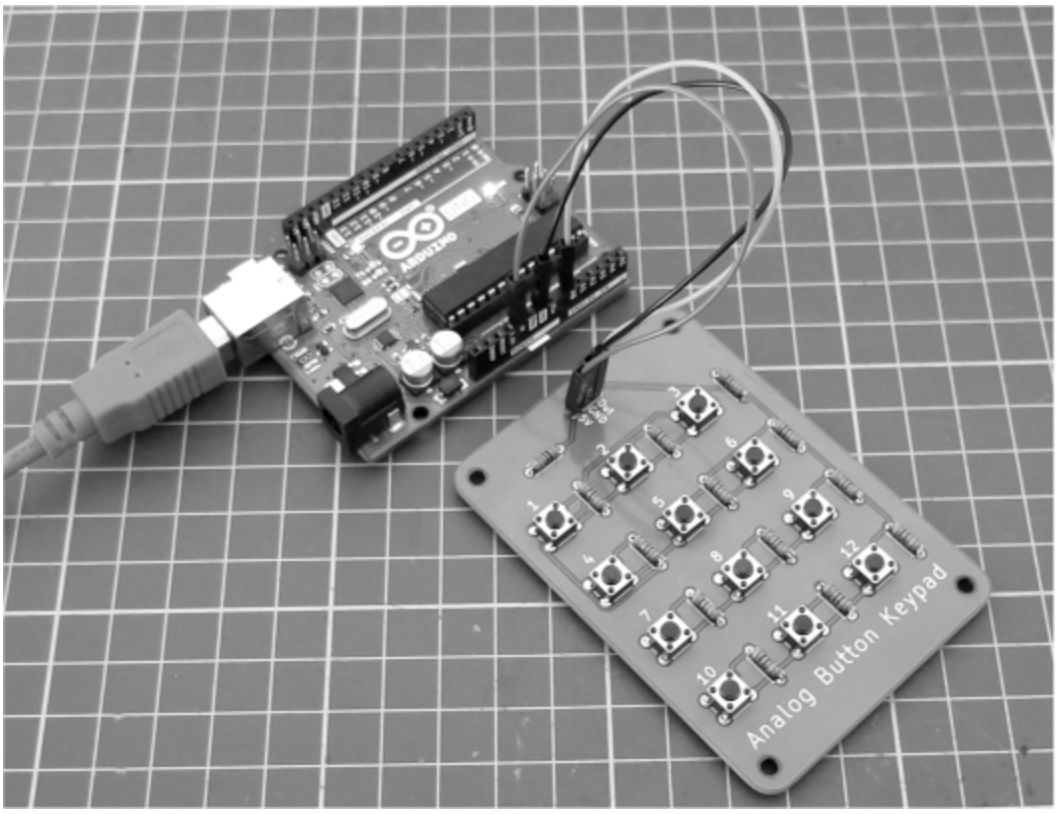


Figure 1-10: Keypad connected to the Arduino Uno

Next, determine the analog values for each button to insert into the library's source file by uploading the sketch from Project #1. Open the Serial Monitor and press each button in turn, noting the value with the monitor. You can use these figures to make ranges for the library for this project.

To simplify using the keypad, install the following analogkeypad library, whose three files are listed next. (For a refresher on writing and installing your own Arduino library, see Appendix A.) The following code is the source file; use the

sketch from Project #1 and the keypad for this project to determine your `ADCvalue` ranges.

```
// analogkeypad.cpp source file

#include "Arduino.h"
#include "analogkeypad.h"

analogkeypad::analogkeypad()
{
}

int analogkeypad::readKeypad()
{
    int adcValue;
    int button;
    adcValue = analogRead(A0);
    if (adcValue>1000) {button = 0;}
    else if (adcValue>941 && adcValue<944) {button
    else if (adcValue>863 && adcValue<868) {button
    else if (adcValue>790 && adcValue<795) {button
    else if (adcValue>716 && adcValue<720) {button
    else if (adcValue>643 && adcValue<648) {button
    else if (adcValue>570 && adcValue<573) {button
    else if (adcValue>493 && adcValue<496) {button
    else if (adcValue>413 && adcValue<418) {button
    else if (adcValue>325 && adcValue<329) {button
    else if (adcValue>231 && adcValue<235) {button
```

```
    else if (adcValue>129 && adcValue<133) {button
    else if (adcValue>13 && adcValue<18) {button =
    return button;
}
```

Here's the header file:

```
// analogkeypad.h header file

#ifndef analogkeypad_h
#define analogkeypad_h

#include "Arduino.h"

class analogkeypad
{
    public:
        analogkeypad();
        int readKeypad();
    private:
        int adcValue;
        int button;
};
#endif
```

Here's the *KEYWORDS.TXT* file:

```
Analogkeypad. KEYWORD1
readKeypad.   KEYWORD2
```

With the library installed, upload the matching sketch for Project #3:

```
// Project #3 - Reading 12 buttons from 1 analog

#include <analogkeypad.h>
❶ analogkeypad keypad;
void setup()
{
    Serial.begin(9600);
}

void loop()
{
  ❷ switch (keypad.readKeypad()) // Read button
    {
    ❸  case 0 : Serial.println("No button presse
        case 1 : Serial.println("Button 1 presse
        case 2 : Serial.println("Button 2 presse
        case 3 : Serial.println("Button 3 presse
        case 4 : Serial.println("Button 4 presse
        case 5 : Serial.println("Button 5 presse
        case 6 : Serial.println("Button 6 presse
        case 7 : Serial.println("Button 7 presse
```

```
          case 8 : Serial.println("Button 8 presse
          case 9 : Serial.println("Button 9 presse
          case 10 : Serial.println("Button 10 pres
          case 11 : Serial.println("Button 11 pres
          case 12 : Serial.println("Button 12 pres
      }
  delay(250);
  }
```

This sketch has the same results as Project #2 but allows for additional buttons in the main loop. It first includes the analogkeypad library ❶ and starts an instance of the keypad; next it retrieves the button number pressed via the library function ❷. The sketch then sends the button number to the Serial Monitor in the `switch…case` function ❸.

Once you have uploaded the sketch, open the Serial Monitor and press the buttons in turn. This should result in output like that shown in Figure 1-11.

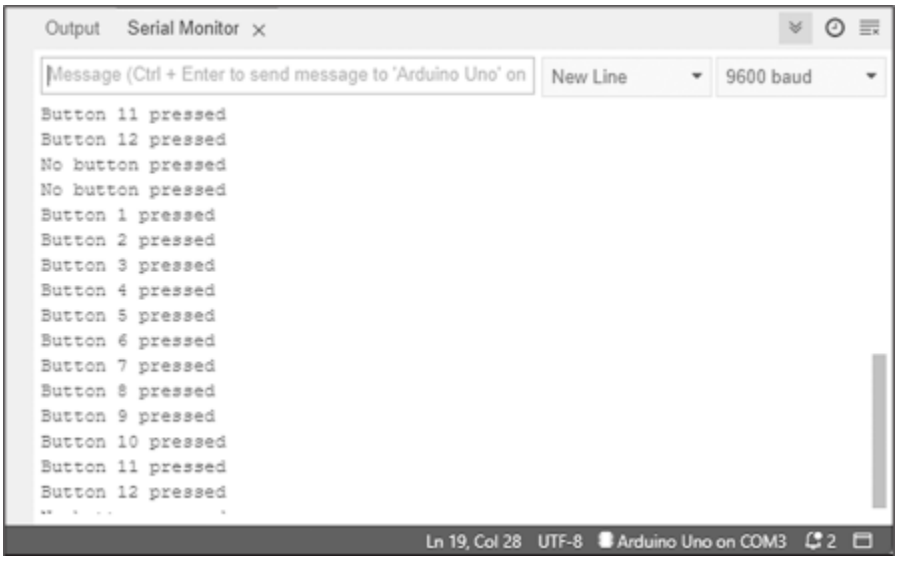


Figure 1-11: Example output from Project #3

You can use this sketch in your own projects, simply interrogating the keypad when required as you use the library and `readKeypad()` function.

NOTE

In your own projects that incorporate the sketches from this chapter, you can use Arduino types other than the Uno, as long as they have an analog input. If your board runs on 3.3 V instead of 5 V, use the Project #1 sketch to determine the analog values to use for your circuit before updating the custom function in Project #2 or the library used in Project #3.

Moving On

In this chapter, you learned to build a circuit that allows your Arduino to read the status of one of up to 12 buttons using only one analog input pin. In the next chapter, you'll learn another way to improve project efficiency: simultaneously reading and writing digital I/O pins using port manipulation.

2

PORT MANIPULATION



This chapter introduces *port manipulation*, the process of simultaneously addressing your Arduino's digital input and output pins to increase the speed of controlling or reading their statuses. Doing so reduces the *response time*, or the time required for a digital read or write to complete, thus increasing the sketch's speed of operation. Port manipulation also uses less code than normal digital write/read functions, reducing the size of your sketch, which comes in handy when you hit the upper limits of your Arduino's memory constraints.

In this chapter, you'll learn to:

- Use port manipulation to rapidly control digital outputs

- Control seven-segment LED displays
- Read multiple digital input pins simultaneously
- Read binary-coded decimal (BCD) switches

Along the way, you'll build a binary number display, a light-emitting diode (LED) blinking wave, and a single-digit LED die. The instructions in this chapter are for an Arduino Uno or compatible board, original Nano, or other Arduino-compatible with an ATmega328-series microcontroller.

Ports, Pins, and Registers

A *port* is a collection of sequential I/O pins. Each has a *register* (a section of memory inside the microcontroller), which is written to control the output status or read from if the pin status is set to an input. You use another register—the data direction register (DDR)—to set pins in a port as inputs or outputs.

Figure 2-1 shows the pins and the ports in which they are grouped. Port B consists of digital pins D13 through D8, port C consists of analog pins A5 through A0, and port D consists of digital pins D7 through D0.

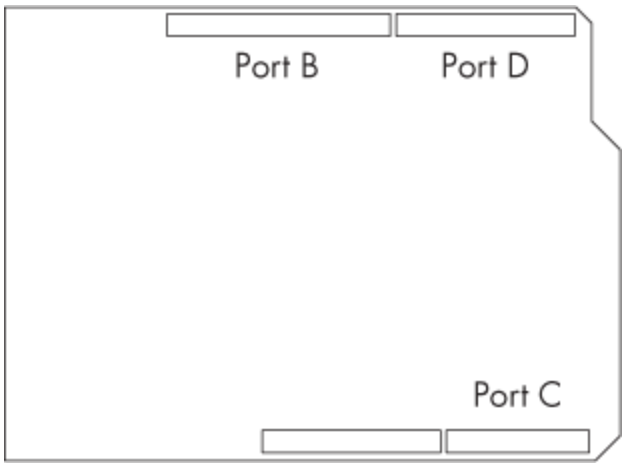


Figure 2-1: Ports on the Arduino Uno

To set a pin as an input or output, you don't use `pinMode()` in `void setup()` as you normally would. Rather, use the following simple syntax:

```
DDRx = Byyyyyyyy
```

`DDRx` is the data direction register, where *x* is the port (B, C, or D) and each *y* matches a pin in the port (from highest to lowest). The B in `Byyyyyyyy` indicates that *yyyyyyyy* is a binary number. For example, to set all of port D to digital outputs, you use the following line of code in `void setup()`:

```
DDRD = B11111111;
```

To set digital pins D13, D12, and D11 to inputs and digital pins D9, D8, and D7 to outputs, use the following code:

```
  DDRB = B000111; // Port B has six pins in total
```

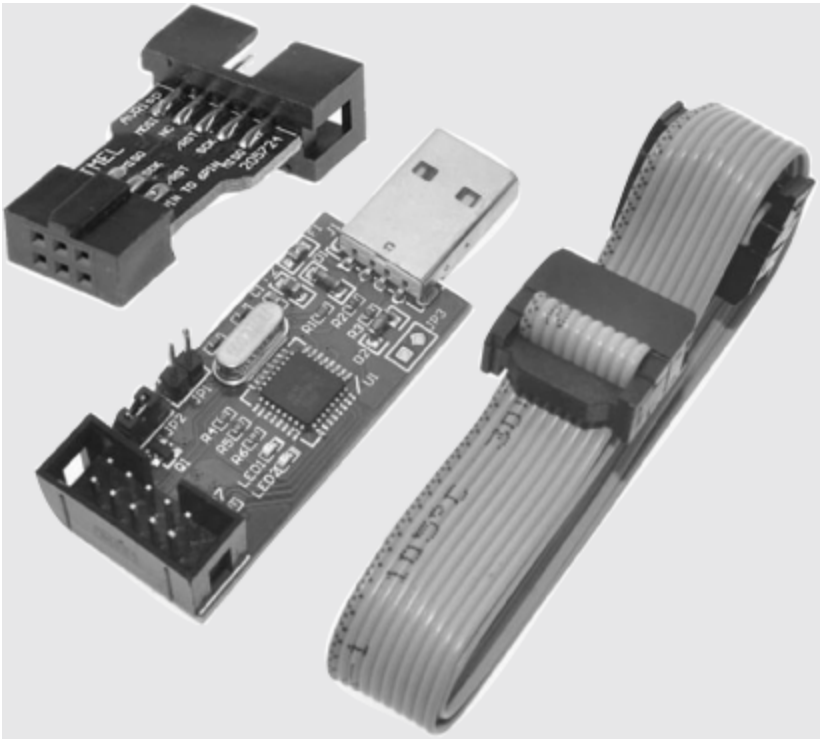
You can see how much sketch space this saves compared to multiples or loops of `pinMode()`.

EXERCISING CAUTION WHEN USING PORT MANIPULATION AND D1/D0 PINS

If you use port manipulation to control digital pins D1 and D0, your sketch may lose the ability to use the Serial Monitor in the IDE or communicate with other devices via the UART. In such cases, you may want to use SoftwareSerial to create a different pair of UART pins to communicate with devices such as GPS modules that use a UART-style interface, as demonstrated in Chapter 15 of *Arduino Workshop*, 2nd edition (No Starch Press, 2021), for example.

Furthermore, any external circuitry you've connected to digital pins D1 and D0 may be triggered when you upload a sketch—the highs and lows of the serial communication are the same as digital writes and reads. In this case, I

recommend you upload sketches with a USBasp programmer such as PMD Way part number 263343, shown here.



In addition to abbreviating your code and shortening the response time, port manipulation can save you hardware costs when controlling digital output pins. For example, in the past, you may have used a shift register IC to control an LED display. Port manipulation allows you to do the same using a spare bank of I/O pins instead.

Let’s put this theory into practice with some projects.

Project #4: Increasing Digital Output Pin Speed

In this project, you’ll practice using port manipulation by controlling eight digital outputs at once, with LEDs used to display output status.

You’ll need the following parts:

- An Arduino Uno or compatible board and USB cable
- Eight 1 kΩ, 0.25 W, 1 percent resistors
- Eight LEDs
- A solderless breadboard
- Male-to-male jumper wires

To minimize the number of part types used in this book and therefore reduce the cost of completing projects, this project uses the same 1 kΩ resistors from the previous chapter instead of the usual 560 Ω or thereabouts. The LEDs will still be plenty bright.

Assemble the circuit as shown in Figure 2-2 on the solderless breadboard, connecting the points labeled D7 through D0 and GND to the Arduino’s respective pins.

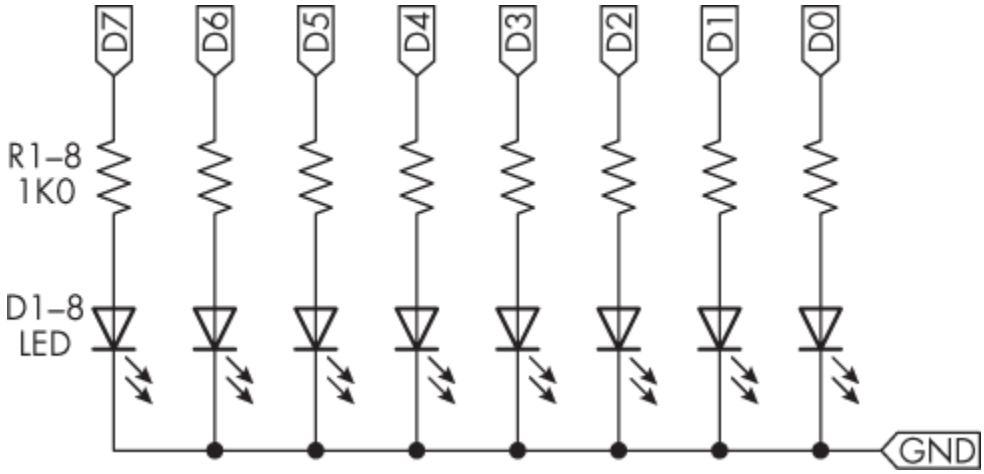


Figure 2-2: Schematic for Project #4

Enter and upload the following sketch to your Arduino:

```
// Project #4 - Fast digital outputs

void setup()
{
  ❶ DDRD = B11111111; // Set PORTD (digital 7 th
}

void loop()
{
 ❷ PORTD = B11111111;
    delay(500);
 ❸ PORTD = B00000000;
    delay(500);
}
```

Once the upload is complete, all the LEDs should blink on and off every half second. The pins on port D are set to outputs ❶ and all turned on ❷ and then off again ❸. This simple example shows how little code is required to accomplish a lot of digital pin control.

Keep the hardware for this project together, as you'll use it again in Project #5.

Speed Advantage

The Arduino environment uses a hardware abstraction layer that simplifies controlling the hardware in your Arduino board or compatible circuit with the help of software functions and classes. This code requires extra processing time, which normally slows down sketch execution. Port manipulation allows you to bypass the hardware abstraction layer, which greatly reduces the amount of code the CPU needs to run to accomplish the same task.

To see how much port manipulation speeds up I/O pin operation, this section shows how to use test equipment such as a frequency counter or a digital storage oscilloscope (DSO) to measure the frequency at which a port of digital pins turns on

and off. If you have an oscilloscope or a frequency counter, you can follow along.

First, test the pin speed with the usual Arduino sketch functions, using the sketch in Listing 2-1 with a digital pin connected to the oscilloscope.

```
void setup()
{
    for (int a = 0; a < 8; a++)
    {
        pinMode(a, OUTPUT);
    }
}

void loop()
{
    for (int a = 0; a < 8; a++)
    {
        digitalWrite(a, HIGH);
    }
    for (int a = 0; a < 8; a++)
    {
        digitalWrite(a, LOW);
    }
}
```

Listing 2-1 A pin speed test, which turns digital pins D7 to D0 on and off in sequence and then repeats

Figure 2-3 shows the results of this measurement: an average frequency of 15.15 kHz, which gives a time interval of around 0.000066 seconds (66 µs) to turn all eight digital pins on and off.

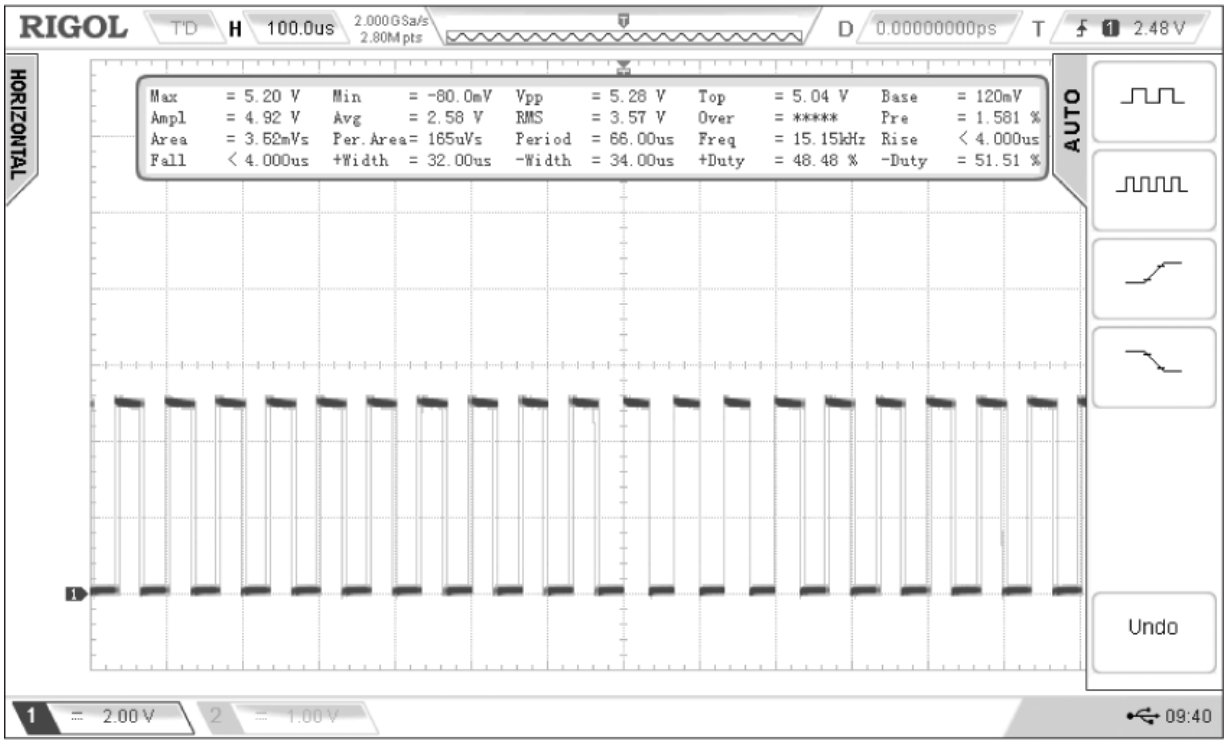


Figure 2-3: The results from running the Listing 2-1 test

Now repeat this test with the modified sketch in Listing 2-2, which uses port manipulation.

```
void setup()
{
```

```
    DDRD = B11111111; // Set PORTD (digital 7 th
}

void loop()
{
    PORTD = B11111111;
    PORTD = B00000000;
    PORTD = B11111111;
    PORTD = B00000000;
    PORTD = B11111111;
    PORTD = B00000000;
    PORTD = B11111111;
    PORTD = B00000000;
    PORTD = B11111111;
    PORTD = B00000000;
    PORTD = B11111111;
    PORTD = B00000000;
    PORTD = B11111111;
    PORTD = B00000000;
}
```

Listing 2-2: Using port manipulation to test pin speed

The `PORTD` commands in Listing 2-2 demonstrate turning the pins on and off long enough for the oscilloscope to capture the true speed of this operation. If there were only two `PORTD` commands (for on and off), the time measurement would

include the period between the pins being turned off and the loop starting over.

Figure 2-4 shows the results of this sketch on the oscilloscope.

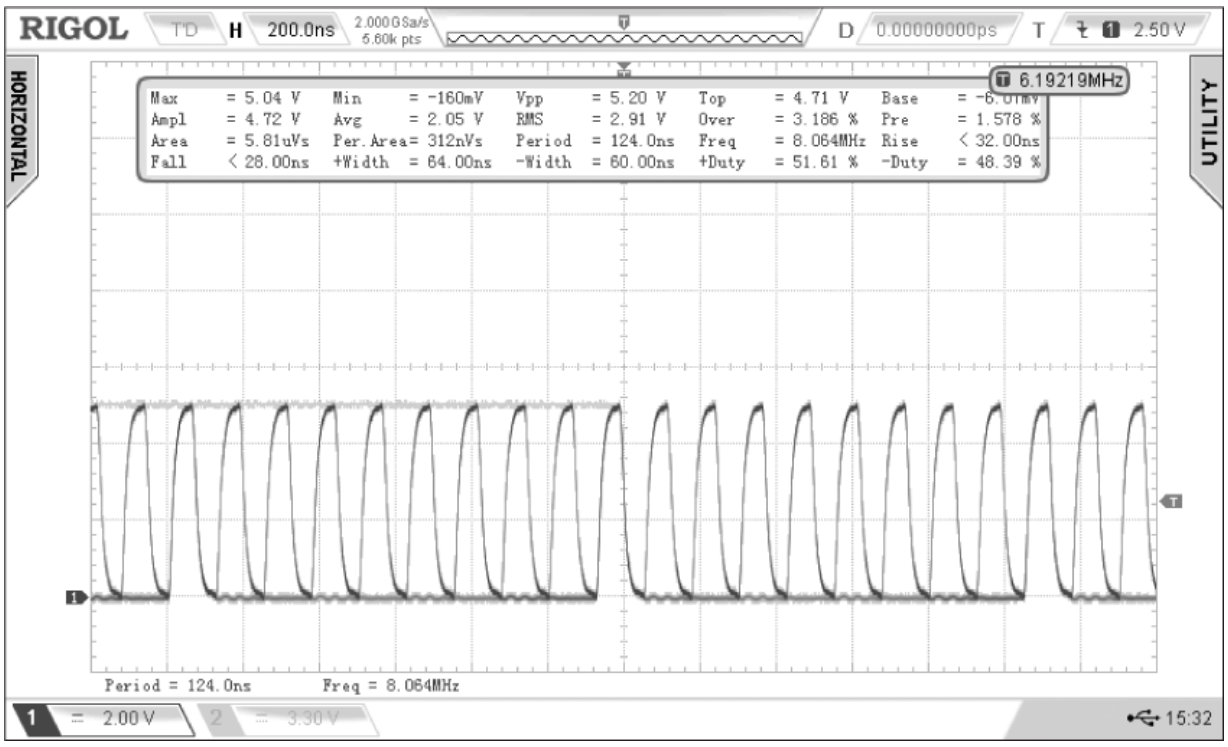


Figure 2-4: Results from the "fast" test

This second test resulted in an average frequency of 8.06 MHz, or a time interval of around 0.00000012406 seconds (0.12405 µs) to turn all eight digital pins on and off. On average, using port manipulation turned those pins on and off around 532 times faster than normal sketch functions.

You can see how much using port manipulation improves operation speed and reduces the amount of required code in your sketches. In the next project, we'll demonstrate these benefits of port manipulation by controlling LEDs for display purposes.

Project #5: Displaying Binary Numbers

In this project, you'll continue to consolidate your knowledge of port manipulation by using the method to display binary numbers with eight LEDs.

This project uses the same hardware as Project #4. Once you've uploaded the sketch, the LEDs should illuminate in turn to display binary numbers from 0 to 255, with the least-significant bit on the right side of the LEDs.

```
// Project #5 - Display binary numbers

void setup()
{
  ❶ DDRD = B11111111; // Set PORTD (digital 7 th
}

void loop()
{
```

```
      for (int a = 0; a < 256; a++)
      {
        ❷ PORTD = a;
          delay(250);
      }
  }
```

Once again, we set the pins on port D to outputs ❶. We then set `PORTD` to the increasing value of the integer `a`. As the port register contains 8 bits, we can assign an integer between 0 and 255 to the port so that the outputs match the binary equivalent of the number assigned to the register ❷.

Before moving on, let's practice mastering bit-shifting in port manipulation by turning LEDs on and off in one direction and then the other.

Project #6: Creating a Blinking LED Wave

In this project, using the hardware from Projects #4 and #5, you'll create a wavelike light pattern to emulate the lights across the bonnet of the famous vehicle K.I.T.T. from the 1980s television show *Knight Rider*. Once you've uploaded this sketch, you should be presented with your own version of K.I.T.T.'s iconic scrolling lights.

```
// Project #6 - Creating a blinking LED wave

void setup()
{
    DDRD = B11111111; // Set PORTD (digital 7 th
}

void loop()
{
  ❶ for (int k = 0; k < 8; k++)
    {
      ❷ PORTD = 1 << k;
         delay(100);
    }

  ❸ for (int k = 6; k > 0; –k)
    {
        PORTD = 1 << k;
        delay(100);
    }
}
```

This sketch has two loops: one to blink the LEDs from right to left and one to blink them in the opposite direction. We use bit-shifting to move the number 1 from right to left along the port register. We loop eight times, for eight LEDs ❶. The LED is

activated by assigning 1 to the port register ❷, which turns on the LED on digital pin D0 in the second run of the loop.

The value of `k` is 1, so we shift the bit one to the left (using `<<`) in the port register, turning on the second LED on digital pin D1. This process repeats until all eight LEDs have been turned on and off. The following list of port register bytes represents each event in this sequence of events:

```
00000001 // PORD = 1 << 0
00000010 // PORD = 1 << 1
00000100 // PORD = 1 << 2
00001000 // PORD = 1 << 3
00010000 // PORD = 1 << 4
00100000 // PORD = 1 << 5
01000000 // PORD = 1 << 6
10000000 // PORD = 1 << 7
```

The process then repeats in reverse in the second loop at ❸, beginning by putting a bit in the seventh position (`01000000`), and then continues along to bit 1. We don't need to put a bit in location 0 for this second loop, as that would illuminate the first LED twice in a row.

The next two projects demonstrate another convenient application for port manipulation: using seven-segment LED

displays, which are simply eight LEDs arranged to form a number with an optional decimal point.

Project #7: Controlling Seven-Segment LED Displays

People generally use an external shift register IC, such as the 74HC595, to display simple numerical information with seven-segment LED displays. If you have the digital I/O pins to spare, however, you can save money on parts by using the method in this project. This project also includes a quick review of bitwise arithmetic.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- Eight 1 kΩ, 0.25 W, 1 percent resistors
- A common-cathode seven-segment LED display (FND500-compatible)
- A solderless breadboard
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 2-5 on the solderless breadboard and connect the points labeled D7 through D0 and GND to the Arduino's respective pins.

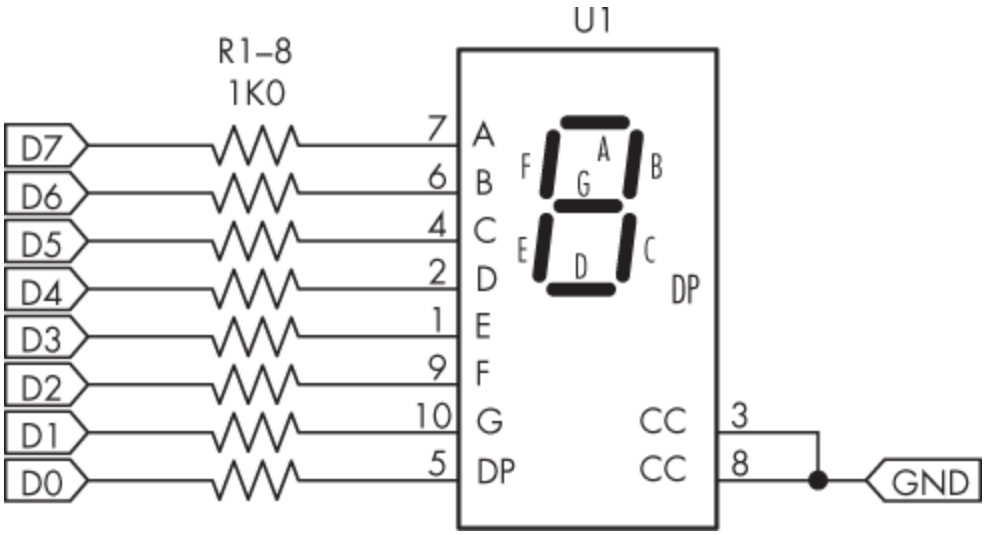


Figure 2-5: The schematic for Project #7

Once you've uploaded the sketch, the numbers 0 through 9 should display in turn and then display once more, this time with the decimal point in the LED display turned on.

Let's see how this works:

```
  // Project #7 - Controlling seven-segment LED di

❶ int digits[] = {B11111100, // 0
                  B01100000, // 1
                  B11011010, // 2
                  B11110010, // 3
                  B01100110, // 4
                  B10110110, // 5
                  B10111110, // 6
                  B11100000, // 7
```

```
                  B11111110, // 8
                  B11110110}; // 9
void setup()
{
  ❷ DDRD = B11111111; // Set PORTD (digital 7 th
}

void loop()
{
    for (int a = 0; a < 10; a++)
    {
      ❸ PORTD = digits[a];
        delay(250);
    }

    for (int a = 0; a < 10; a++)
    {
      ❹ PORTD = digits[a]|B00000001; // Activate
        delay(250);
    }
}
```

The sketch defines an array with 10 elements ❶, each including a binary number representing the LEDs of the display needed to power on or off to display each digit. From the schematic in Figure 2-5, you can see that the display is wired so its first pin

(A) is connected to digital pin D7, and so on. In this project, port D is set at ❷. Displaying the required digit is a simple matter of allocating the required array element (0 to 9) to `PORTD` at ❸ and ❹.

The loop ❹ turns the decimal point on. Bit 0 in each array element represents the decimal point LED in the display, so the sketch needs to change the bit in bit 0 using the OR function (`|`) in bitwise arithmetic. When you use OR to compare 2 bits, the result is 1 if either bit is 1 or if both are 1.

```
1|0 = 1 // 1 or 0 = 1
0|1 = 1 // 0 or 1 = 1
1|1 = 1 // 1 or 1 = 1
0|0 = 0 // 0 or 0 = 0
```

Therefore, you can add the decimal point to a numerical byte using the following (in the sketch, the numerical byte is set to display the number 1 and the decimal point):

```
B01100000 | // Byte to display digit 1
B00000001 = // "or" decimal point bit
B01100001 // Gives us digit 1 with decimal point
```

In the future, if you're using the LED display in your own projects with port manipulation, you can take advantage of the following custom function to display numbers with or without the decimal point. You can also test it by adding it to Project #7.

```
void display(int n, boolean point)
{
    if (point == false) // No decimal point
    {
        PORTD = digits[n];
    }
    else if (point == true) // Decimal point
    {
      PORTD = digits[n]|B00000001;
    }
}
```

If you want to turn off a particular output that you know is currently on without affecting the other pins, use the bitwise AND function &. When a bit is compared with & 0, the result is 0 if the bit is 1. For example:

```
1&0 = 0 // 1 or 0 = 0
0&1 = 0 // 0 or 1 = 0
```

To turn off the first and last output when all pins are on, you'd use the following code, which leaves bits 6 through 1 on, with 7 and 0 off:

```
PORTx = B11111111 &
        B01111110;
```

Now that you have the LED display circuit connected, let's make an electronic die.

Project #8: Creating an Electronic Die

In this project, you'll create an electronic die that generates a random number between one and six to mimic rolling a physical die.

This project uses the same hardware as Project #7. Upload the sketch; the display should rapidly scroll through the numbers and then slow down bit by bit until it stops.

```
// Project #8 - Single-digit electronic die

int digits[] = {B11111100, // 0
                B01100000, // 1
                B11011010, // 2
                B11110010, // 3
```

```
                  B01100110, // 4
                  B10110110, // 5
                  B10111110, // 6
                  B11100000, // 7
                  B11111110, // 8
                  B11110110}; // 9

❶ void display(int n, boolean point)
  {
      if (point == false)
      {
          PORTD = digits[n];
      } else if (point == true)
      {
          PORTD = digits[n] | B00000001;
      }
  }

  void displayRandom(int _delay)
  {
      display(random(1, 7), false); // Display the
      if (_delay > 0)
      {
          delay(_delay); // Hold the display on fo
      }
      else if (_delay == 0)
      {
          do // The delay entered was 0, hold the
```

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

void setup()
{
    DDRD = B11111111; // Set PORTD (digital 7 th
 ❷ randomSeed(analogRead(0)); // Seed the randor
}

void loop()
{
    int a;
    // Cycle the LEDs around for effect
 ❸ for (a = 0; a < 100; a++)
    {
        displayRandom(50);
    }
    // Display numbers with increasing delay
 ❹ for (a = 1; a < 10; a++)
    {
        displayRandom(a * 100);
    }
    // Stop at the final random number and LED
 ❺ displayRandom(0);
}
```

The sketch is a combination of new code and the functions that were used in Project #7. The sketch first uses the custom function from that project to display the digits ❶ and then seed the random number generator ❷. It rapidly displays 100 random numbers for visual effect ❸ and then displays the next 10 random numbers with an increasing time delay between each number ❹. Finally, after one last random number is displayed ❺, everything stops. To display another random number, press the reset button.

Project #9: Reading Multiple Digital Inputs Simultaneously

You can also read digital input pins rapidly using the port-style operations, rather than by using multiple `digitalRead()` functions. Since each port returns 8 bits of data (including ports B and C, which have only six physical pins), we can simply set the required port to inputs and then assign the value of a port to a variable. The value is an 8-bit number that represents the status of the input pins.

Once inputs have been set using the `DDRx` function in `void setup()`, just assign the port to a variable to read the status of the pins. For example, to read the status of port D once set to inputs, use this:

```
  int port = PIND; // Value of pins on PORTD is st
```

This project demonstrates how to read an entire port at once and then display the status of that port using another port of LEDs.

You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- Four 1 kΩ, 0.25 W, 1 percent resistors
- Four 10 kΩ, 0.25 W, 1 percent resistors
- Four LEDs
- Four tactile buttons
- A solderless breadboard
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 2-6 on the solderless breadboard and connect the points labeled D7 through D4, D13 through D10, 5V, and GND to the Arduino's respective pins.

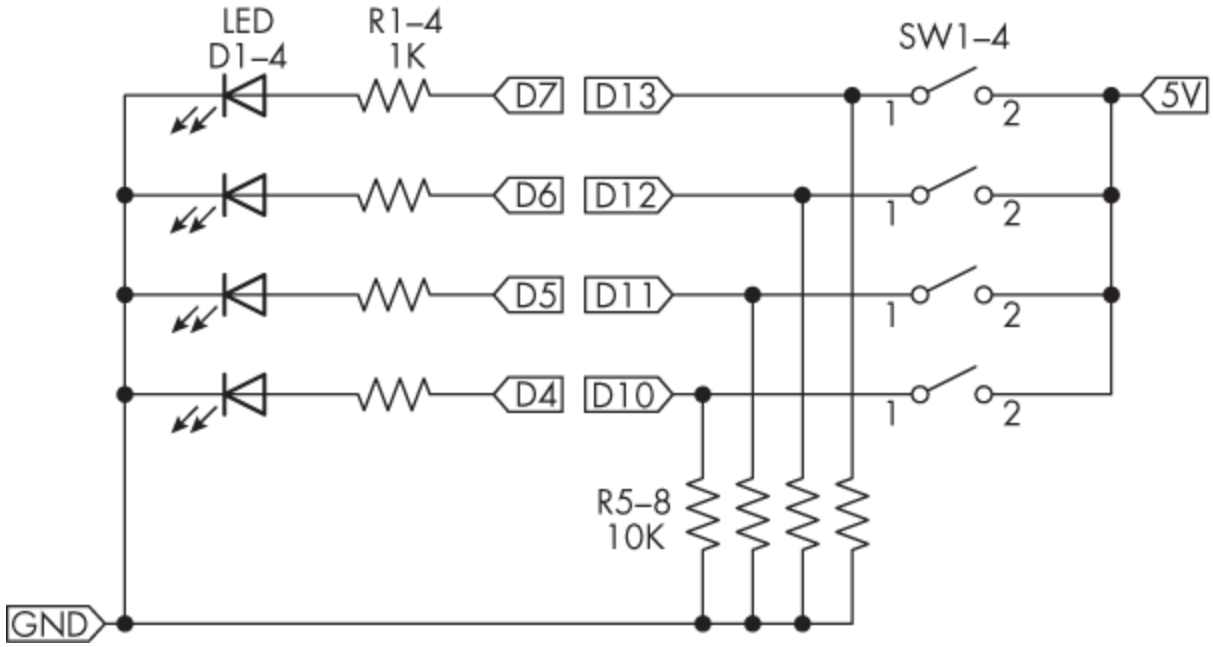


Figure 2-6: The schematic for Project #9

Once you've uploaded the sketch, pressing a button should illuminate the matching LED.

Let's see how this works:

```
// Project #9 - Simultaneous reading of Arduino

void setup()
{
  ❶ DDRB = B00000000; // Set PORTB (digital 13 t
  ❷ DDRD = B11111111; // Set PORTD (digital 7 th
}

void loop()
```

```
  {
    ❸ PORTD = PINB<<2; // Bit-shift PORTB value as
  }
```

This sketch demonstrates the code minimalization made possible using port manipulation: you can read the status of up to eight buttons in one line, instead of using eight separate `digitalRead()` functions. After setting the pins' status ❶, the sketch assigns the value of the pins (`PINB`) to port D, which sets the outputs as instructed ❷.

Next, the code bit-shifts the value of `PINB` 2 bits to the left ❸. Port B has only six physical pins but still returns an 8-bit number when read. The two most significant bits are 0, and the final six represent the status of D13 to D8. Therefore, this project moves the `PINB` value to the left so the bits match port D for the output control.

Binary-Coded Decimal Switches

The form of reading digital inputs demonstrated in Project #9 lends itself well to input devices with more than one output, such as rotary switches with BCD outputs representing the binary values of numbers 0 through 9. BCD switches are a

convenient method of accepting user input for setting options, values, or other project requirements where the user inputs a number between 0 and 9.

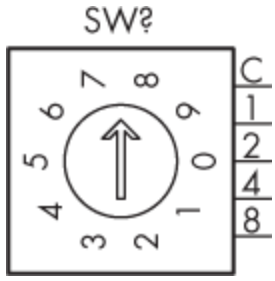


Figure 2-7: The schematic symbol for a typical BCD switch

BCD switches offer 10 position selections and have 4 output pins easily readable with 4 digital input pins. The fifth pin is commonly connected to the 5V pin, which allows the current to flow through to the switched output pins. Figure 2-7 shows the schematic symbol for a typical BCD switch.



Figure 2-8: A push-wheel BCD switch

Examples of BCD switches include the retro-style “push-wheel” switches, as shown in Figure 2-8, and rotary switches, as shown in Figure 2-9.



Figure 2-9: A rotary BCD switch

For performing simple BCD experiments, it’s easier to use the rotary BCD switch shown in Figure 2-9, as it doesn’t require any soldering.

With any switch, users must be careful to move the lever or pointer to the correct position. For example, if you change a rotary switch and leave the pointer between two digits, it will not return a correct value. The next project demonstrates how to implement these switches with port manipulation.

Project #10: Reading BCD Switches

This project demonstrates reading a BCD switch by using port manipulation to return the value in both binary and decimal to

the Serial Monitor, allowing you to receive numerical user input simply and efficiently.

You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- A BCD rotary or push-wheel switch
- A solderless breadboard
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 2-10 on the solderless breadboard and connect the points labeled 5V and D11 through D8 to the Arduino's respective pins.

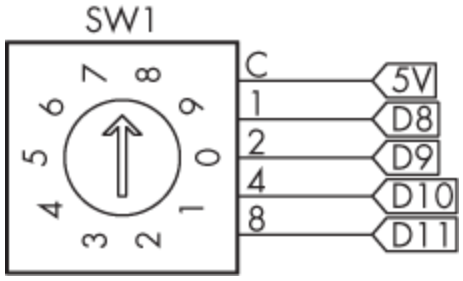


Figure 2-10: The schematic for Project #10

Enter and upload the sketch and then open the Serial Monitor. You should see the value of the switch's setting displayed in both binary and decimal, as shown in Figure 2-11.

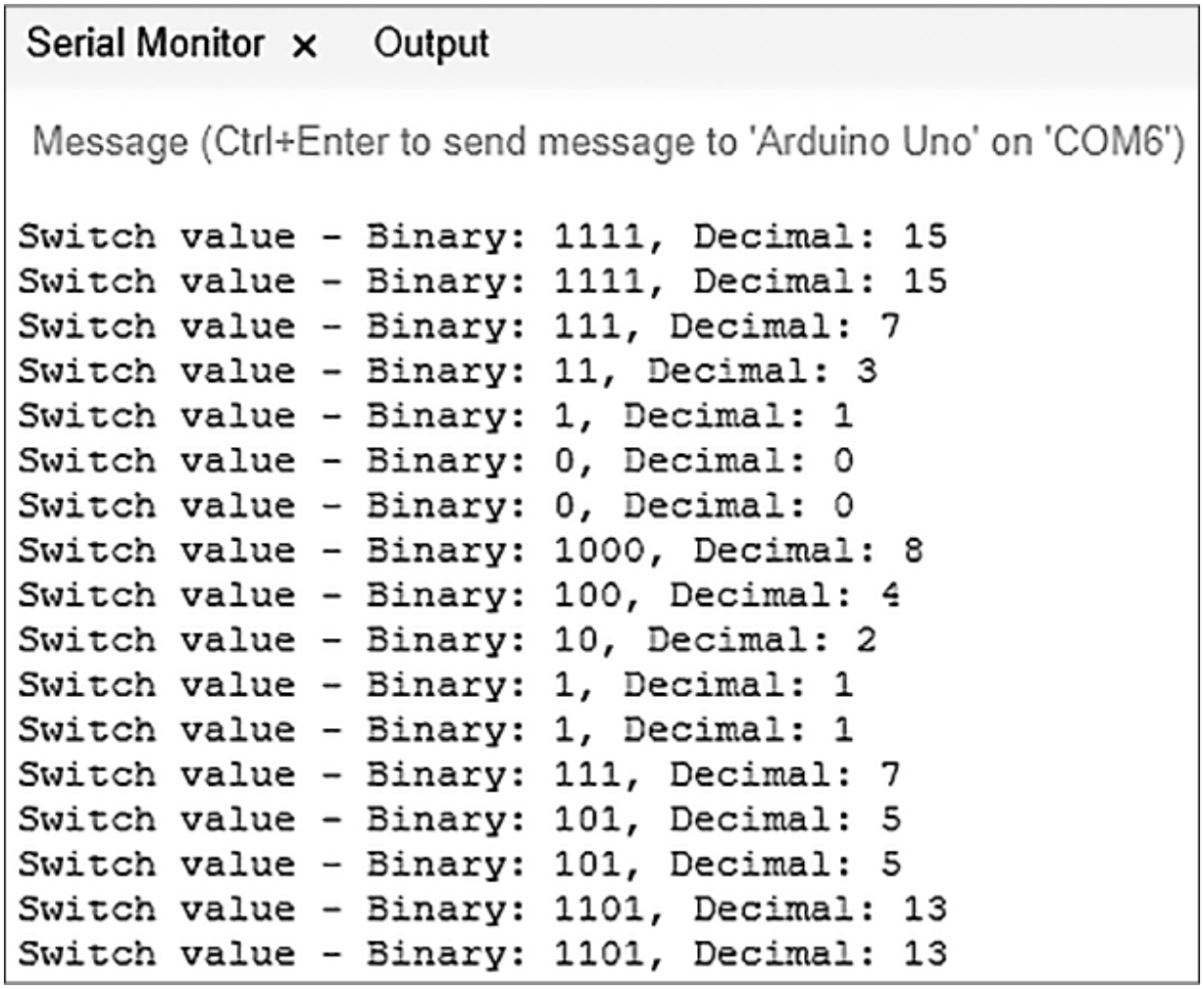


Figure 2-11: Example output for Project #10

Let's see how this works:

```
// Project #10 - Reading BCD switches

void setup()
{
  ❶ DDRB = B11110000; // Set PORTB (digital 11 th
    Serial.begin(9600);
}
```

```
void loop()
{
    Serial.print("Switch value - Binary: ");
  ❷ Serial.print(PINB,BIN);
    Serial.print(", Decimal: ");
  ❸ Serial.println(PINB);
    delay(250);
}
```

The code sets pins D11 through D8 to inputs and sets D12 and D13 to outputs ❶, since it doesn't want to return the values of pins D12 and D13. It then simply returns the value of port B to the Serial Monitor in binary ❷ and decimal ❸ forms. The binary value should reflect the actual signals from the switch back to the Arduino's input pins, while the decimal is the integer equivalent of the same signals.

Moving On

This chapter showed you how to work more quickly and efficiently with your Arduino's digital input and output pins by using port manipulation, controlling many LEDs without external ICs and receiving settings without needing to use a display or multiple switches. In the next chapter, you'll learn

how to use the physically smaller and cheaper ATtiny series of microcontrollers for less complex Arduino projects.

3

USING ATTINY MICROCONTROLLERS



In your Arduino journey so far, you've likely most often used the microchip ATmega328P-PU. For smaller projects, however, you can save money and power by using smaller microcontrollers such as the ATtiny85. In this chapter, you'll learn how to configure the Arduino IDE for ATtiny microchips.

After setting up the Arduino IDE for your ATtiny85 and flashing an LED to demonstrate that the toolchain is working, you'll learn to:

- Understand the ATtiny pin references for the Arduino environment and the functions available for ATtiny use

- Add a reset button to ATtiny circuits
- Use port manipulation with ATtiny
- Change the ATtiny operating speed

You'll also construct a quick-read thermometer and an Arduino programming shield that lets you easily upload code to ATtiny85 microcontrollers.

The ATtiny85 Microcontroller

Figure 3-1 shows the compact ATtiny85 microcontroller.

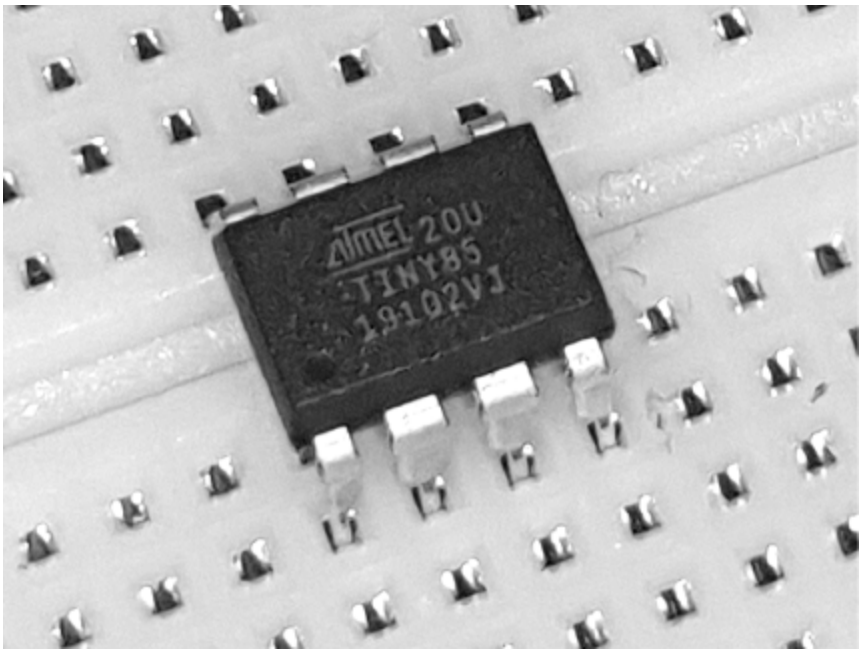


Figure 3-1: The ATtiny85 microcontroller in through-hole packaging

In addition to size, there are several important differences between the ATtiny85 and the usual ATmega328P-PU

microcontrollers that affect their use in the Arduino environment, as demonstrated in Figure 3-2 and Table 3-1.

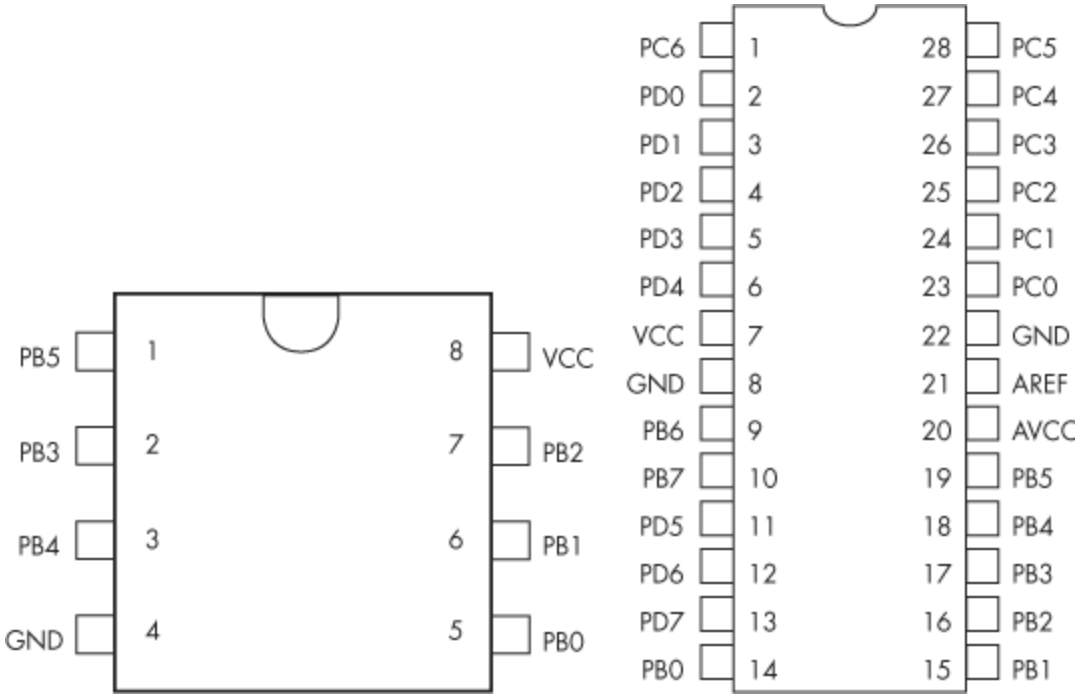


Figure 3-2: Pin diagrams for the ATtiny85, left, and the ATmega328P-PU, right.

Table 3-1: Specifications for the ATtiny85 and the ATmega328P-PU

	ATtiny85	**ATmega328P-PU**
Maximum processing speed	20 MHz	16 MHz

	ATtiny85	ATmega328P-PU
Number of digital pins	Up to 5	14
Number of analog input pins	Up to 3	6
Flash memory	8KB	32KB
Static random access memory (SRAM)	512 bytes	2KB

This table helps you determine which electronic components you can realistically use with your ATtiny85. In short, the ATtiny lends itself to projects with a limited budget that require few input and output pins but that could benefit from increased energy efficiency.

ATtiny Chips with the Arduino IDE

The High-Low Tech group at the Massachusetts Institute of Technology first created the code to use ATtiny85

microcontrollers with the Arduino IDE. Before you implement this code, however, you'll need to install ATtiny support on the Arduino IDE.

With an ATtiny85 on hand, open the Arduino IDE and then select **File** ▸ **Preferences**. Once the Preferences dialog appears, click the small button to the right of the Additional Boards Manager URLs field. The Additional Boards Manager URLs dialog will appear, as shown in Figure 3-3.

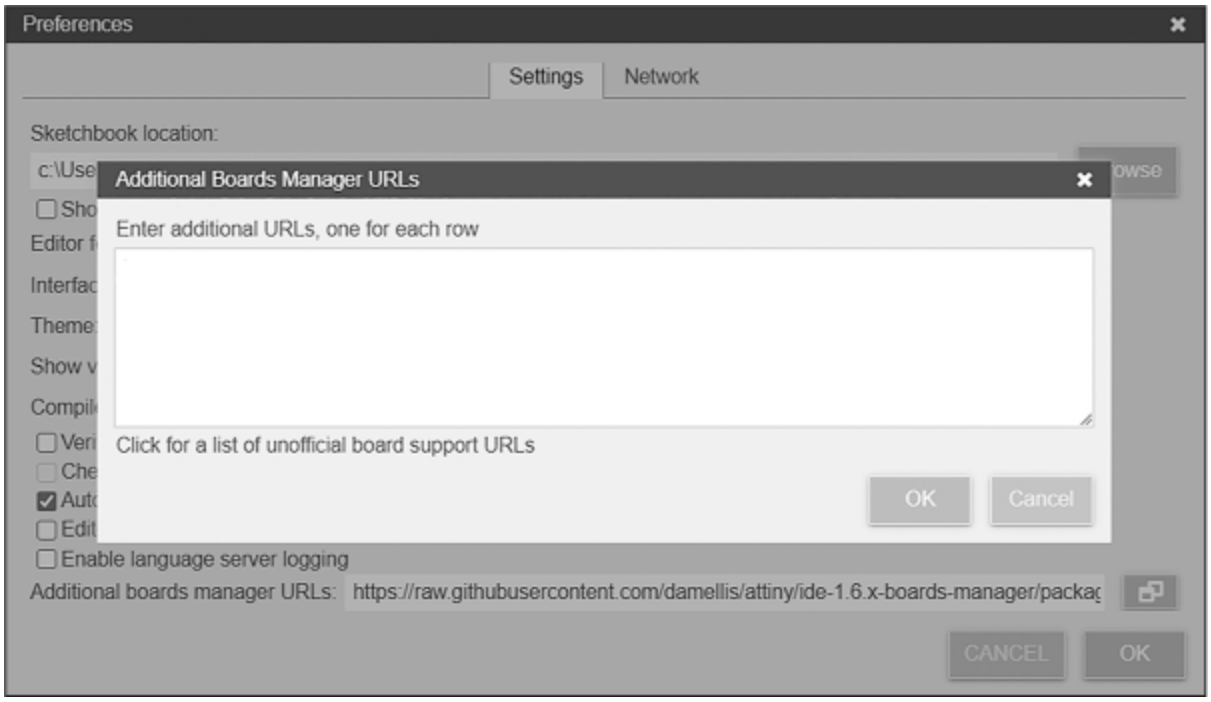


Figure 3-3: The Arduino IDE Additional Boards Manager URLs dialog

Enter the following URL into the field. If you already have other URLs in the box left over from previous projects, place a comma

at the end of the last URL and enter this new line after the comma:

```
  https://raw.githubusercontent.com/damellis/attiny
```

Click **OK** to close the additional URL dialog and then click **OK** to close the Preferences dialog.

Close and reopen the IDE. If your computer is not connected to the internet, connect it now. Select **Tools** ▸ **Boards Manager**. Once the Boards Manager opens on the left side of the IDE, enter **attiny** in the search box.

The ATtiny package will appear, as shown in Figure 3-3. Click **Install** and then wait a few moments for the installation to complete. You can see the progress in the IDE output window at the bottom of the screen:

```
  Downloading packages
  Attiny:avr@1.0.2
  Installing attiny:avr@1.0.2
  Configuring platform
  attiny:@1.0.2 installed
```

Finally, check that the installation was successful by selecting **Tools** ▸ **Board** ▸ **attiny** ▸ **ATtiny25/45/85**, as shown in Figure 3-4.

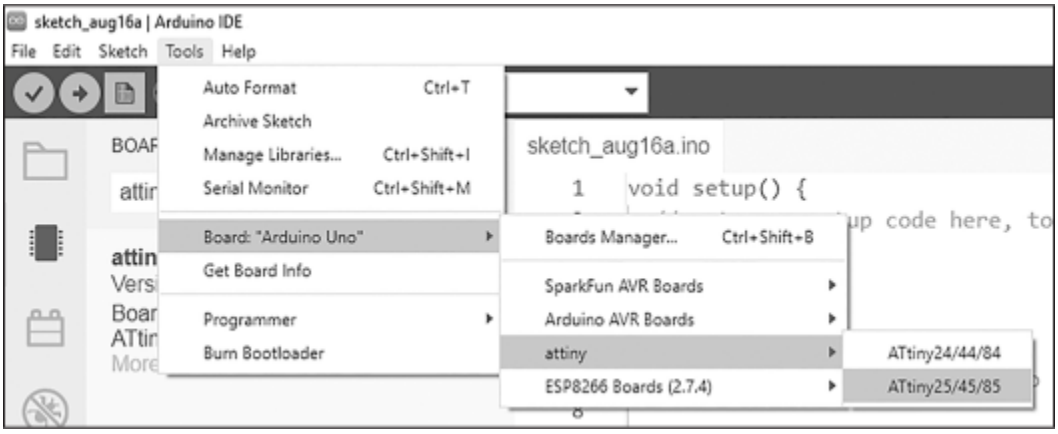


Figure 3-4: Board menu in which the ATtiny85 is now available

Now that you've configured the IDE, you'll need to configure the *hardware programmer*, or the interface between the computer and the ATtiny85, using an Arduino Uno or compatible board. Open the Arduino IDE and select **File** ▸ **Examples** ▸ **11.ArduinoISP** ▸ **ArduinoISP**, as shown in Figure 3-5.

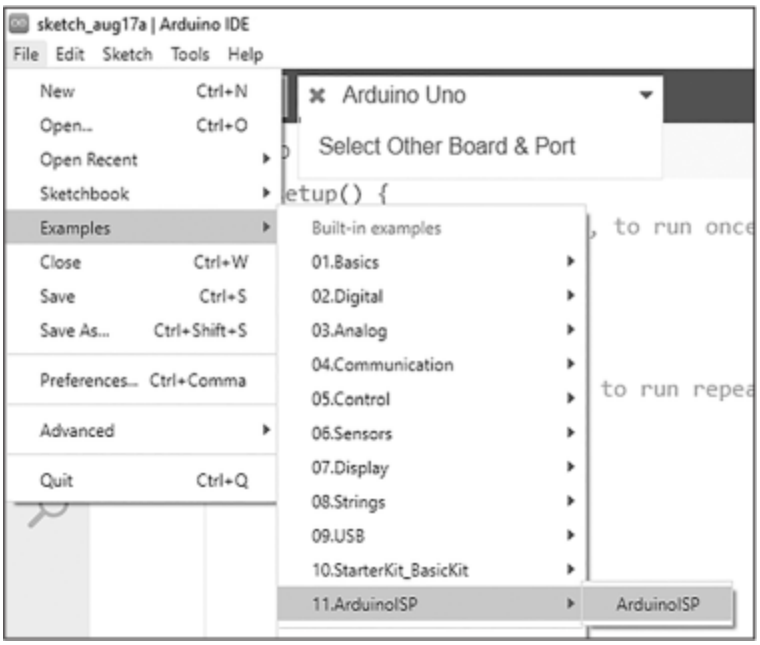


Figure 3-5: ArduinoISP sketch selection

After you upload this sketch to your Arduino Uno or compatible board, the board will act as a hardware programmer. You can always upload other sketches to the Uno, but if you want to upload a sketch to your ATtiny, you will need to upload the ArduinoISP sketch beforehand on the Arduino.

Let's check that your hardware and software environment is functioning with a typical sketch that blinks an LED.

Project #11: Building the "Hello, World" of Hardware

This project won't cause your ATtiny85 to print "Hello, world" to a screen but rather does the hardware equivalent: flashing an LED to demonstrate that the toolchain is working.

You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- An ATtiny85 microcontroller
- One 1 kΩ, 0.25 W, 1 percent resistor
- One 10 µF, 16 V electrolytic capacitor
- One LED
- A solderless breadboard
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 3-6 on your solderless breadboard and connect the points labeled 5V, GND, RESET, and D10 through D13 to the Arduino's respective pins.

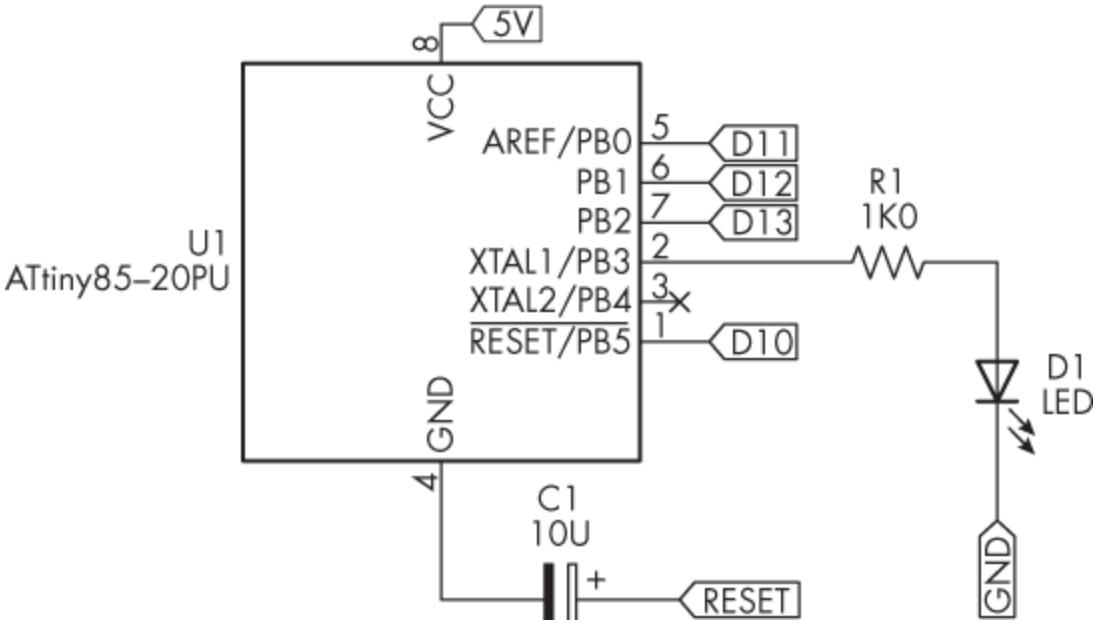


Figure 3-6: The schematic for Project #11

The connections to the Arduino's SPI bus via D10 through D13, as well as the connection between the 10 µF capacitor and the Arduino's reset pin, are required only for uploading each ATtiny project. Remove them once you've finished uploading the sketch.

Next, you'll change the IDE settings to accommodate the hardware changes. Open the IDE and change the board type to **ATtiny25/45/85**, as was shown in Figure 3-4. Select **Tools** ▸ **Clock** ▸ **Internal 1 MHz**. Finally, change the programmer by selecting **Tools** ▸ **Arduino as ISP**.

Now enter and upload the Project #11 sketch. The LED should start blinking with a delay of about one second, confirming that you have a working ATtiny development environment.

```
// Project #11: Build the "Hello, world" of hardv

void setup()
{
  ❶ pinMode(3, OUTPUT);
}

void loop()
{
  ❷ digitalWrite(3, HIGH);
```

```
    delay(1000);
  ❸ digitalWrite(3, LOW);
    delay(1000);
}
```

The physical pin 2 on the ATtiny85 is digital pin 3 in the Arduino environment, so the sketch declares pin 3 as an output ❶, turns it on ❷, and then turns it off after a delay ❸.

I'll now introduce three things you need to know before building the next project: the differences between Arduino Uno and ATtiny pin references, how to add a reset button to your ATtiny circuit, and the Arduino functions that are available to the ATtiny versions of Arduino projects.

The Arduino Uno vs. ATtiny85 Pin References

The reference numbers in the Arduino sketches for physical pins on the ATtiny85 differ from those you're used to with typical Arduino Uno and compatible boards. Thus, when you're making your own ATtiny85-based projects using the Arduino environment, you'll need to refer to Table 3-2 to determine the

ATtiny85's physical pins and their matching pins in the Arduino Uno environment.

Table 3-2: ATtiny85-to-Arduino Pin Reference

ATtiny physical pin	Arduino (1)	Arduino (2)
2	Digital pin 3 (D3)	Analog pin 3 (A3)
3	Digital pin 4 (D4)	Analog pin 2 (A2)
5	Digital pin 0 (D0)	Pulse-width modulation (use `analogWrite(0)`)
6	Digital pin 1 (D1)	Pulse-width modulation (use `analogWrite(1)`)
7	Digital pin 2 (D2)	Analog pin 1 (A1)

The HIGH and LOW signals that appear on D0 through D2 while a sketch is uploading will trigger any connected external items, such as relays. In these cases, it's better to upload the sketch to the microcontroller and then insert it into the final project.

As the ATtiny85 shares the same type of basic architecture with the ATmega328-type microcontrollers, you can use port manipulation as described in Chapter 2 to control the I/O pins. The ATtiny's I/O pins are on the least 4 bits of `PORTB` (bits 3 to 0) and match physical pins 2, 7, 6, and 5. Just leave the higher 4 bits as 0 with your `DDRB` and `PORTB` functions.

Adding Reset Buttons to ATtiny85 Circuits

As with other Arduino and compatible boards, it's possible to add a reset button to your ATtiny85 circuits. You'll use this in the next project, and it may come in handy for easily restarting future projects.

An ATtiny85 reset button connects physical pin 1 and GND, along with a 10 kΩ pull-up resistor, as the circuit resets when pin 1 is connected to GND. Figure 3-7 shows the schematic.

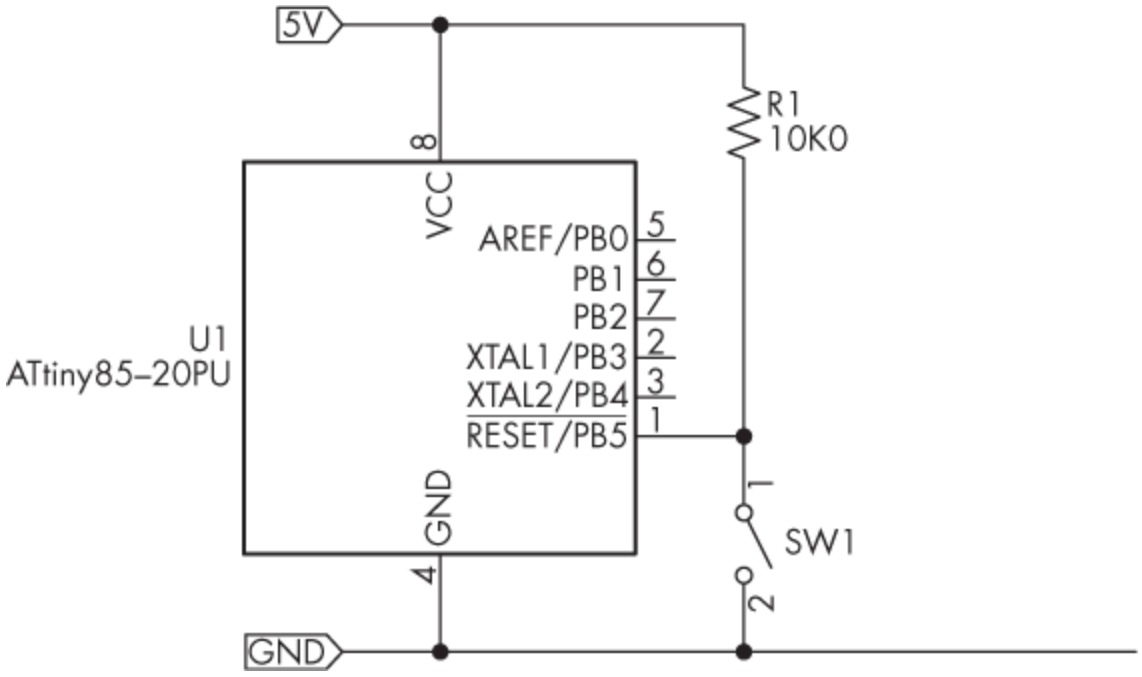


Figure 3-7: The reset button schematic for the ATtiny85

When designing circuits that won't connect to a host Arduino board during operation, place a 0.1 µF capacitor across the 5V and GND lines to maintain a smoother power supply.

Arduino Functions Available for the ATtiny85

If you're familiar with the Arduino environment and want to get straight into your own ATtiny-based projects, note that there are fewer Arduino functions available for the ATtiny85 than on larger microcontrollers, but there are still plenty:

`analogRead()` Tells the ADC to return a value between 0 and 1,023 representing a voltage range of 0 to 5 V DC

`analogWrite()` Generates PWM output

`delay()` Pauses operation of the sketch (in milliseconds)

`delayMicroseconds()` Pauses operation of the sketch (in microseconds)

`digitalWrite()` Turns digital output pins on or off

`digitalRead()` Reads the status of digital input pin

`micros()` Returns the number of microseconds since the sketch started operation

`millis()` Returns the number of milliseconds since the sketch started operation

`pinMode()` Sets the status of the digital pins

`pulseIn()` Returns the length of a pulse measured on a digital input

`shiftOut()` Shifts a byte of data out of a digital pin

If you're unfamiliar with any of these functions, you can review my book *Arduino Workshop*, 2nd edition, or visit the Arduino language reference at *https://www.arduino.cc/reference/en/*.

You may also use various Arduino libraries when working with the ATtiny85. However, if they're designed to access GPIO pins, you'll need to modify the library code to update them for ATtiny85 pin references. Some Arduino libraries may also require more memory than the ATtiny85 provides.

Project #12: Creating a Quick-Read Thermometer

In this project, you'll use the popular TMP36 temperature sensor to create a thermometer that records and displays the temperature in three predetermined ranges: too cold, just right, and too hot. This also demonstrates how to use the analog input and digital output pins on your ATtiny85 in the Arduino environment.

You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- An ATtiny85 microcontroller
- Three 1 kΩ, 0.25 W, 1 percent resistors
- One 10 kΩ, 0.25 W, 1 percent resistor
- One tactile button

- One 0.1 µF ceramic capacitor
- Three LEDs
- A TMP36 analog temperature sensor
- A solderless breadboard
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 3-8 on the solderless breadboard. This schematic shows only the final product, but you'll first need to wire the connections to upload the sketch—you can refer to Project #11 to refresh your memory of how to do this. Use your Arduino Uno as a quick 5 V power supply.

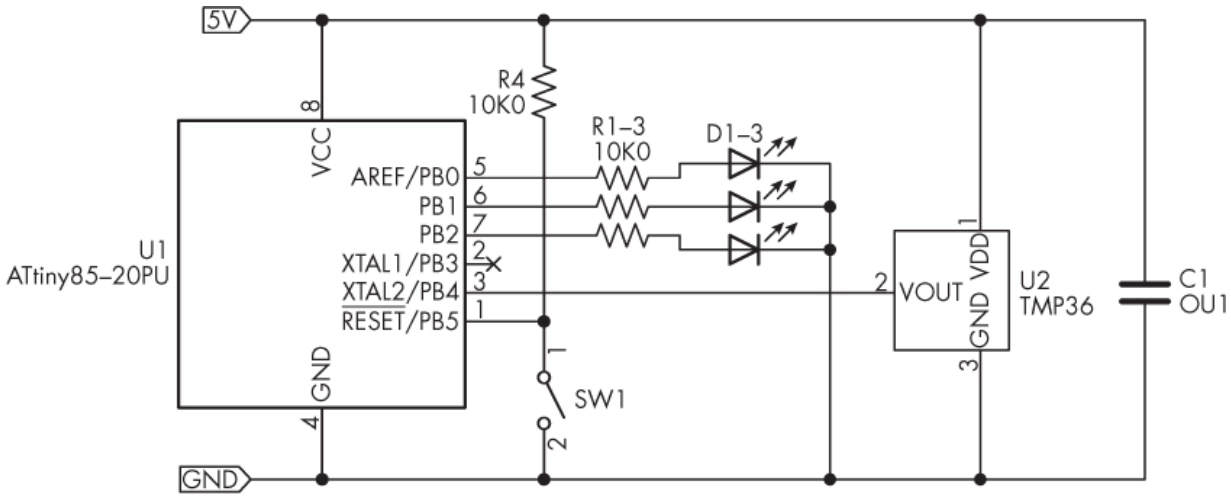


Figure 3-8: The schematic for Project #12

Once you've uploaded the sketch, remove the wires from the Arduino Uno's D10 through D13 pins and the RESET pin, but leave the 5V and GND to power the thermometer. When the

sketch starts running, one of three LEDs should depict the temperature range set in the sketch.

Let's see how this works:

```
  // Project #12 - A "quick-read" thermometer

  // Define the pins that the LEDs are connected t
❶ #define HOT 2
  #define NORMAL 1
  #define COLD 0

  float voltage = 0;
  float celsius = 0;
❷ float hotTemp = 25;
  float coldTemp = 15;
  float sensor = 0;

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

```
void loop()
{
    // Read sensor and convert result to degrees
 ❹ sensor = analogRead(2);
 ❺ voltage = (sensor * 5000) / 1024;
    voltage = voltage - 500;
    celsius = voltage / 10;
    // Act on temperature range
 ❻ if (celsius < coldTemp)
    {
        digitalWrite(COLD, HIGH);
        delay(250);
        digitalWrite(COLD, LOW);
    }
 ❼ else if (celsius > coldTemp && celsius < hot
    {
        digitalWrite(NORMAL, HIGH);
        delay(250);
        digitalWrite(NORMAL, LOW);
    }
    else
    {
     ❽ // Celsius is > hotTemp
        digitalWrite(HOT, HIGH);
        delay(250);
        digitalWrite(HOT, LOW);
```

```
    }
}
```

The sketch defines the values for the readout LED pins and sets them to LOW ❶. You can set your own values for hot and cold ❷; a “normal” temperature will be above the `coldTemp` value and below or equal to the `hotTemp` value.

The sketch sets the digital pins to outputs ❸ and then loops, taking the value from the temperature sensor ❹ and converting it to degrees Celsius ❺. Finally, it determines the temperature to be cold ❻, normal ❼, or hot ❽ and turns on the appropriate LED.

Increasing the ATtiny85’s Speed

The ATtiny85 can operate at three different speeds (in the Arduino environment and otherwise): 1 MHz (the default speed), 8 MHz, or 16 MHz. So far, your projects have used the default speed, which requires the least power and also works well for battery-powered projects. For projects involving more computation, you’ll need to change the speed to 8 MHz or greater, which will use more power.

In preparation for the next project, you'll change the operating speed by using the IDE to burn a new *bootloader*—the software loaded in the microcontroller that enables it to receive code via the SPI or USB connection—to the ATtiny85. This erases the sketch last uploaded and sets the speed of the oscillator inside the microcontroller, which determines the operation speed.

Oscillator Speed

Set up your circuit as you would to upload a sketch to your microcontroller. Next, open the IDE and select **Tools** ▸ **Clock** ▸ **Internal 16 MHz**, as shown in Figure 3-9.

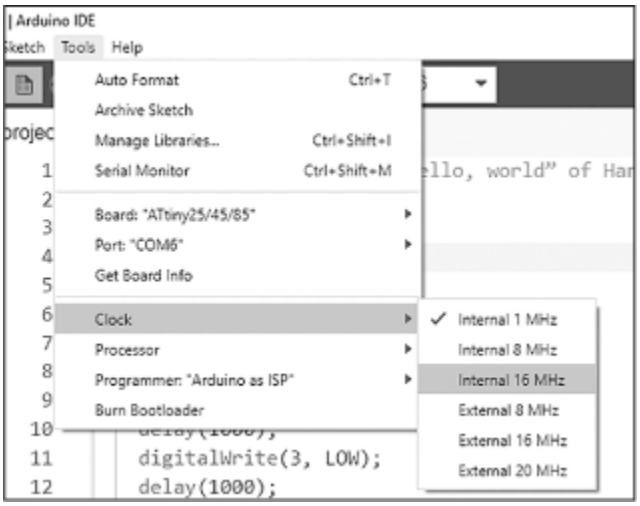


Figure 3-9: Preparing to change the clock speed

Next, select **Tools** ▸ **Burn Bootloader**. The operation should complete after a few moments, as shown in Figure 3-10.

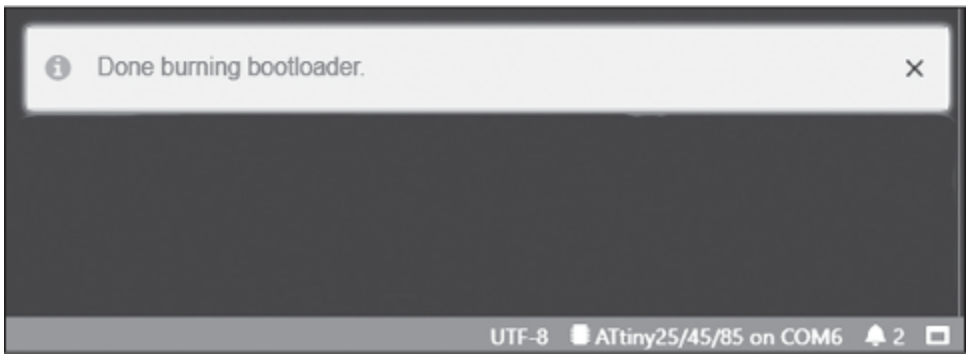


Figure 3-10: The Arduino IDE after completing the bootloader burn

Your sketches should now operate at 16 MHz when uploaded.

Oscillator Accuracy

The surrounding temperature can alter the accuracy of your microcontroller's speed. This is generally not a problem when you're simply using delays to blink LEDs or other simple tasks, but it can become an issue if you're using the `millis()` and `micros()` functions for more exact timing. For example, the speed can vary +/—10 percent with an external temperature of around 25°C.

The solution is to use an external crystal circuit, like the Arduino Uno and other boards do. You'll need two 22 pF ceramic capacitors and an 8, 16, or 20 MHz through-hole HC-49-type crystal like the one in Figure 3-11.



Figure 3-11: A through-hole HC-49-type crystal

NOTE

If you made the "breadboard Arduino" described in Arduino Workshop*, 2nd edition, you're familiar with this type of circuit.*

Figure 3-12 shows the crystal oscillator circuit addition to your ATtiny85.

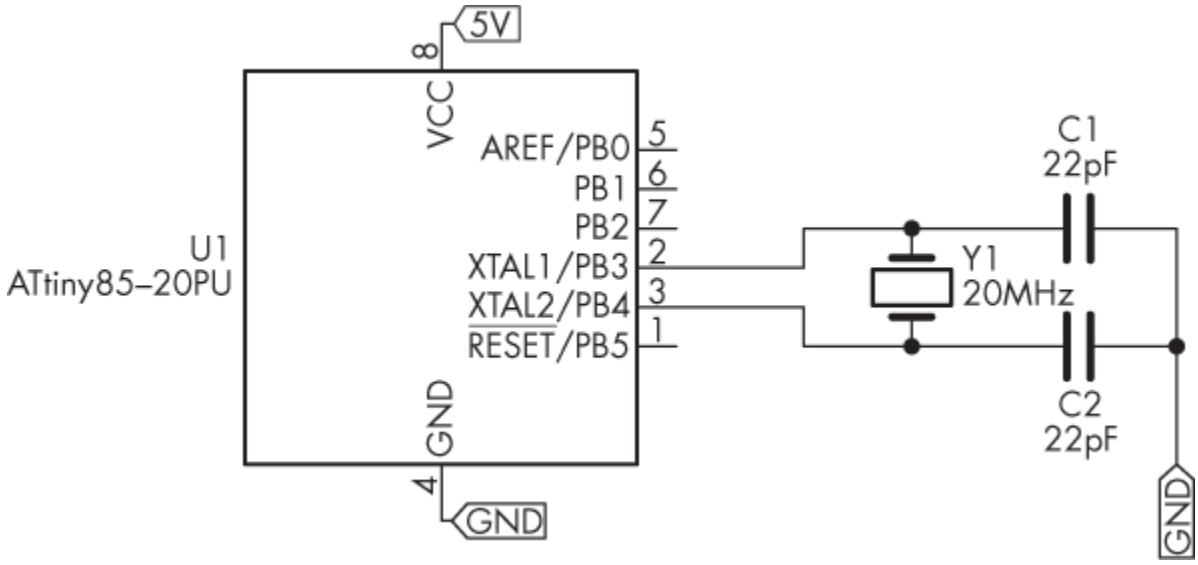


Figure 3-12: An ATtiny85 circuit with an external crystal subcircuit

When using an external crystal, don't forget to set the oscillator speed to external and match the crystal frequency. The one drawback to implementing an external crystal is that it uses digital pins 2 and 3, meaning you can no longer use them for outputs or inputs.

The next project prepares you to use the ATtiny85 in compact Arduino-compatible projects on a regular basis in your own future work.

Project #13: Creating an ATtiny85 Programming Shield

This project creates a small shield you can use to upload sketches to your ATtiny microcontrollers using the Arduino

Uno. The shield contains all the required circuitry, as well as two LEDs for quick prototyping or experimentation. You can then remove the microcontroller from the shield and use it for your own separate projects.

You'll need the following parts:

- The Project #13 PCB
- Two 1 kΩ, 0.25 W, 1 percent resistors
- A 10 µF, 16 V electrolytic capacitor
- Two 5 mm LEDs
- 1 × 40 inline 2.54 mm header pins
- An eight-pin IC socket

Figure 3-13 shows the schematic for this project.

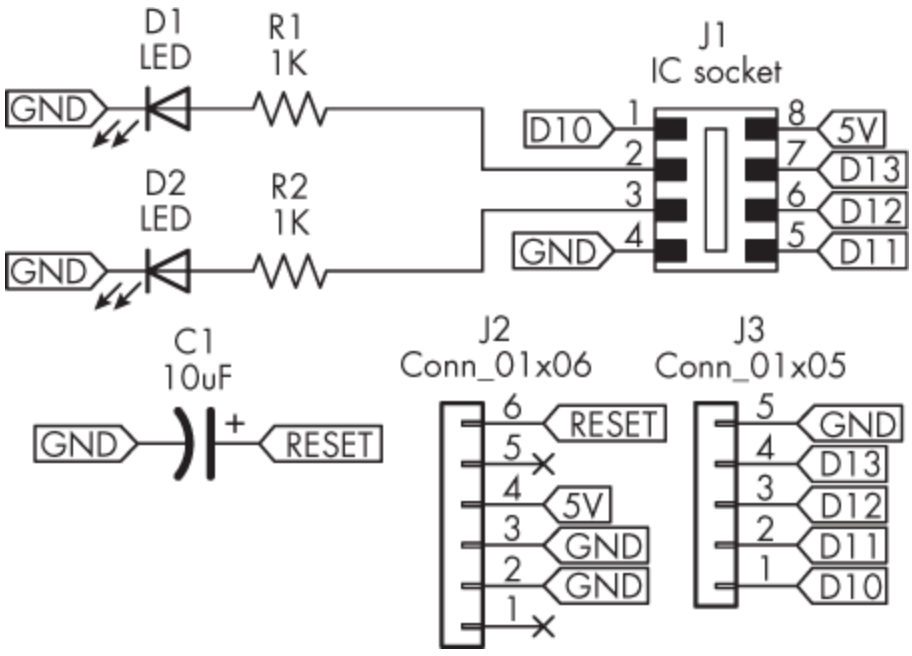


Figure 3-13: The schematic for Project #13

To assemble the circuit, connect the parts as labeled on the PCB in Figure 3-14. Start with the resistors and then the IC socket. Next, fit the capacitor, taking note of the polarity as shown on the PCB. Then fit the LEDs, with their short legs matching the PCB's square holes.

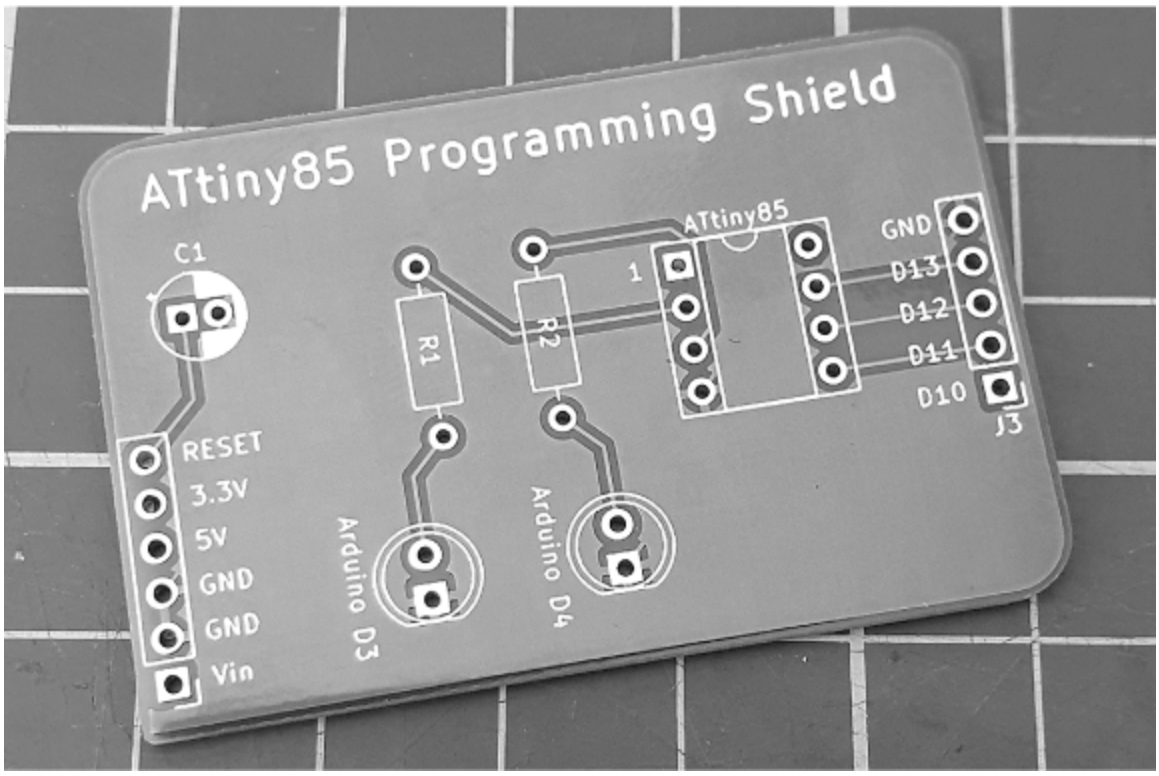


Figure 3-14: The PCB for Project #13

Finally, trim the inline header pins so you have one six-pin and one five-pin length, and then drop them into an Arduino Uno, as shown in Figure 3-15.

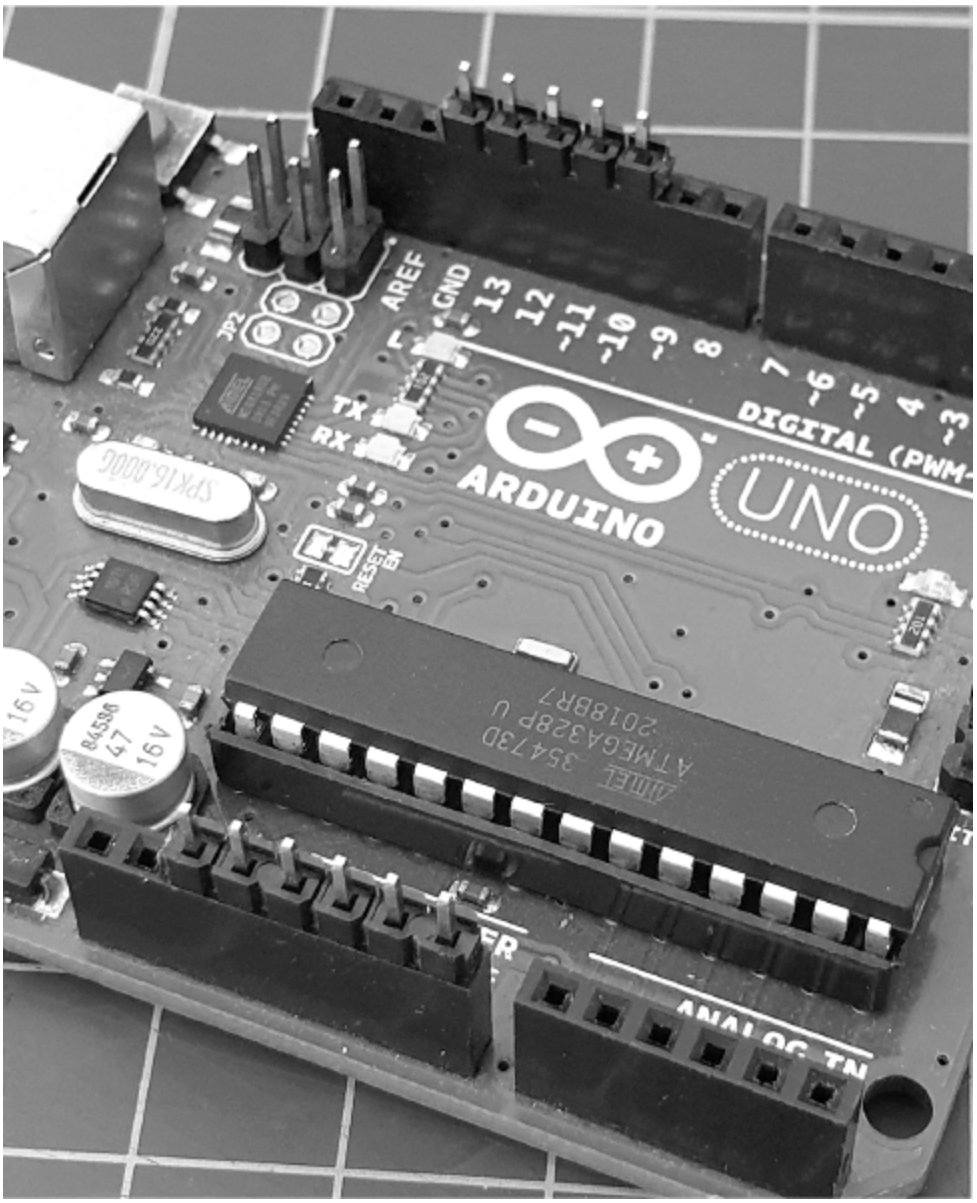


Figure 3-15: Preparing the inline header pins

Place the PCB on top of the headers and solder the pins as shown in Figure 3-16.

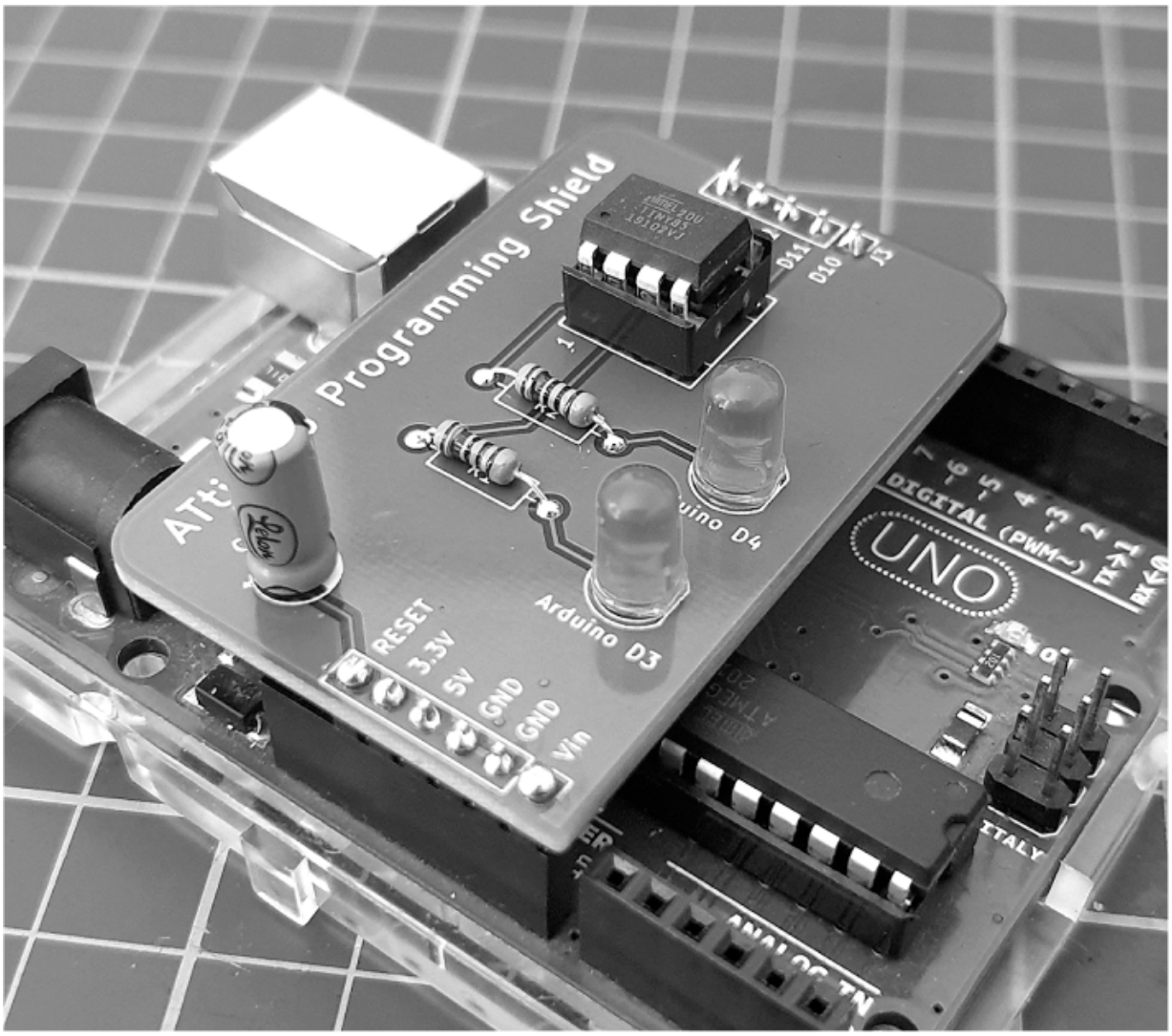


Figure 3-16: The assembled shield

You can now use this shield to easily upload sketches by removing the shield when uploading the Arduino as an ISP sketch and then inserting the shield when uploading the sketch for the ATtiny. This makes it faster and easier to set up ATtiny85-based projects in the future, allowing you to avoid manually wiring from the Uno to a solderless breadboard.

Moving On

This chapter showed you how to use ATtiny microcontrollers for smaller, simpler, and less expensive Arduino-compatible projects. The Arduino programming shield also gave you a faster way to upload code to the ATtiny.

In the next chapter, you'll learn how to keep your Arduino running with a watchdog timer.

4

BUILDING WATCHDOG TIMERS



Projects that sit unattended for some time may lock up or stop working because of accidents, suboptimal design, or power issues. To solve these problems, you can use a *watchdog timer*, a small circuit that sits dormant while the project is operating normally but hard resets the microcontroller to a restart operation if it doesn't receive a normal signal. These timers can help you build more advanced, professional, and reliable projects.

In this chapter, you'll learn to:

- Configure the 555 timer IC as an astable timer

- Use the formulas to calculate 555 timer IC delay periods
- Build your own watchdog timer using a PCB

Ensuring Constant and Reliable Operation

As you gain microcontroller expertise, you may find yourself building projects or products operating without your constant attention over long periods of time. This could be a weather-monitoring system, a data-logging device, a hardware web host that allows remote reviews of current data, or some other device whose physical location is hard to access (buried in a basement or an attic).

When the final project is out of your hands, how can you ensure constant and reliable operation? Even when your hardware and sketch or code seems foolproof, unexpected situations can cause your Arduino to stop operating. A temporary power brownout could freeze the microcontroller (that is, halt the sketch). An unexpected temperature extreme could also cause issues. Or your sketch may have unanticipated bugs. For these reasons and more, you'll need a watchdog.

In this section, I'll explain the theory behind watchdog timers and then show you how to build your own watchdog timer

circuit.

Watchdog Timer Theory

A watchdog timer circuit is an external timer connected to the RESET pin on the Arduino board. The watchdog circuit begins with a HIGH signal set to change to LOW after a predetermined length of time, which will cause the Arduino to reset and start again. However, the Arduino can prevent this reset by sending a regular signal, or *heartbeat*, via a digital output pin connected to the watchdog circuit. The heartbeat resets the timer circuit before the watchdog timer can reset the Arduino.

Consider the timing diagram in Figure 4-1.

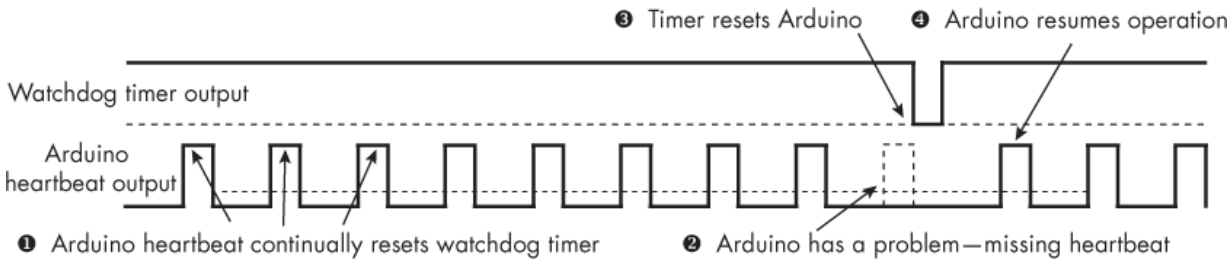


Figure 4-1: A timing diagram of a watchdog timer output and Arduino heartbeat output

The top signal is the output from a watchdog timer, continually oscillating between HIGH and LOW. The period for the HIGH signal is much longer than that for LOW. The watchdog output is connected to the Arduino's RESET pin. When the watchdog

timer resets, the output starts at HIGH, meaning the Arduino must keep resetting the watchdog timer to stop the watchdog timer resetting the Arduino.

To enable this, an Arduino digital output pin connected to the watchdog timer sends a quick pulse (HIGH and then LOW) to the timer ❶, resetting the timer and stopping the timer output from going LOW. If the Arduino has a problem and stops pulsing the heartbeat signal to the watchdog timer ❷, the watchdog timer oscillations continue and its signal will go LOW ❸, resetting the Arduino. After this reset, the Arduino should recover and begin sending out the heartbeat signal again ❹.

The watchdog timer is a hardware-based solution, making it less error-prone than a software-based function. If the reset doesn't get the Arduino going again, there's a major fault in the project that will require you to physically examine the hardware or code.

Watchdog Timer Circuit Configuration

The heart of the watchdog timer circuit is the 555 timer IC shown in Figure 4-2. This compact part contains a customizable timing circuit you can use in several ways.



Figure 4-2: A 555 timer IC

To use the 555 as a watchdog timer, you'll configure it as an *astable timer*, one that generates a continuous signal output with a longer HIGH than LOW, as shown in Figure 4-3.

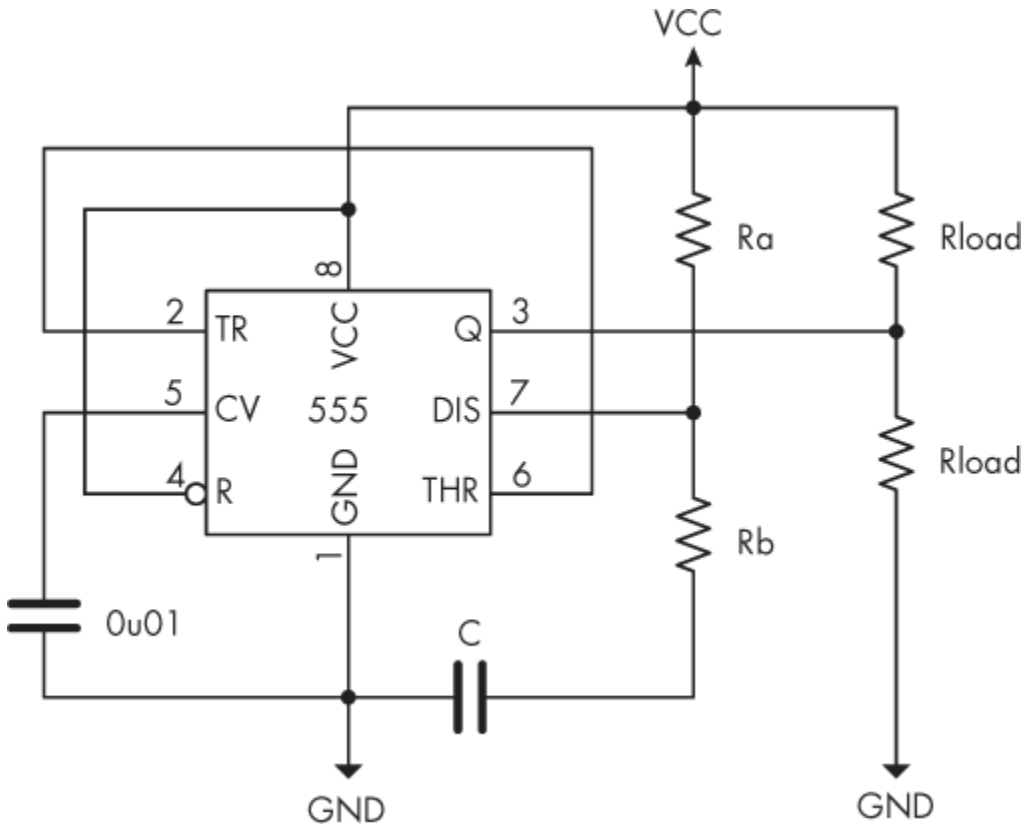


Figure 4-3: A basic astable timer circuit

The astable timer circuit is fairly simple. Inside the 555 are a voltage comparator circuit and a flip-flop, which changes state between LOW and HIGH to drive the output pin. When power is applied, the voltage at pin 2 (the trigger pin) is less than one-third V_{CC}, meaning the flip-flop switches to set the 555 output to HIGH. C then charges via R_A and R_B.

After a period with output at HIGH, the voltage at pin 6 (the threshold pin) goes above two-thirds V_{CC}. The flip-flop switches to set the 555 output to LOW. This also enables the discharge function so that C discharges via R_B.

After a period with the output at LOW, the voltage at pin 2 is less than one-third V_{CC}, so the flip-flop switches to set the 555 output to HIGH … and the cycle repeats. The output is sourced from pin 3, which operates as an open collector—that is, when LOW, it can sink current to GND.

The duty cycle of the output signal and the HIGH and LOW periods are determined by the values of resistors R_1 and R_2 and the capacitor C. The R_L parts are the loads, or the circuitry controlled by the output. Current can flow from V_{CC} through the higher R_L when the output is off, and from the output through the lower R_L to GND when the output is on. Up to 200 mA of current can flow in or out of the 555's output pin.

To determine the HIGH and LOW period times for the output—and if you are curious, the frequency of the output signal (which we don't use but might as well set!)—we use the following three formulas:

$$T_{low} = 0.693\ (R_1 + R_2)\ C$$

$$T_{high} = 0.693\ R_2C$$

$$\text{frequency} = \frac{1.44}{(R_1 + 2R_2)C}$$

T_{low} is the length of time the astable output from the 555 timer is LOW, and T_{high} is the length of time the output is HIGH. For example, with a value of 100 kΩ for R_1, a value of 4.7 kΩ for R_2, and a capacitor C of 100 uF, the calculations are as follows:

T_{high} = 0.693 (4,700) 0.0001 = 7.255 s

T_{low} = 0.693 (100,000 + 4,700) 0.0001 = 0.3257 s

A spreadsheet with these formulas is available in the sketch download from *https://nostarch.com/arduino-arduinians*. Use these formulas to easily calculate the values of resistors and the capacitor needed for your required HIGH and LOW times. The value of R_2 generally remains constant, as R_1 has more effect on the final values.

When building a 555 timer circuit, use 1 percent tolerance resistors. Larger capacitors can have a very large tolerance, especially during temperature fluctuations—sometimes up to +/−20 percent—so try to keep the capacitor value as small as possible.

Now that you know how to operate a 555 timer, let's build a watchdog timer circuit for our Arduino.

Project #14: Building a Watchdog Timer

In this project, you'll build a convenient and adjustable watchdog timer for your Arduino Uno or other microcontroller that utilizes the type of reset circuit discussed. You can build the circuit on a solderless breadboard for a more temporary setup or download the PCB files for this project and have your own PCB manufactured to create a permanent watchdog timer.

You'll need the following parts:

- The Project #14 PCB or solderless breadboard
- An NE555 timer IC (not the CMOS 7555 version!)
- An eight-pin IC socket
- A 1N4001 diode
- A 5 mm LED
- 1 × 40 2.54 mm inline header pins (if using PCB)
- Various jumper wires
- A 2N7000 N-MOSFET
- Two 1 kΩ, 0.25 W, 1 percent resistors
- A 10 kΩ, 0.25 W, 1 percent resistor
- A 100 Ω, 0.25 W, 1 percent resistor
- Other resistors (see the following section)
- A 0.01 µF ceramic capacitor
- A 0.1 µF ceramic capacitor
- A 100 µF, 16 V electrolytic capacitor

Figure 4-4 shows the schematic for this project.

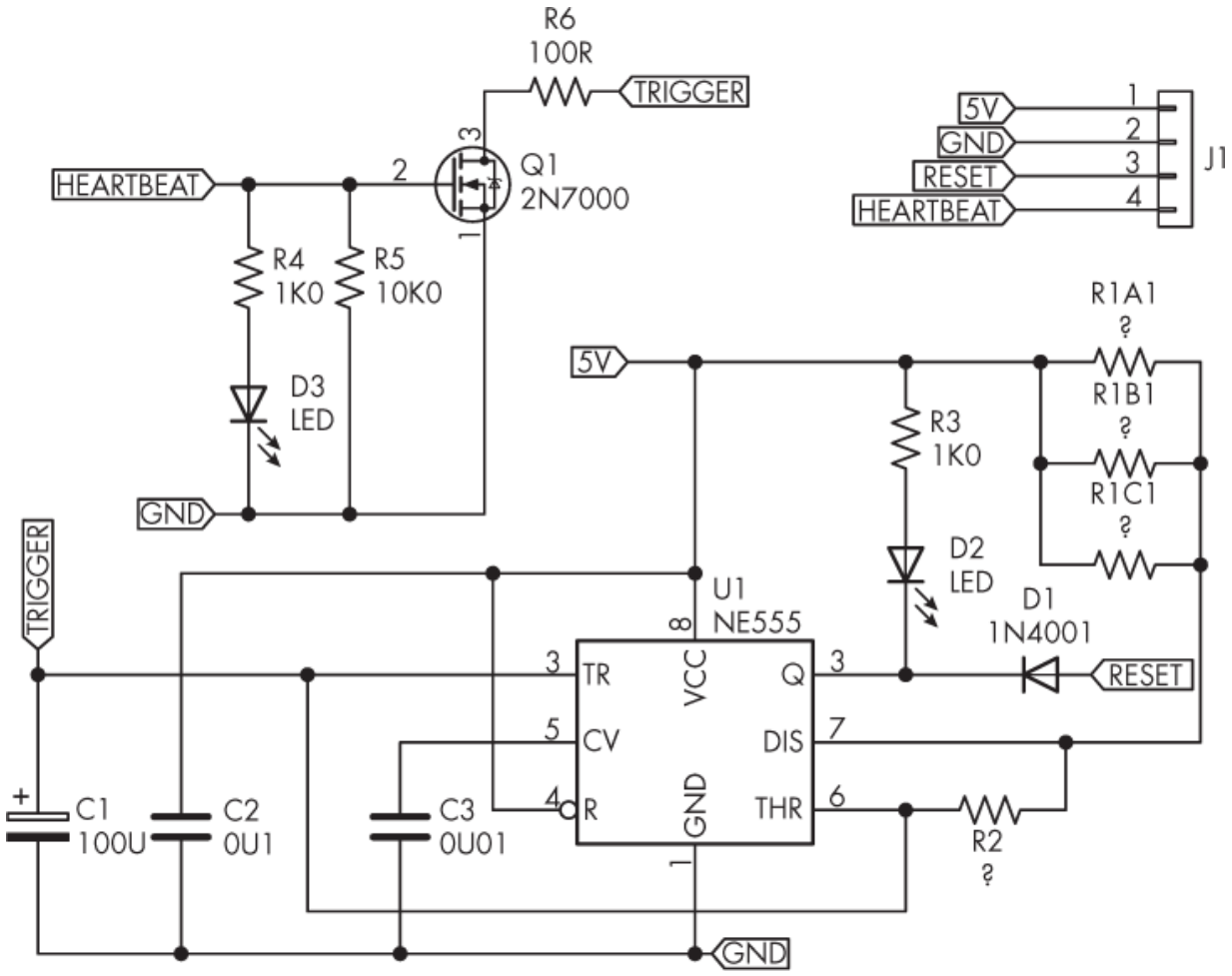


Figure 4-4: The schematic for Project #14

Before assembling the circuit, determine which values to use for R_1 and R_2 to control the length of the HIGH and LOW periods of the output, using the formulas provided in the previous section. The results will differ depending on whether you're using a PCB or solderless breadboard, as the contacts inside a breadboard also have their own resistance value.

For those of you building with a PCB, the schematic shows three resistors for R_e: R_{1A}, R_{1B}, and R_{1C}. I have left space for up to three resistors to be used as a combined value for R_1 so that you can change resistor values if required by adding new resistors in the R_{1B} or R_{1C} spaces and cutting out the unwanted values. Alternately, you may need to use two or three resistors in parallel to get your exact required value.

To calculate the value of resistors in parallel, use the following:

$$R_T = 1 / (1 / R_{1A} + 1 / R_{1B} + 1 / R_{1C} + \ldots)$$

If you just need to use one resistor for R_1, insert it into the R_{1A} space on the PCB.

If you're not sure what values to use for this project, a good default option is to use around 100 kΩ for R_1 and 4.7 kΩ for R_2. This will create a HIGH delay of just over 57 seconds, close to a minute. With this delay, your Arduino must reset the timer more often than once every 7 seconds to avoid being reset by the timer itself.

The 555 Circuit's Inner Workings

Let's take a closer look at the 555 circuit's inner workings. When you apply power to the circuit, the 555 should start operating as

described earlier, with the output signal connected to the Arduino's RESET pin and starting as HIGH. The Arduino should operate as normal. Diode D_1, connected between the 555's output pin and the Arduino RESET pin, ensures no stray signals can reset the Arduino.

Over time, as determined by your R_1 values, capacitor C_1 should start charging via the 555's trigger pin. When the Arduino sends a HIGH signal via the heartbeat pin, it should turn on the N-MOSFET Q_1. Doing so not only blinks LED D_3 (a visual indication of the Arduino heartbeat signal) but also shorts the capacitor C_1 to GND via R_6, emptying the capacitor. The capacitor C_1 then starts charging again, resetting the amount of time before the 555 can reset the Arduino.

If the Arduino keeps sending heartbeat pulses, the 555 timer never has a chance to change the output state to LOW and reset the Arduino, as discussed. However, if the Arduino stops sending heartbeat signals, capacitor C_1 will continue to charge until the 555's threshold pin voltage reaches two-thirds of 5 V. At that point, the 555's output changes state to LOW for a short period (determined by R_2). This resets the Arduino and blinks LED D_2 (a visual indication that the reset circuit has been activated). When the 555's output pin is LOW, current can flow from 5V, through R_3 and the LED, and into the 555 output pin.

When the output pin is LOW, current can flow into the pin, as opposed to current flowing out from the output pin when it is HIGH. The output then changes back to HIGH, and the process starts again.

Resistors R_3 and R_4 limit current to the LEDs, while resistor R_5 pulls down the N-MOSFET until activated, avoiding unplanned activation. Resistor R_6 protects C_1 while the capacitor is discharging, avoiding a dead short when a heartbeat signal is received. C_2 smooths the power supply over the circuit. Finally, the 555 requires C_3 for normal operation.

Circuit Assembly

Assemble the project using the PCB shown in Figure 4-5 as usual: start with the lowest-profile parts, such as the resistors, and then work your way to the tallest parts (the capacitors and inline header pins).

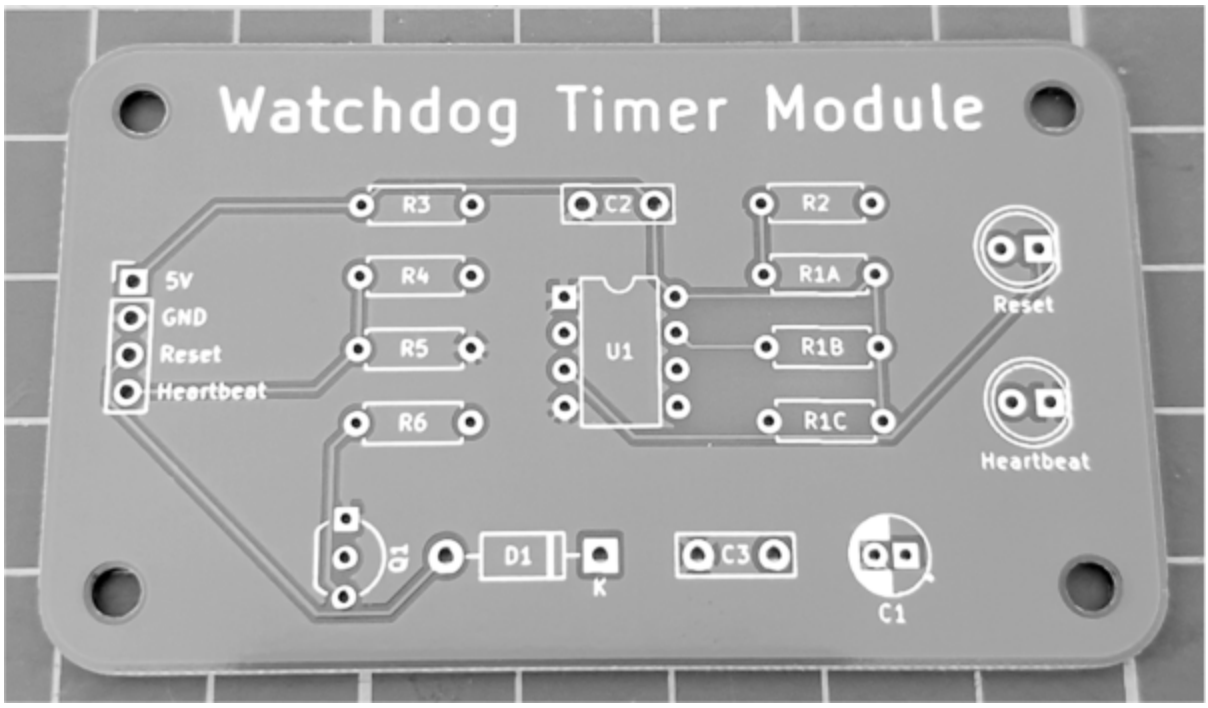


Figure 4-5: The PCB for Project #14

Don’t forget to use an IC socket for the 555 and match the notch at the end of the IC with the notch on the PCB. You may need to trim your inline header pins to get one strip of four pins.

Once assembled, your watchdog timer should resemble the one shown in Figure 4-6.

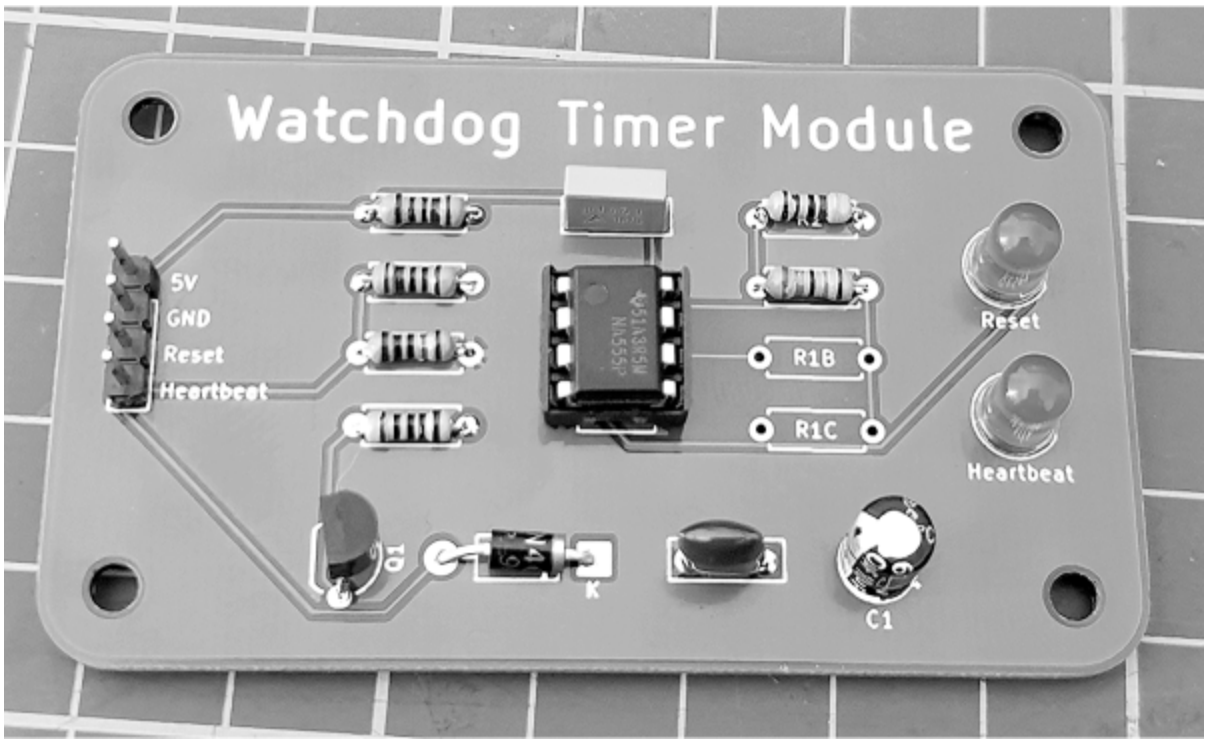


Figure 4-6: The completed watchdog timer board

Connect the 5V, RESET, and GND pins to the matching Arduino pins and connect the heartbeat pin to an unused digital pin on your Arduino. To make your sketch send a pulse at a frequency faster than the operation of the watchdog timer, consider the following example of using the watchdog timer. Upload the sketch after connecting the watchdog timer to your Arduino.

```
  // Project #14 - Watchdog timer

❶ #define heartBeatPin 5

❷ void heartBeat()
```

```
  {
      digitalWrite(heartBeatPin, HIGH);
      delay(100);
      digitalWrite(heartBeatPin, LOW);
  }

❸ void setup()
  {
      pinMode(heartBeatPin, OUTPUT);
      heartBeat(); // Send heartbeat signal to wat
  }

  void loop()
  {
      // Do things
    ❹ heartBeat(); // Send heartbeat signal to wat
  }
```

This sketch defines digital pin 5 as the heartbeat connection to the watchdog timer circuit ❶. The `heartBeat()` function generates the heartbeat pulse ❷, which simply flips the pin on and off for a moment—long enough to reset the watchdog timer circuit. The `void setup()` function ❸ sends the heartbeat to reset the watchdog as soon as possible after the Arduino is reset or powered up. Finally, the main loop ❹ sends the heartbeat regularly.

Placing the `heartbeat()` function in `void loop()` is an ideal spot, assuming the time taken by a single run of the code in the loop is less than the timeout period of the watchdog. You can always insert more calls to `heartBeat()` in your other functions as well if you want to increase the number of watchdog-initiated resets of your project.

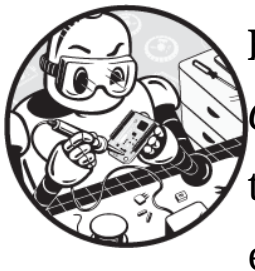
Moving On

This chapter covered why you'd want to use a watchdog timer, along with the theory of operating and building your own circuit. You learned how to use the watchdog timer method in order to maintain the reliability of future Arduino-based projects, as well as how to use other microcontrollers such as the ATtiny or with other electronic devices that have an output and reset on LOW input.

In the next chapter, you'll learn to control multiple LEDs with a minimal number of digital output pins.

5

CONTROLLING LEDS WITH CHARLIEPLEXING



It's easy to control many LEDs with one Arduino board. However, you can also use the *Charlieplexing* method in the Arduino environment to control multiple LEDs one at a time without external display driver ICs. This is a great way to reduce part count (and expenses) in a project involving numeric or character displays.

In this chapter, you'll learn to:

- Use Charlieplexing with an Arduino Uno or compatible board or an ATtiny85 microcontroller
- Use Charlieplexing with 6 and 12 LEDs

- Simplify planning Charlieplexed circuits with the help of logic tables

You'll also build a 30-LED matrix display for alphanumeric data that you can use for your own projects

Introduction to Charlieplexing

The term *Charlieplexing* is named after Charlie Allen, who originally proposed the idea during his time at Maxim Integrated (now part of Analog Devices), which produced integrated circuits such as the popular MAX7219 LED display driver. The word is a portmanteau composed of "Charlie" and "multiplexing," the latter being a method of controlling many LEDs with few I/O pins on a microcontroller.

Charlieplexing is a type of multiplexing that relies on the *tri-state* capability of microcontroller I/O pins to control the required current to activate multiple LEDs in circuits. You can set each pin to be in one of three states:

High Current flows from an I/O pin, such as when using a `digitalWrite()` function.

Low Current does not flow from an I/O pin, when the pin can also sink (receive current).

Input state (or high-Z) The pin is set to a digital input. In this state, little to no current will flow in either direction. This is also known as the *high impedance* or *high-Z* state.

You can set these I/O pins to different states to direct current through a circuit and create a controllable matrix of LEDs. Only one LED may be turned on at any given time, and all the LEDs used in one circuit must be the same specification—that is, they must all have the same forward voltage and operating current.

To help understand Charlieplexing, let's work through some small examples. First, consider the schematic in Figure 5-1.

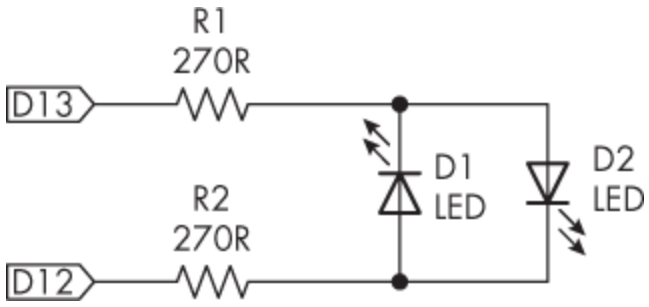


Figure 5-1: The schematic for two LEDs connected to the Arduino outputs

If pin D13 is HIGH and pin D12 is LOW, current flows from pin D13 through R_1, through D_2, through R_2, and into D12, meaning LED D_2 will turn on. If you reverse the outputs so pin D12 is HIGH and pin D13 is LOW, current flows from pin D12 through D_1, through R_1, and into D13, meaning LED D_1 will turn on. Current will always flow through both resistors, so the sum of

their values should be suitable for running an LED from a digital output. With a 5 V output, values of 270 Ω through around 510 Ω would enable sufficient brightness.

To test Charlieplexing, build the circuit in Figure 5-1; then run the sketch in Listing 5-1.

```
void setup()
{
    pinMode(13, OUTPUT);
    pinMode(12, OUTPUT);
}

void loop()
{
    digitalWrite(13, HIGH);
    digitalWrite(12, LOW);
    delay(500);
    digitalWrite(13, LOW);
    digitalWrite(12, HIGH);
    delay(500);
}
```

Listing 5-1: Two-LED demonstration

This sketch shows how to direct current to control one of two LEDs without a GND pin. It's intended only as a basic

introduction to the concept of Charlieplexing; there's no real benefit to Charlieplexing two LEDs, as it doesn't reduce the number of I/O pins used. However, the possibilities of Charlieplexing become obvious once you add another I/O pin to the circuit, as shown in the next project.

Project #15: Charlieplexing with Six LEDs

In this project, you'll control six LEDs with only three digital output pins, demonstrating the benefits of Charlieplexing.

You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- Three 270 Ω, 0.25 W, 1 percent resistors
- Six LEDs
- Two solderless breadboards
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 5-2.

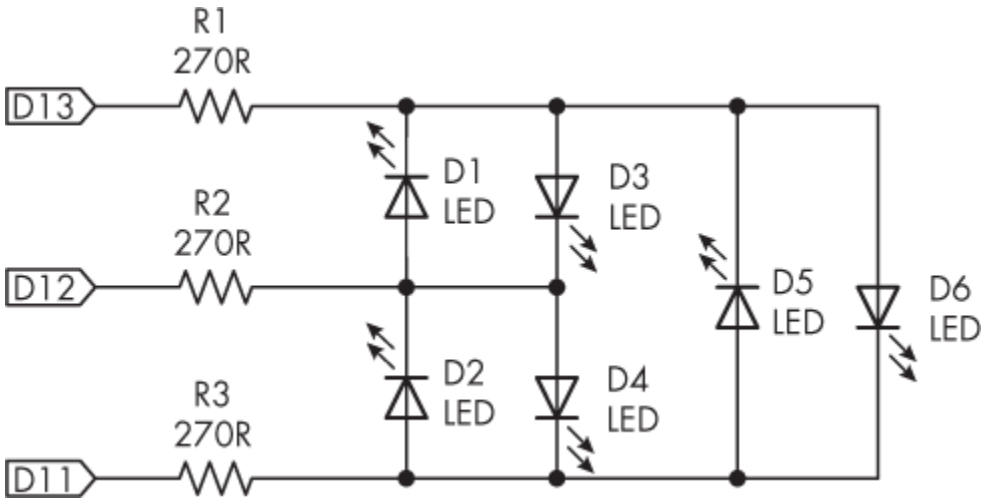


Figure 5-2: The schematic for Project #15

For easy assembly, use two solderless breadboards side by side, as shown in Figure 5-3.

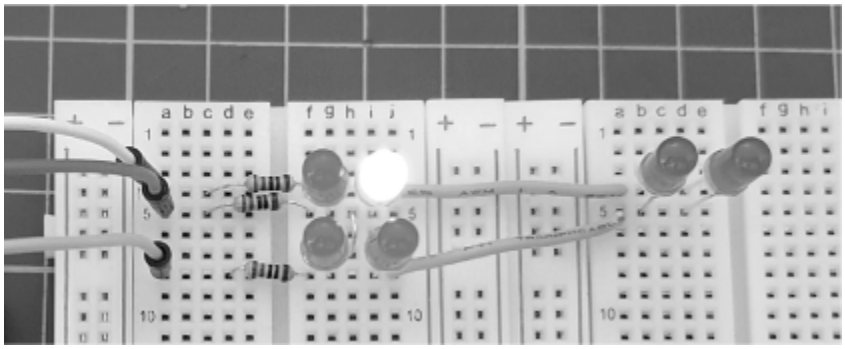


Figure 5-3: The assembled Project #15

Now enter and upload the following sketch to your Arduino. A moment after uploading the sketch, each LED from 1 through 6 should light in turn.

```
  // Project #15 - Charlieplexing six LEDs

❶ #define d 1000

❷ void LED1()
  {
    ❸ DDRB = B00110000;
    ❹ PORTB = B00010000;
  }

  void LED2()
  {
      DDRB = B00110000;
      PORTB = B00100000;
  }

  void LED3()
  {
      DDRB = B00011000;
      PORTB = B00001000;
  }

  void LED4()
  {
      DDRB = B00011000;
      PORTB = B00010000;
  }
```

```
  void LED5()
  {
    ❺ DDRB = B00101000;
       PORTB = B00001000;
  }

  void LED6()
  {
      DDRB = B00101000;
      PORTB = B00100000;
  }

  void setup() {}

  void loop()
❻ {
      LED1(); delay(d); LED2(); delay(d); LED3(); d
      LED4(); delay(d); LED5(); delay(d); LED6(); d
  }
```

The first few moments of operation may seem like magic if you're accustomed to using one LED per digital output pin, but you can achieve these effects simply by carefully directing current through the appropriate pins and resistors.

Consider the first LED in the schematic, D_1, controlled by the function `LED1()` ❷. Trace the path through the schematic as follows: current needs to flow from pin D12 through R_2, through the LED itself, then through R_1, and into pin D13. You don't want current to flow through D_4, so you set pin D11 to an input, blocking current flow. Although D_4 and D_5 are in the current's path, they will not activate, as the resistor and their forward operating voltage reduce the current's voltage so greatly.

Instead of using lots of `pinMode()` and `digitalWrite()` functions, this sketch uses port manipulation (covered in Chapter 2) to control the digital pins. This greatly reduces the amount of required code, which also frees up more memory for the rest of your project.

To turn on D_1 in the code, the sketch turns on LED D_1 and sets pin D13 to HIGH and D12 to LOW using port manipulation. The sketch first activates PORTB pin types D13 to D8 ❸, setting D13 to output, setting D12 to output, setting D11 to input, and leaving the unused pins at 0. It then activates the output pins ❹, setting D13 to LOW, setting D12 to HIGH, and leaving input as 0. This turns on LED D_1.

For another example in the project, consider LED D_5. To turn it on, current must flow from pin D11 through R_3, D_4, and R_1, and then into D13. The sketch sets pin D11 to HIGH, pin D13 to LOW, and pin D12 as an input ❺ to stop current flowing in that direction.

The sketch sets the flow of current in this way for each LED in turn, within a function for each LED from ❶ onward. Unlike using conventional `digitalWrite()` functions where you need to first use `pinMode()` functions to configure a pin as an output or input, nothing is required in `void setup()`, so the code then lights each LED in turn from ❻.

You can adjust the amount of time between each LED being switched on by changing the defined value of `d` ❶. Although the sketch doesn't use this option, you can also turn off all the LEDs at once at any point in the code with the following line of code:

```
DDRB = B00000000; // Set all pins to inputs (hig
```

Keep this project's hardware together to use in the following project. In the next section, I'll show you a more systematic way of determining which pins to set as HIGH or LOW.

Larger Charlieplexing Displays

When building larger Charlieplexing circuits, creating a logic table with the LEDs and their required pin types and outputs can help you plan how to write your code. This makes it easier to create the required `DDRB` and `PORTB` functions to control the LEDs. To do this, trace the current path you require for each LED and then determine the pin status for each LED. For example, Table 5-1 is the logic table for Project #15, where L stands for LOW, H for HIGH, and Z for input.

Table 5-1: Logic Table for Project #15

LED#	Pin D13	Pin D12	Pin D11
1	L	H	Z
2	H	L	Z
3	Z	L	H
4	Z	H	L
5	L	Z	H

LED#	Pin D13	Pin D12	Pin D11
6	H	Z	L

Creating such a table simplifies the process of determining the port manipulation commands, as you can just match each letter with its respective ones, zeros, and inputs. To make your own table, trace the current flow in the schematic required for each LED and then note the status required for each I/O pin. For example, to activate LED 4 in Project #15, current must flow out from pin D12 (so it's HIGH) through R_2, through LED D_4, then through R_3, and back to D11 (so it's LOW). You don't want current to flow in or out of D13, so it will be Z. You can then use this HIGH, LOW, and Z information to create the required `DDR` and `PORTB` functions in your code.

Once you have more than two I/O pins in use, you can connect a pair of LEDs to each of the other I/O lines. To determine how many LEDs you can control with any number of available I/O pins on your Arduino, use the formula $L = n^2 - n$, where L is the number of LEDs and n is the number of I/O pins. For example, if you have 4 pins, you can use 12 LEDs ($4^2 - 4 = 12$).

Let's try this in the next project.

Project #16: Charlieplexing with 12 LEDs

In this project, you'll practice using Charlieplexing to control an even larger LED display with only four digital output pins. You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- Four 270 Ω, 0.25 W, 1 percent resistors
- 12 LEDs
- Two solderless breadboards
- Male-to-male jumper wires

Assemble the circuit as shown in Figure 5-4.

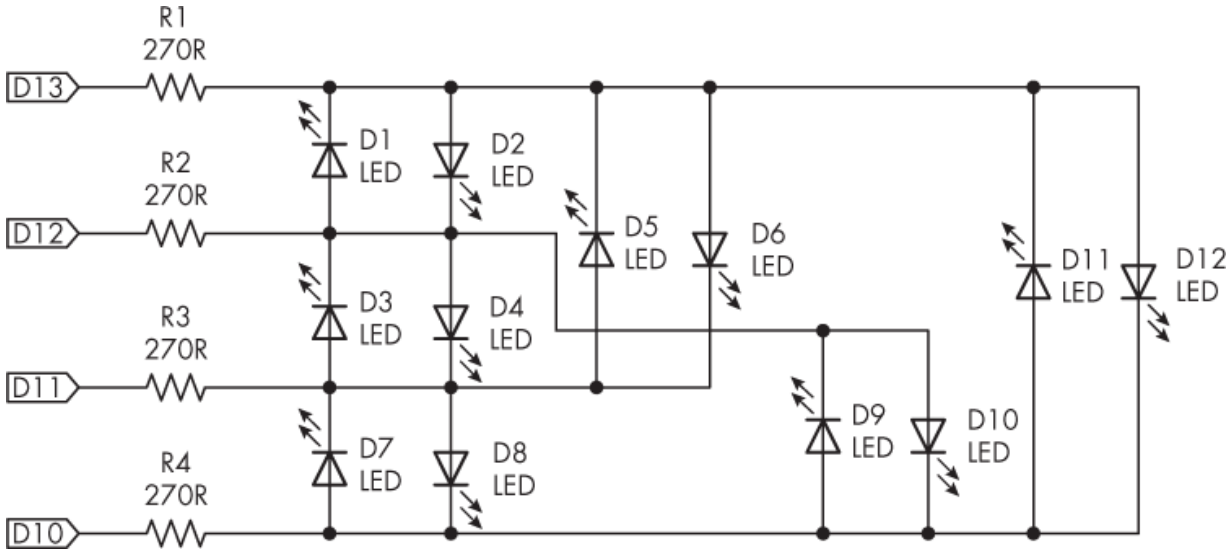


Figure 5-4: The schematic for Project #16

You can build this project by expanding on the hardware setup from Project #15, using two solderless breadboards side by side, as shown in Figure 5-5.

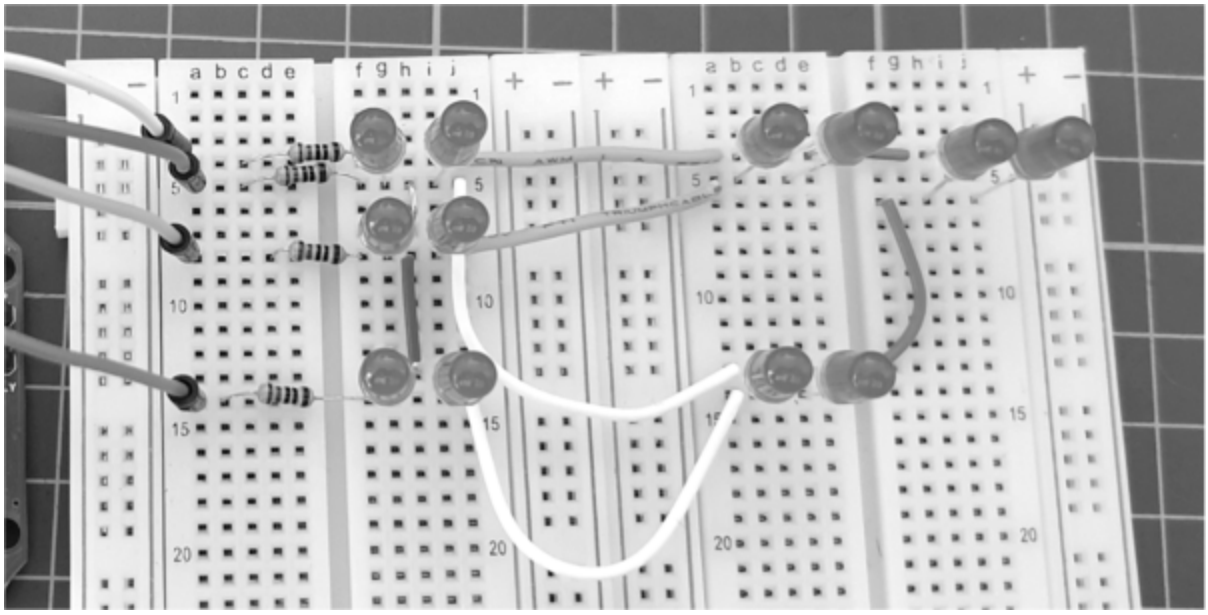


Figure 5-5: The assembled Project #16

Table 5-2 is Project #16's logic table. You can compare this with the port manipulations in the subsequent sketch to increase your familiarity with controlling the LEDs. For practice, trace the current for various LEDs through the schematic and compare with the table.

Table 5-2: Logic Table for Project #16

LED#	Pin D13	Pin D12	Pin D11	Pin D10

LED#	Pin D13	Pin D12	Pin D11	Pin D10
1	L	H	Z	Z
2	H	L	Z	Z
3	Z	L	H	Z
4	Z	H	L	Z
5	L	Z	H	Z
6	H	Z	L	Z
7	Z	Z	L	H
8	Z	Z	H	L
9	Z	L	Z	H
10	Z	H	Z	L
11	L	Z	Z	H

LED#	Pin D13	Pin D12	Pin D11	Pin D10
12	H	Z	Z	L

Enter and upload the Project #16 sketch to your Arduino. Once again, a moment or two after uploading the sketch, each LED should illuminate in turn.

Let's see how this works:

```
// Project #16 - Charlieplexing with 12 LEDs

#define d 250 ❶
void LED1()
{
    DDRB = B00110000; ❷
    PORTB = B00010000;
}

void LED2() {DDRB = B00110000; PORTB = B00100000
void LED3() {DDRB = B00011000; PORTB = B00001000
void LED4() {DDRB = B00011000; PORTB = B00010000
void LED5() {DDRB = B00101000; PORTB = B00001000
void LED6() {DDRB = B00101000; PORTB = B00100000
void LED7() {DDRB = B00001100; PORTB = B00000100
void LED8() {DDRB = B00001100; PORTB = B00001000
```

```
void LED9() {DDRB = B00010100; PORTB = B00000100
void LED10(){DDRB = B00010100; PORTB = B00010000
void LED11(){DDRB = B00100100; PORTB = B00000100
void LED12(){DDRB = B00100100; PORTB = B00100000

void noLED() ❸
{
    DDRB = B00000000; // All to inputs, high-imp
}

void setup() {}

void loop()
{
LED1(); delay(d); LED2(); delay(d); LED3(); dela
LED5(); delay(d); LED6(); delay(d); LED7(); dela
LED9(); delay(d); LED10(); delay(d); LED11(); de
}
```

As with Project #15, each LED is activated by its own function from ❷ onward. After `void LED1()`, I've compressed each function into one line of code to save space. Again, port manipulation greatly improves the sketch's efficiency.

You can change the delay between each LED's turning on and off ❶. The sketch includes a function to turn off all the LEDs if

required for other future projects ❸. Once again, to save space, I've placed multiple functions on one line ❹, where each LED is turned on after a preset delay.

The placement of the LEDs in this project may seem random, but you now have the skills to create more useful, complicated displays with Charlieplexing, which you'll do in Project #18. First, however, I'll show you how to use Charlieplexing with an ATtiny85.

Project #17: Charlieplexing with the ATtiny85

Using Charlieplexing with an ATtiny85 microcontroller gives you a cheaper and smaller-circuit solution for controlling lots of LEDs without external devices. This project demonstrates how to control 12 LEDs using the ATtiny85.

You'll need the assembled LED circuit from Project #16, as well as either the ATtiny85 programming shield from Project #13 in Chapter 3 or the following parts:

- An Arduino Uno or compatible board and USB cable
- An ATtiny85 microcontroller
- A 10 µF electrolytic capacitor
- Two solderless breadboards
- Male-to-male jumper wires

First, assemble your ATtiny85 programming setup using the instructions from Chapter 3 (using Project #13 if you built it), using the programming shield or the parts listed for this project, and then upload the code. Next, connect your ATtiny85 to the LED circuit, as shown in Figure 5-6. The LED circuit has different pin connections from those in Project #16 but is otherwise the same.

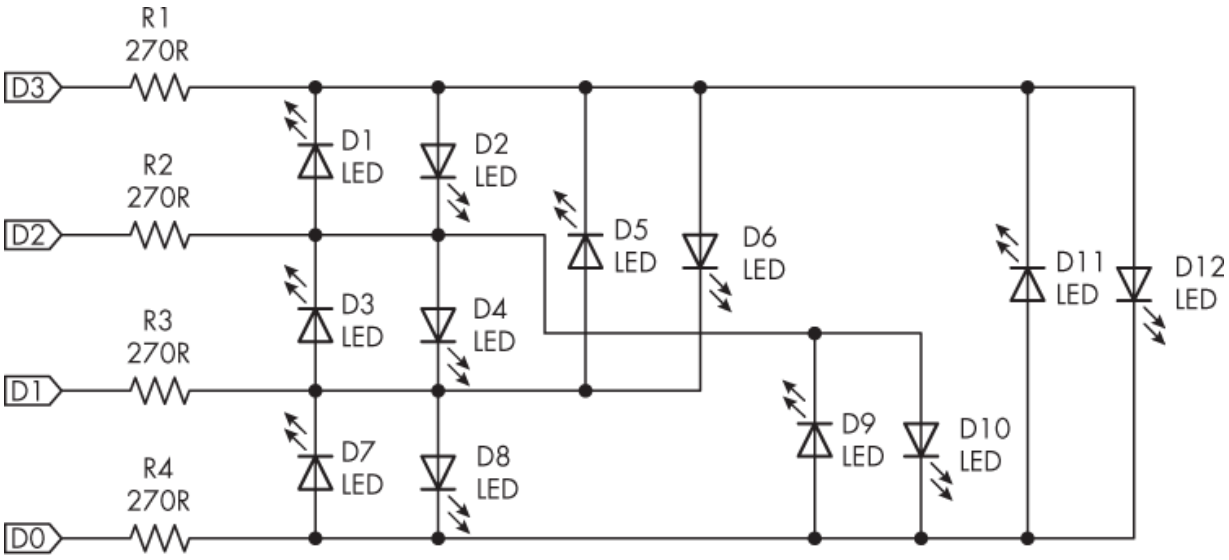


Figure 5-6: The schematic for Project #17

Next, upload the sketch:

```
// Project 17 - Charlieplexing with ATtiny85

#define d 250
void LED1() {DDRB = B00001100; PORTB = B00000100
void LED2() {DDRB = B00001100; PORTB = B00001000
```

```
void LED3() {DDRB = B00000110; PORTB = B00000010
void LED4() {DDRB = B00000110; PORTB = B00000100
void LED5() {DDRB = B00001010; PORTB = B00000010
void LED6() {DDRB = B00001010; PORTB = B00001000
void LED7() {DDRB = B00000011; PORTB = B00000001
void LED8() {DDRB = B00000011; PORTB = B00000010
void LED9() {DDRB = B00000101; PORTB = B00000001
void LED10(){DDRB = B00000101; PORTB = B00000100
void LED11(){DDRB = B00001001; PORTB = B00000001
void LED12(){DDRB = B00001001; PORTB = B00001000

void noLED(){DDRB = B00000000;} // All to inputs

void setup() {}

void loop()
{
LED1(); delay(d); LED2(); delay(d); LED3(); dela
LED5(); delay(d); LED6(); delay(d); LED7(); dela
LED9(); delay(d); LED10(); delay(d); LED11(); de
}
```

Once uploaded, this sketch should operate in the same manner as Project #16, with each of 12 LEDs lighting up in turn. However, the code sets up the port manipulation differently. Since you're using `PORTB` pins 3, 2, 1, and 0 (digital D3 to D0) on

the ATtiny85, your `DDRB` and `PORTB` functions need to address bits 0 to 3.

Project #18: Building a 30-LED Matrix Display

This project takes Charlieplexing to the next level with a larger, 30-LED circuit you can use to display digits or characters of your own design.

It's theoretically possible but extremely difficult to build the circuit for this project on a pair of solderless breadboards, so I highly recommend downloading the PCB files for this project and having your own PCB manufactured. You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- Six 270 Ω, 0.25 W, 1 percent resistors
- Thirty 5 mm LEDs
- Two solderless breadboards or Project #18 PCB
- 1 × 40 2.54 mm inline header pins (if using PCB)
- Various jumper wires

Assemble the circuit as shown in Figure 5-7.

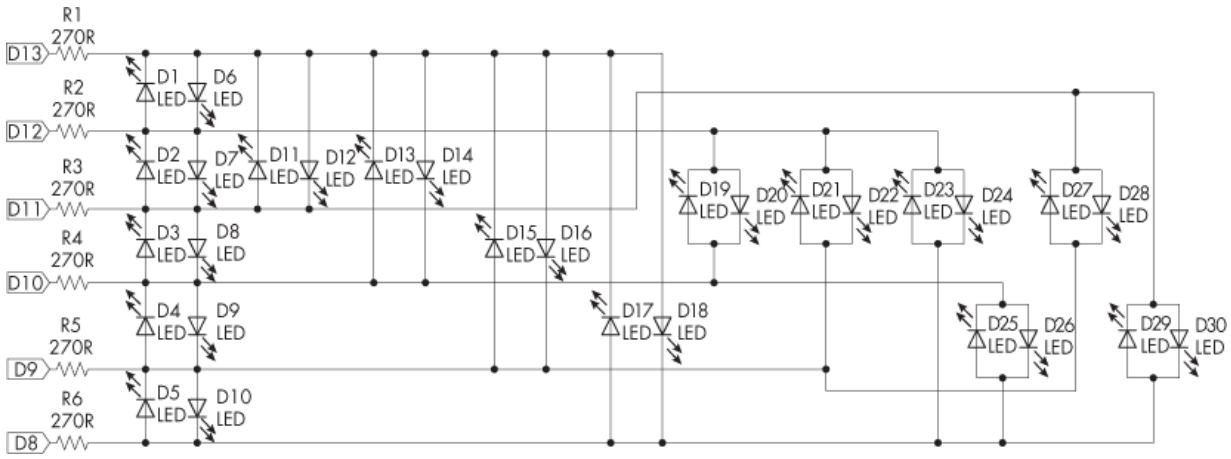


Figure 5-7: The schematic for Project #18

If you ordered the PCB, assembly is simple. You can see from the layout on the board that all the LEDs have the anode pin pointing toward the top and the cathodes (the flat side) toward the bottom, as shown in Figure 5-8.

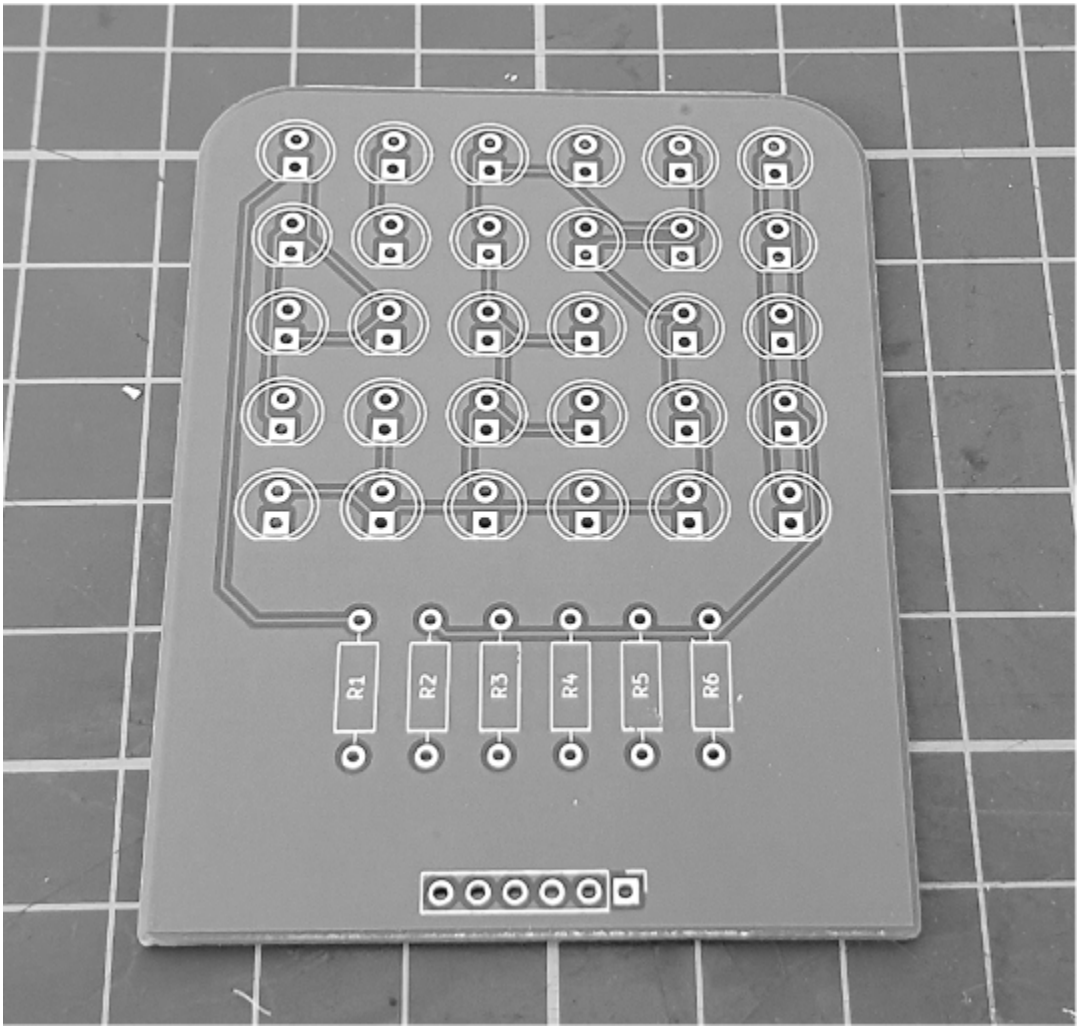


Figure 5-8: The PCB for Project #18

Start with the resistors first, then the LEDs, followed by the inline header pins. Place the pins so they emerge from the rear, and then prop the PCB up with jumper wires to the Arduino, or just place the PCB directly into the Arduino itself, as shown in Figure 5-9. The pins on the PCB are labeled with matching Arduino pin numbers on the rear of the PCB.

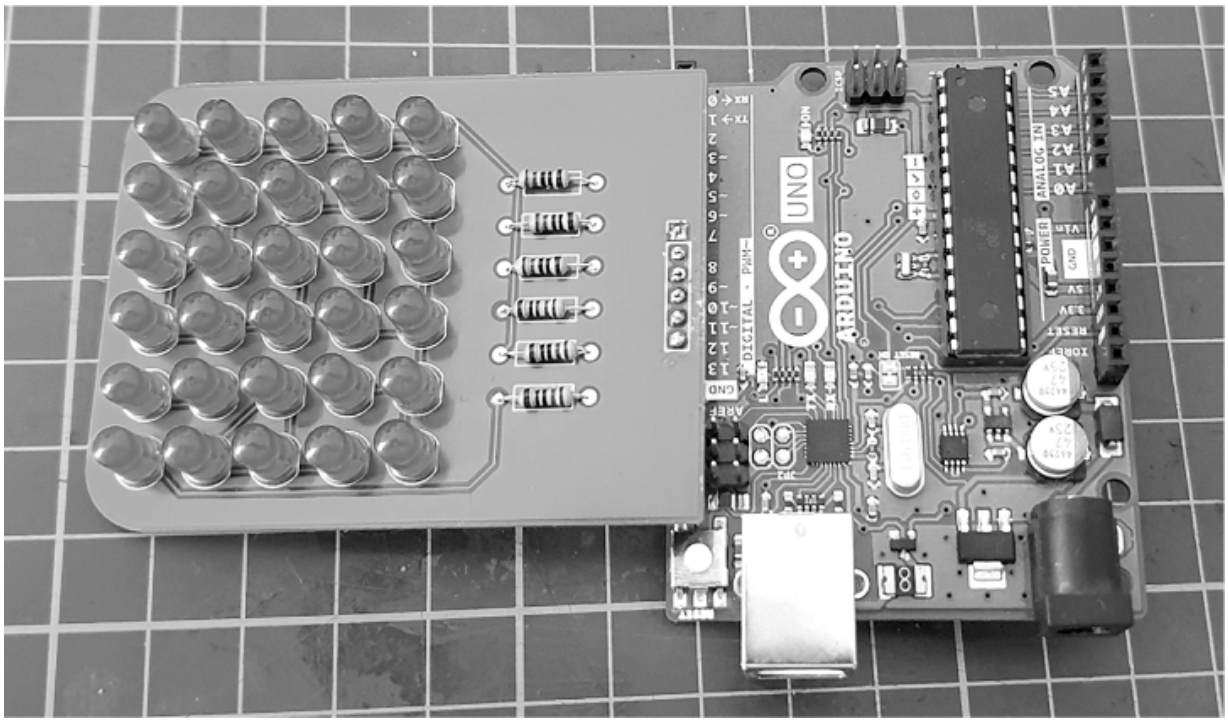


Figure 5-9: The completed Project #18 circuit

Table 5-3 is the logic table for Project #18, which you can compare with the port manipulations in the sketches for Listing 5-2 and that for Project #18 listed shortly afterward or the schematic in Figure 5-7. Again, for practice, trace the current for various LEDs through the schematic and compare with the table to increase your familiarity with controlling the LEDs.

Table 5-3: Logic Table for LED Control in Project #18

LED#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
D8	Z	Z	Z	Z	H	Z	Z	Z	Z	L	Z	Z	Z	Z	Z	Z	H	L	Z	Z	Z	Z	H	L	H	L	Z	Z	H	L
D9	Z	Z	Z	H	L	Z	Z	Z	L	H	Z	Z	Z	Z	H	L	Z	Z	Z	Z	H	L	Z	Z	Z	Z	H	L	Z	Z
D10	Z	Z	H	L	Z	Z	Z	L	H	Z	Z	Z	H	L	Z	Z	Z	Z	H	L	Z	Z	Z	Z	L	H	Z	Z	Z	Z
D11	Z	H	L	Z	Z	Z	L	H	Z	Z	H	L	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	L	H	Z	Z
D12	H	L	Z	Z	Z	L	H	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	L	H	L	H	L	H	Z	Z	Z	Z	L	H
D13	L	Z	Z	Z	Z	H	Z	Z	Z	Z	L	H	L	H	L	H	L	H	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z

Before moving on, please make sure your LEDs are working correctly by entering and uploading the sketch from Listing 5-2, which should make each LED turn on and off in succession.

```
#define d 250 ❶
void LED1() {DDRB = B00110000; PORTB = B00010000
void LED2() {DDRB = B00011000; PORTB = B00001000
void LED3() {DDRB = B00001100; PORTB = B00000100
void LED4() {DDRB = B00000110; PORTB = B00000010
void LED5() {DDRB = B00000011; PORTB = B00000001
void LED6() {DDRB = B00110000; PORTB = B00100000
void LED7() {DDRB = B00011000; PORTB = B00010000
void LED8() {DDRB = B00001100; PORTB = B00001000
void LED9() {DDRB = B00000110; PORTB = B00000100
void LED10(){DDRB = B00000011; PORTB = B00000010
void LED11(){DDRB = B00101000; PORTB = B00001000
void LED12(){DDRB = B00101000; PORTB = B00100000
void LED13(){DDRB = B00100100; PORTB = B00000100
void LED14(){DDRB = B00100100; PORTB = B00100000
void LED15(){DDRB = B00100010; PORTB = B00000010
void LED16(){DDRB = B00100010; PORTB = B00100000
void LED17(){DDRB = B00100001; PORTB = B00000001
void LED18(){DDRB = B00100001; PORTB = B00100000
void LED19(){DDRB = B00010100; PORTB = B00000100
void LED20(){DDRB = B00010100; PORTB = B00010000
void LED21(){DDRB = B00010010; PORTB = B00000010
void LED22(){DDRB = B00010010; PORTB = B00010000
```

```
  void LED23(){DDRB = B00010001; PORTB = B00000001
  void LED24(){DDRB = B00010001; PORTB = B00010000
  void LED25(){DDRB = B00000101; PORTB = B00000001
  void LED26(){DDRB = B00000101; PORTB = B00000100
  void LED27(){DDRB = B00001010; PORTB = B00000010
  void LED28(){DDRB = B00001010; PORTB = B00001000
  void LED29(){DDRB = B00001001; PORTB = B00000001
  void LED30(){DDRB = B00001001; PORTB = B00001000

  void noLED(){DDRB = B00000000;} // All LEDs to i

  void setup() {}

  void loop()
  {
  LED1(); delay(d); LED2(); delay(d); LED3(); dela
  LED5(); delay(d); LED6(); delay(d); LED7(); dela
  LED9(); delay(d); LED10(); delay(d); LED11(); de
  LED13(); delay(d); LED14(); delay(d); LED15(); d
  LED17(); delay(d); LED18(); delay(d); LED19(); d
  LED21(); delay(d); LED22(); delay(d); LED23(); d
  LED25(); delay(d); LED26(); delay(d); LED27(); d
  LED29(); delay(d); LED30(); delay(d);
  }
```

Listing 5-2: Testing the 30-LED matrix display

The sketch sets the delay between each LED lighting up and turning off by defining `d` ❶. From ❷ onward, functions set the GPIO pin port to control each required pin to activate the LEDs in turn. The function ❸ turns all the LEDs off. After that, the main loop endlessly activates each LED in turn. I've minimized the code spacing for display efficiency.

Now that you've tested the LEDs, you're ready to enter and upload the project sketch itself. (Rather than typing the code yourself, I recommend you download it from the book's web page, *https://nostarch.com/arduino-arduinians*, if you haven't already done so.) The digits 0 through 9 should appear on the LED display in succession and then start again at 0.

```
// Project #18 - Using the 30-LED matrix display

int pixels[10][30] = ❶
{ {0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, (
  {0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, (
  {0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, (
  {1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, (
  {1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1,
  {1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, (
  {0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, (
  {1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, (
  {1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1,
```

```
  {1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1,

void displayDigits(int i, int duration) ❷
{
    for (int a = 0; a < duration; a++)
    {
        for (int b = 0; b < 30; b++)
        {
            if (pixels[i][b] == 1)
            {
                turnOnLED(b + 1); ❸
                delay(1); // Adjust as required
            }
        }
    }
    noLED();
}

void LED1() {DDRB = B00110000; PORTB = B00010000
void LED2() {DDRB = B00011000; PORTB = B00001000
void LED3() {DDRB = B00001100; PORTB = B00000100
void LED4() {DDRB = B00000110; PORTB = B00000010
void LED5() {DDRB = B00000011; PORTB = B00000001
void LED6() {DDRB = B00110000; PORTB = B00100000
void LED7() {DDRB = B00011000; PORTB = B00010000
void LED8() {DDRB = B00001100; PORTB = B00001000
void LED9() {DDRB = B00000110; PORTB = B00000100
void LED10(){DDRB = B00000011; PORTB = B00000010
```

```
void LED11(){DDRB = B00101000; PORTB = B00001000
void LED12(){DDRB = B00101000; PORTB = B00100000
void LED13(){DDRB = B00100100; PORTB = B00000100
void LED14(){DDRB = B00100100; PORTB = B00100000
void LED15(){DDRB = B00100010; PORTB = B00000010
void LED16(){DDRB = B00100010; PORTB = B00100000
void LED17(){DDRB = B00100001; PORTB = B00000001
void LED18(){DDRB = B00100001; PORTB = B00100000
void LED19(){DDRB = B00010100; PORTB = B00000100
void LED20(){DDRB = B00010100; PORTB = B00010000
void LED21(){DDRB = B00010010; PORTB = B00000010
void LED22(){DDRB = B00010010; PORTB = B00010000
void LED23(){DDRB = B00010001; PORTB = B00000001
void LED24(){DDRB = B00010001; PORTB = B00010000
void LED25(){DDRB = B00000101; PORTB = B00000001
void LED26(){DDRB = B00000101; PORTB = B00000100
void LED27(){DDRB = B00001010; PORTB = B00000010
void LED28(){DDRB = B00001010; PORTB = B00001000
void LED29(){DDRB = B00001001; PORTB = B00000001
void LED30(){DDRB = B00001001; PORTB = B00001000

void noLED(){DDRB = B00000000;} // all LEDs to i

void turnOnLED(int l)
{
    switch (l)
    {❻
        case 1 : LED1(); break;
```

```
        case 2 : LED2(); break;
        case 3 : LED3(); break;
        case 4 : LED4(); break;
        case 5 : LED5(); break;
        case 6 : LED6(); break;
        case 7 : LED7(); break;
        case 8 : LED8(); break;
        case 9 : LED9(); break;
        case 10 : LED10(); break;
        case 11 : LED11(); break;
        case 12 : LED12(); break;
        case 13 : LED13(); break;
        case 14 : LED14(); break;
        case 15 : LED15(); break;
        case 16 : LED16(); break;
        case 17 : LED17(); break;
        case 18 : LED18(); break;
        case 19 : LED19(); break;
        case 20 : LED20(); break;
        case 21 : LED21(); break;
        case 22 : LED22(); break;
        case 23 : LED23(); break;
        case 24 : LED24(); break;
        case 25 : LED25(); break;
        case 26 : LED26(); break;
        case 27 : LED27(); break;
        case 28 : LED28(); break;
        case 29 : LED29(); break;
```

```
            case 30 : LED30(); break;
        }
    }

    void setup(){}

    void loop()
    {
        for (int q = 0; q < 10; q++) ❼
        {
            displayDigits(q, 50); ❽
            delay(250);
        }
    }
```

The two-dimensional array ❶ contains 10 arrays, one for each digit from 0 through 9. Each digit array has 30 elements, to match the 30 LEDs on the display board. The display board's LEDs are arranged from 1 at the top left to 30 at the bottom right. In the arrays, each `1` represents the LEDs that need to turn on in quick succession for each number to be displayed.

The custom `displayDigits()` function ❷ accepts the digit you want to display and the number of times the function will display the LEDs required in turn to show the digit. The greater the number in the second parameter, the longer the digit will

appear on the display. The `displayDigits()` function uses another custom function, `turnOnLED()` ❸, to turn on the required LED. The `turnOnLED()` function, in turn, uses a `switch…case` ❻ to call the appropriate function from the list ❺ and turn on each individual LED.

The `delay()` function ❹ within `displayDigits()` is set to keep each LED on for 1 millisecond before the next is illuminated. Because the LEDs flip on and off so quickly, the human eye perceives all the LEDs required for a given digit as on at the same time, as shown in Figure 5-10. You can alter the length of the delay to tailor the display to your own preferences.

Finally, in `void loop()`, the digits 0 through 9 are displayed in order ❼, with a short delay in between. Each digit is displayed using 50 refreshes of the digit ❽.

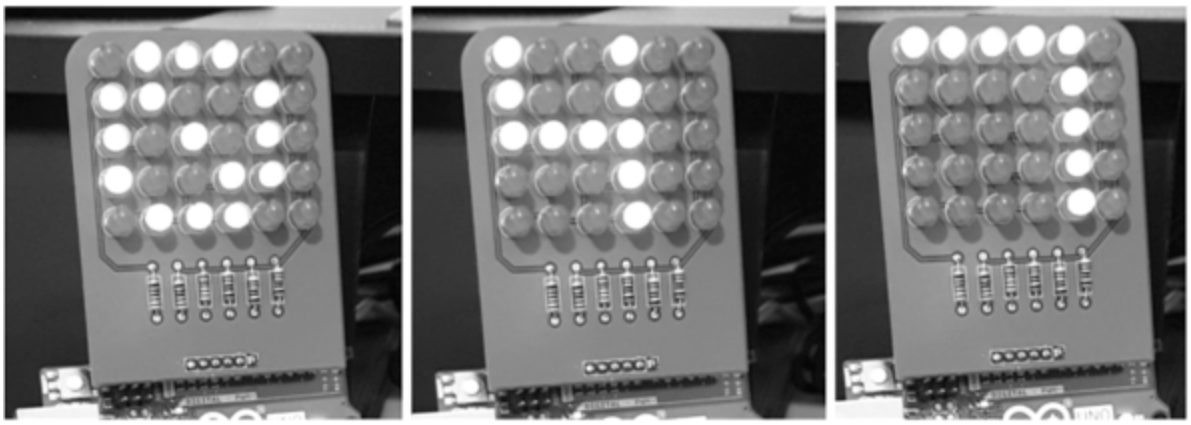


Figure 5-10: Examples of Project #18

This sketch displays digits using the Monomin font, as shown in Figure 5-11.



Figure 5-11: The Monomin 6×5 font digits

You can find more information about this font by visiting *https://fontstruct.com* and searching for the Monomin 6 × 5 font.

Making Custom Character Displays

It's possible to create display types beyond just the numbers shown in Project #18. To make your own characters, first draw how you would like them to appear using some grid paper or a spreadsheet. For example, to create a boundary rectangle, draw the diagram shown in Figure 5-12.

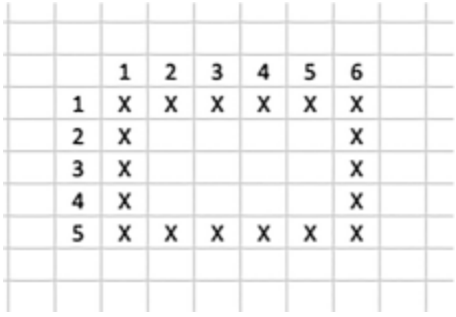


Figure 5-12: Planning the character to display

The top-left X represents LED 1, and the bottom-right X represents LED 30. You then create your own array that represents the status of each LED. For this example, the array would look like this:

```
  {1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0,
```

You can then add or swap this array into the sketch in the same way as for the other digits in Project #18 and use the `displayDigits()` function to show your custom character on the display, as demonstrated in Figure 5-13.

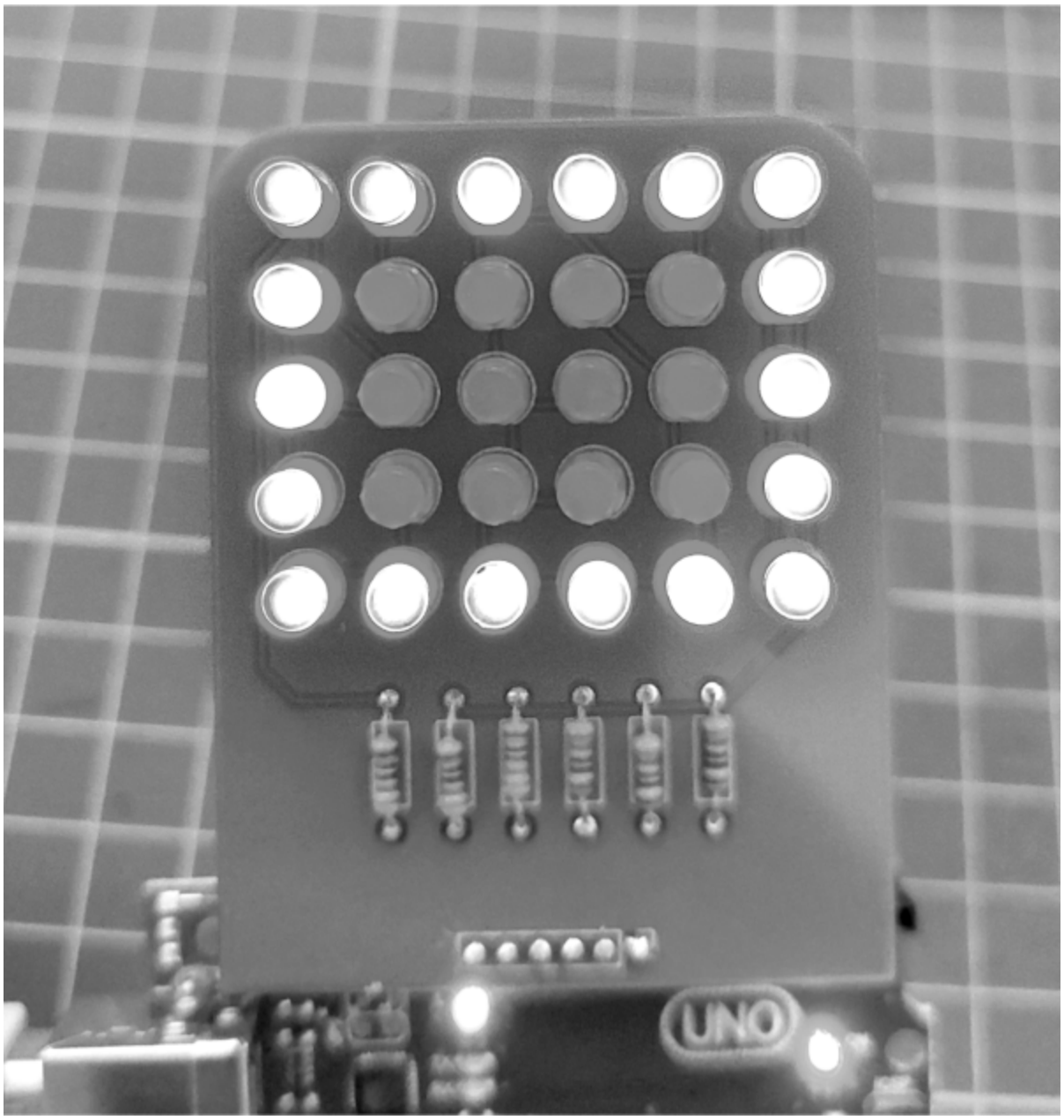


Figure 5-13: The custom character

When designing your own custom characters, it can help to use a spreadsheet or graph paper to plan the layout of the LEDs you'd like to turn on and off.

Moving On

With Charlieplexing in your toolbox, you can now control multiple LEDs with minimal I/O pins and without using external ICs. You learned how to control 2, 6, 12, or 30 LEDs with a minimum of digital output pins, as well as how to use Charlieplexing with the compact ATtiny microcontrollers.

In the next chapter, you'll learn to give your Arduino the power to turn itself off.

6

ADDING PROFESSIONAL POWER CONTROL



Instead of controlling power to your Arduino with a switch or USB cable, you can use *soft power control*: turning power on and off with buttons or having your Arduino turn itself off via the sketch or external sensor. Soft power increases the professionalism of your projects. If you're designing, say, a fancy hi-fi amplifier, a pair of on/off buttons looks far nicer than the type of on/off switch you'd find on industrial equipment.

This chapter introduces various methods of soft power control. You'll learn to:

- Control larger currents with MOSFETs
- Use the 555 timer IC in bistable mode
- Turn on Arduino circuits with external devices
- Conserve power by building projects that turn on only when required
- Use the DS3231 real-time clock (RTC) library for Arduino

You'll also create a soft on/off switch for your Arduino and build a low-power event logger.

Power the Arduino without a Physical Switch

It's common to supply power to Arduino projects with the USB socket, battery pack, or external AC adapter. Turning the project on and off in these cases means removing a cable or DC plug, which works well if you're just experimenting or building something for your own use. However, soft power control allows you to control the power to your Arduino project by using buttons or signals from other devices or allowing an Arduino project to turn itself off completely. This doesn't require a "hard" physical switch to connect or cut current flow.

Soft power control saves on hardware wear and tear, as there are no moving parts for power control, and makes interfaces

simpler and easier to use. Turning power on and off with buttons or further automating device controls creates a better user experience.

This section further discusses two types of components that are required for these soft power control circuits: the MOSFET, used for switching current; and the 555 timer IC, in this case used as an interface between on and off signals in soft power control.

Operating MOSFETs

MOSFETs allow you to switch large voltages and currents with a small signal, such as that from our Arduino's digital output pins. They are available in various sizes, such as those shown in Figure 6-1.

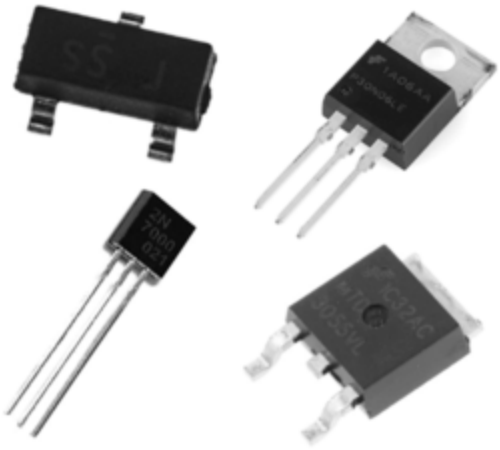


Figure 6-1: Various MOSFETs

The bottom left of Figure 6-1 shows the small 2N7000 N-MOSFET from Project #14 in Chapter 4, which we'll also use in this chapter. Looking at the front of the 2N7000 (the flat-faced side), the three pins are, from left to right:

- Source
- Gate
- Drain

Figure 6-2 shows the schematic symbol for the 2N7000 MOSFET.

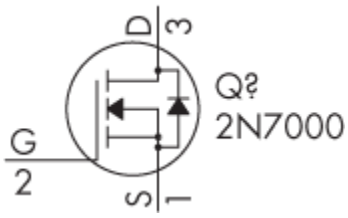


Figure 6-2: The schematic symbol for a 2N7000 MOSFET

Operating a MOSFET is easy. When you apply a small current to the gate pin, a large current can flow in through the drain pin and out through the source pin. You can also use PWM with a MOSFET, allowing various controls of lights, motors, and more. Your 2N7000 MOSFET can handle up to 60 V DC at 200 mA continuously, or 500 mA in bursts. When choosing a MOSFET for other projects, be sure to check the voltage and current maximums against the signal you want to switch.

You can use larger MOSFETs to control larger currents, such as the IRF520 shown in Figure 6-3.



Figure 6-3: An IRF520 N-MOSFET

When using MOSFETS, be sure to check the datasheet for the pinouts, as they vary between units. You can find larger MOSFETs in breakout-board form for convenient prototyping. Figure 6-4 shows two examples: on the left is the N-MOSFET module from Freetronics, and on the right is an IRF520 breakout from PMD Way (part 759300).

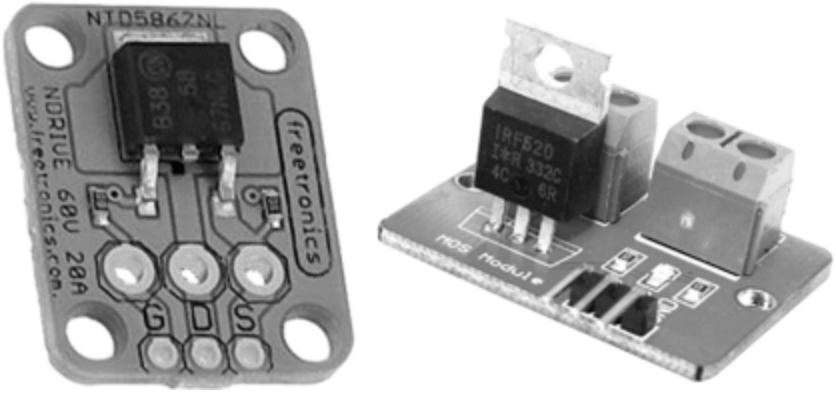


Figure 6-4: Two example MOSFET breakout boards

In this chapter, you'll use the 2N7000 to control the power supply of your Arduino. You'll also connect a 10 kΩ resistor between the gate and the source pins to keep the gate switched off when a current is not applied to it, stopping the MOSFET from turning slightly on or off at random.

Using the 555 Timer IC in Bistable Mode

To create an on/off switch, you can harness the 555 timer IC (originally used in Chapter 4) in *bistable mode*, configuring the circuit to use two buttons to toggle the output pin HIGH or LOW. The output will then be used to control the MOSFET, which will be used to switch power going to the Arduino on and off.

Figure 6-5 shows the bistable 555 circuit.

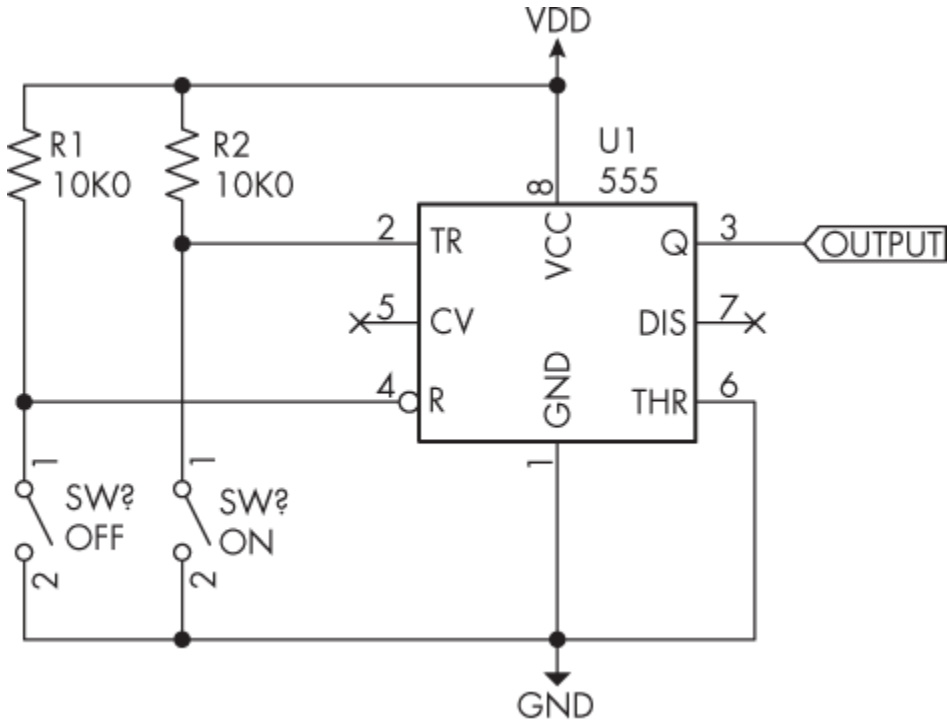


Figure 6-5: The schematic for a 555 timer IC in bistable mode

Pulling pin 2 (the trigger pin) to GND with SW_1 will turn on the output, while pulling pin 4 (the reset pin) to GND with SW_2 turns off the output. That's all you need for a soft on/off switch! The circuit can operate on anything between 5 V and 12 V DC. Build it yourself and test it, perhaps using an LED and 1 kΩ resistor on the output as an indicator.

For the projects in this chapter, however, you need to switch the power to an entire Arduino. In the following project, instead of using the output pin as a power supply, you'll switch the 2N7000 N-MOSFET on and off with the bistable circuit, allowing you to control more current.

Project #19: Creating a Soft On/Off Switch

In this project, you'll create a soft on/off switch for your Arduino. You can use a power supply of between 9 and 12 V DC to operate the 555; the Arduino will regulate the voltage to its required 5 V.

You can build this circuit using a solderless breadboard for a more temporary experiment or using a custom PCB created with the downloadable project files if you'd like a more permanent version. You'll use the parts listed here for the remaining projects in this chapter as well:

- An Arduino Uno or compatible board and USB cable
- A 9 to 12 V DC, 1A power supply, wall wart, or plugback
- One solderless breadboard or Project #19 PCB
- Three 10 kΩ, 0.25 W, 1 percent resistors
- One 2N7000 N-MOSFET
- A 555 timer IC (and 8-pin IC socket if using a PCB)
- Two tactile buttons
- 1 × 40 2.54 mm inline header pins (if using a PCB)
- A PCB-mount DC socket (if using a PCB)
- Male-to-male jumper wires

If you're using a power supply with a DC plug, such as a wall wart, you may want to use a DC socket breakout such as PMD Way part 51201299 to avoid having to cut the plug off the end of the power lead, as shown in Figure 6-6.

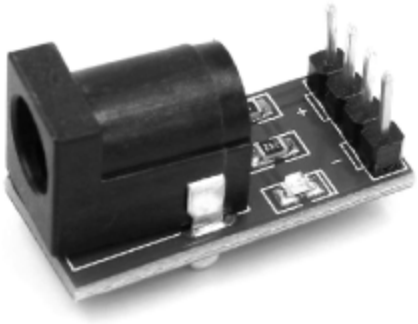


Figure 6-6: A DC socket breakout

Assemble the circuit as shown in Figure 6-7.

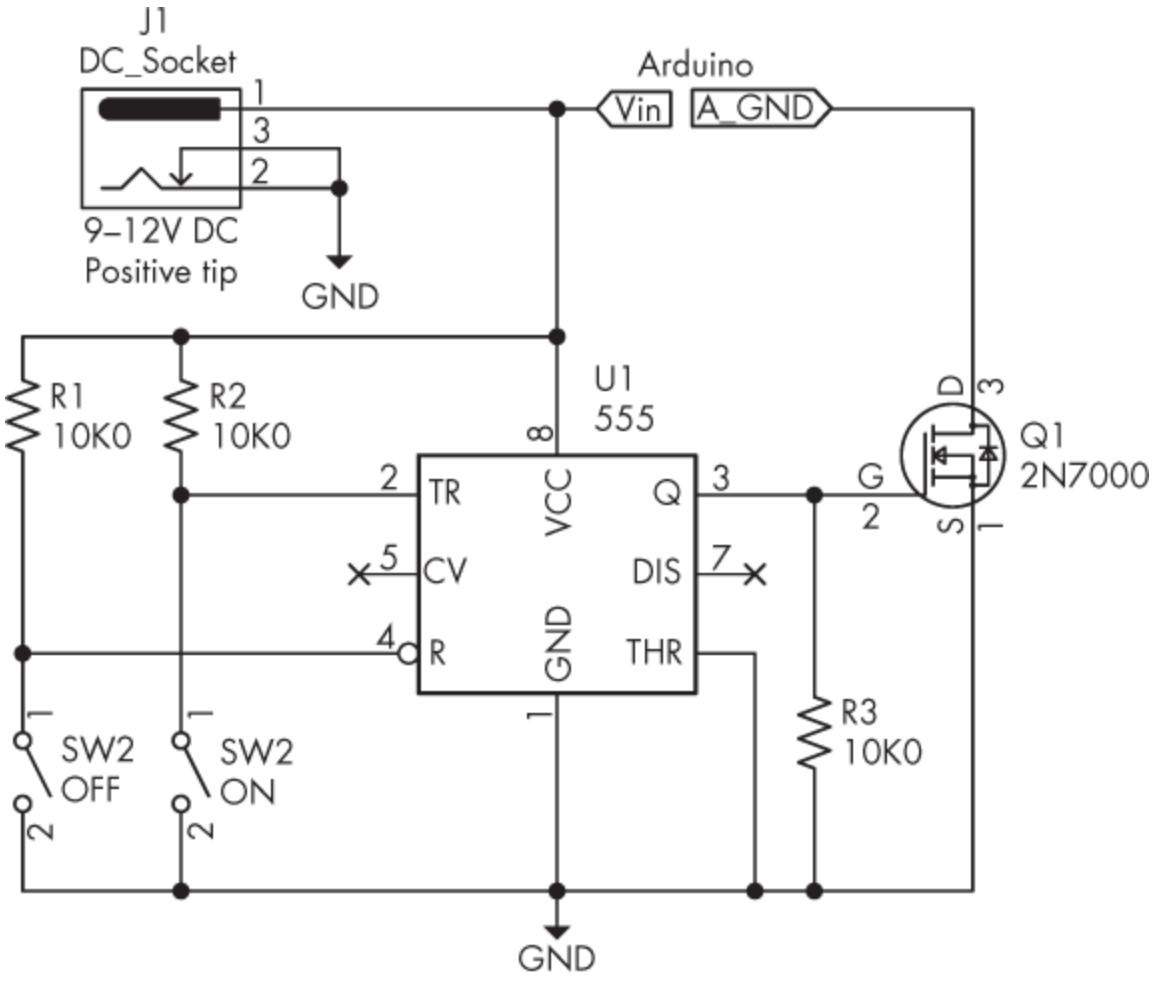


Figure 6-7: The schematic for Project #19

If you are using a solderless breadboard, the DC socket is just a way of getting the required 9 to 12 V DC to the circuit. The Vin and GND labels at the top right of the schematic are connections to the Arduino's Vin and GND pins, respectively.

If you are using the PCB, shown in Figure 6-8, assembly is straightforward. Start with the resistors first, then the buttons, and then the IC socket, DC socket, and inline header pins. (There

are a few extra connections not listed for this project, which you'll use in Project #20.)

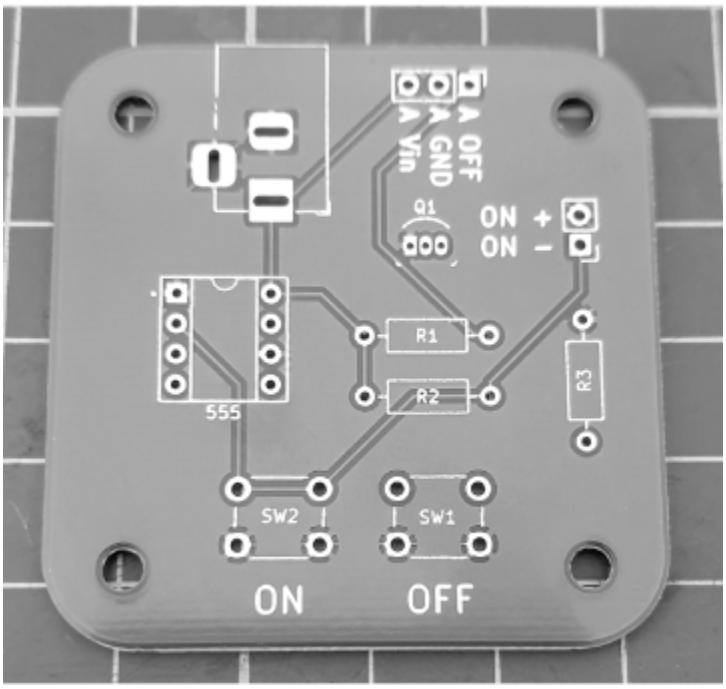


Figure 6-8: The PCB for Project #19

Once assembled, your PCB should resemble that shown in Figure 6-9.

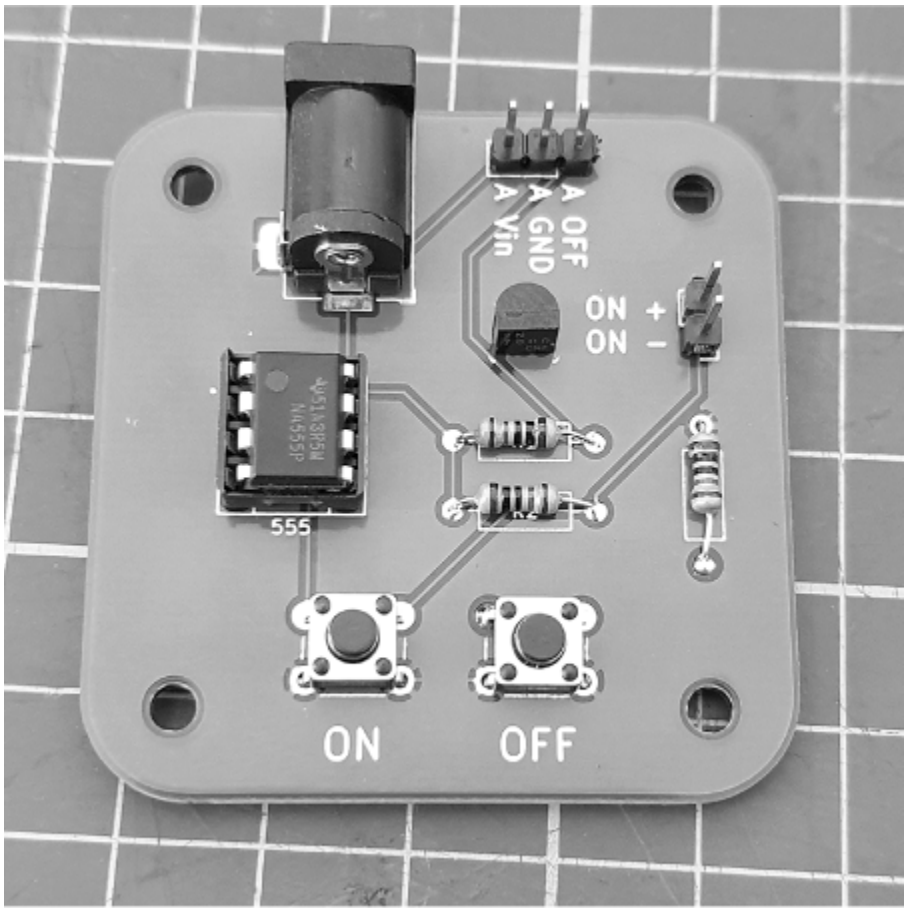


Figure 6-9: The completed PCB for Project #19

Connect the Arduino to the PCB and then connect the external power of between 6 and 12 V DC. You can then simply turn the Arduino and off with switches SW_1 and SW_2, respectively. The Arduino's power-on LED will tell you at a glance whether the Arduino is on or off.

As mentioned earlier, when you press SW_2, the 555's output should go HIGH. In this circuit, current flows to the gate (G) pin of the N-MOSFET, which then turns on the MOSFET, which then

allows current to flow through the drain (D) pin to the source (S) pin. (Even if the Arduino is switched off, the circuit itself will draw around 5 mA of current at 9 V DC.) When the MOSFET is on, current can flow from the external power supply into the Arduino via the Vin pin, out via its GND pin, through the MOSFET, and finally to GND, completing the power circuit. When you press SW_1, the 555's output goes LOW and the MOSFET is switched off. This means no current can flow through the Arduino, turning it off as well.

If you're using a solderless breadboard, keep the circuit together for the rest of the chapter. You can also use this circuit with other devices if they meet the same power requirements. For now, let's use this circuit to give the Arduino the ability to turn itself off.

Project #20: Powering Off the Arduino Automatically

As an extension to the soft on/off circuit used in Project #19, you can have your Arduino turn itself off by pulling pin 4 of the 555 to LOW (as you did with SW_1 in the previous project). This is useful for creating applications you don't want to run continuously, from simple games to more complex data-logging projects that need to write data only when an event occurs.

To modify the Project #19 circuit for this purpose, first connect digital output pin D12 to the junction of 555 pin 4 and resistor R_1 to become the Off Signal connection if using the solderless breadboard, as shown in Figure 6-10, or to the pin on the PCB labeled A OFF.

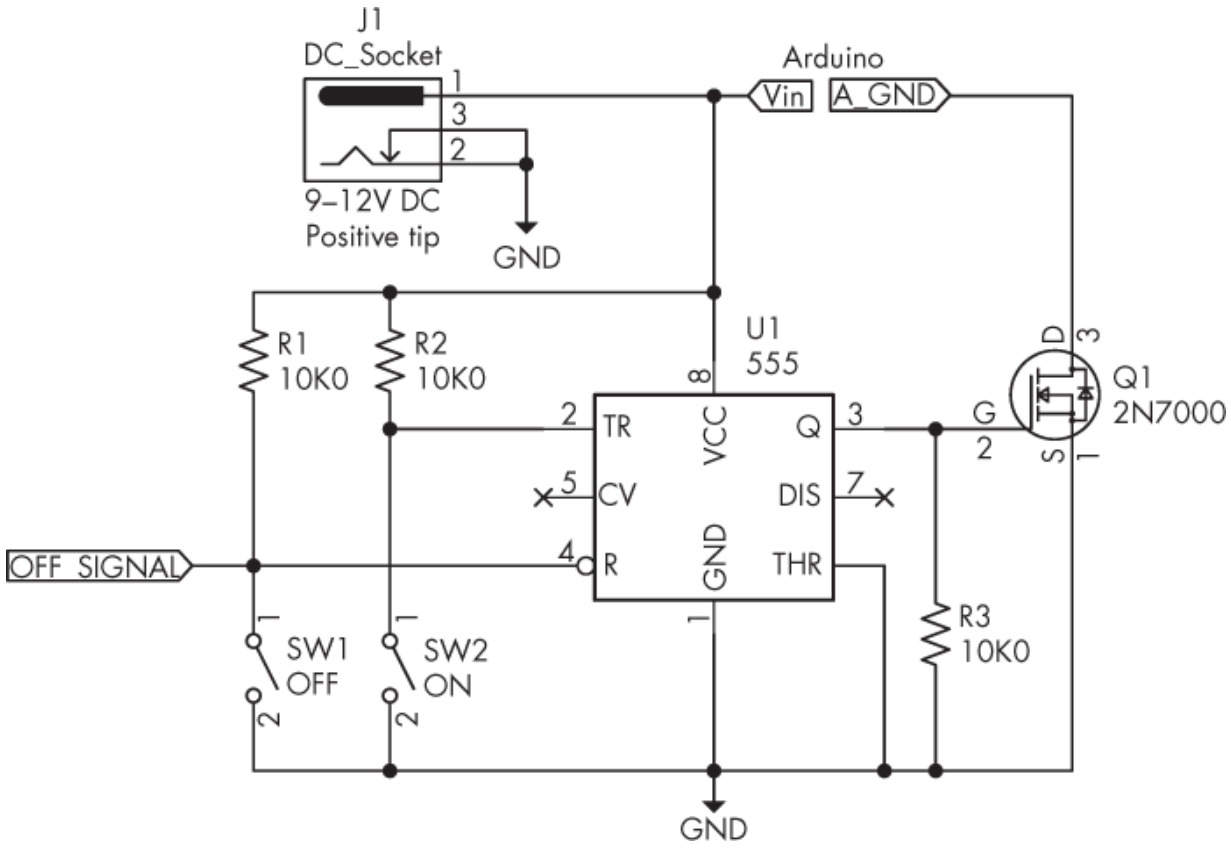


Figure 6-10: The Project #20 circuit with Arduino self-off contact

Connect the Vin and GND as well. Turn on the Arduino (which should light up the onboard power LED), and the D13 LED should also turn on. After five seconds the D13 LED should turn off; then, a second later, the Arduino should turn itself off.

Let's see how this works:

```
  // Project #20 - Arduino self power off

❶ void turnOff()
  {
      digitalWrite(12, LOW);
  }

  void setup()
  {
      pinMode(12, OUTPUT);
      pinMode(13, OUTPUT);
      digitalWrite(12, HIGH);
  }

  void loop()
  {
    ❷ digitalWrite(13, HIGH);
      delay(5000);
    ❸ digitalWrite(13, LOW);
      delay(1000);
    ❹ turnOff();
  }
```

The sketch first ensures D12 is HIGH in `void setup()`. Next, a simple custom function `turnOff()` ❶ drops D12 to LOW,

triggering the 555 to turn off the MOSFET and thus turn off the power to the Arduino. The sketch turns the D13 LED on ❷ and off ❸ as an example of activity; then the Arduino turns itself off ❹.

Activating an Arduino from an External Device

So far, you've learned how an Arduino can shut itself off. I'll now explain how to program an external device or sensor to activate power to an Arduino, which completes a task and then powers itself off, ready for the next event.

Normally Open Contact Devices

The switch used in Projects #18 and #19 closes the circuit between 555 pin 2 and GND. This means you can replace the switch with (or use in parallel with) something that has normally open (NO) contacts—in other words, a device that closes a circuit in the same way a press-button or switch does. This could be anything such as a simple doorbell button, a pressure mat that acts as a switch when walked on, a passive infrared (PIR) motion sensor that detects movement with relay

output contacts, a normally open reed switch, a door magnet that can detect the opening of a door, and more.

Any device that can close the two points in the circuit will activate your Arduino. Simply connect the two wires from the device to 555 pin 2 and GND, respectively, as shown in Figure 6-11.

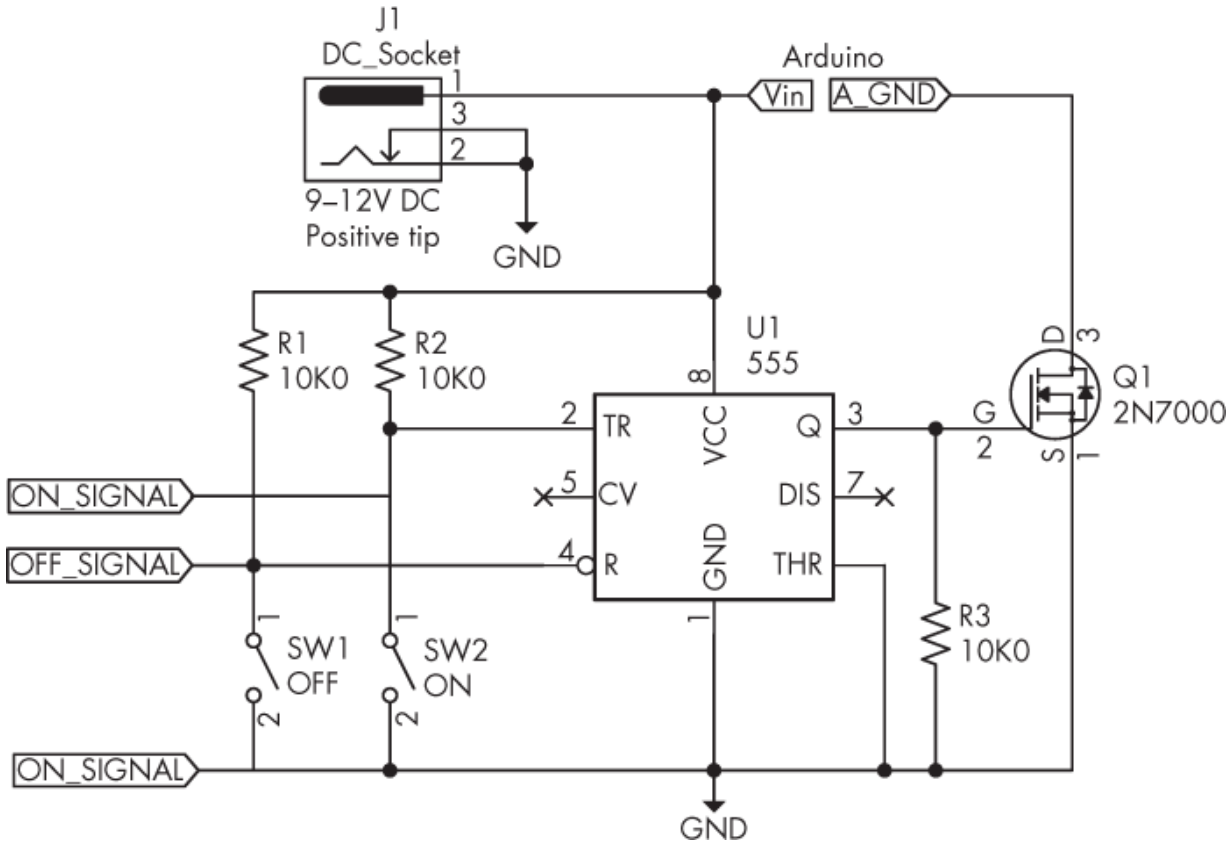


Figure 6-11: The schematic for a soft on/off circuit with external trigger connections

The two contacts in question are shown as ON_SIGNAL points in the schematic. If you're using the project PCB for this chapter,

these are the ON+ and ON– pins.

Output Logic Devices

You may want to have your Arduino turned on by devices that give out an electrical signal when they're activated, such as a logic-level 5 V output. These devices include some passive infrared (PIR) motion detectors, thermostats, some industrial programmable logic controller (PLC) device outputs, and more. However, such devices require a little more circuitry to bridge the gap for activation, as their output signal isn't the same as the normally open method described in the previous section.

To take advantage of these sorts of devices, you'll use their logic output to switch another 2N7000 MOSFET to bridge the gap and act as the On switch for you. The devices' output voltage must be high enough to turn on the MOSFET, so be sure to check this with the device supplier or datasheet before getting started. The MOSFET you'll use in the next project requires 3 to 5 V DC to activate.

Figure 6-12 shows an example circuit, with the requisite additional MOSFET and inputs on the left side. The TRIGGER point is connected to the device's output, and TR_GND is connected to the device's negative or GND connection.

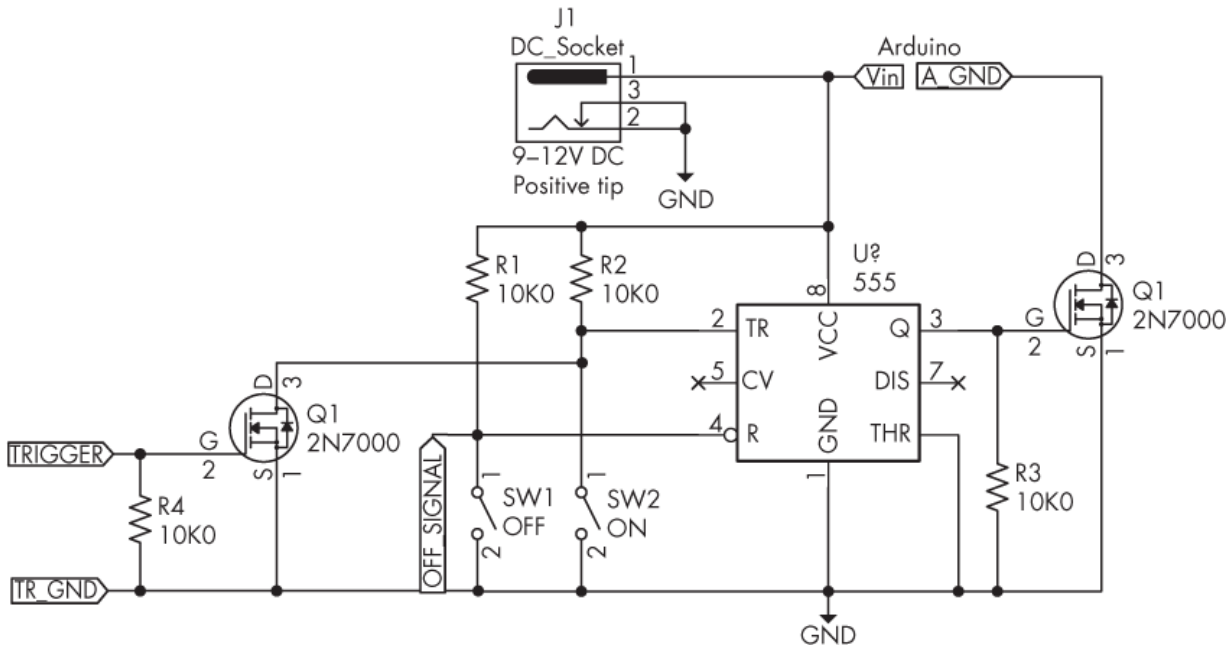


Figure 6-12: The soft on/off schematic with logic level trigger

Now before moving onto the next project, I'll show you an easier way to use DS1307 and DS3231 real-time clock ICs by installing a useful Arduino library.

The DS3231 Real-Time Clock Library

The next project uses a DS1307 or DS3231 real-time clock IC to keep time for a data-logging project. To simplify this, you'll now learn how to use the matching DS3231 RTC library to save coding effort and space. This library works similarly to that for the DS1307.

First, download the library code ZIP file from *https://github.com/NorthernWidget/DS3231*. Open the Arduino IDE and select **Sketch** ▸ **Include Library** ▸ **Add .ZIP library**. Navigate to the downloaded file and then click **OK** to install the library. You can check whether the library has installed by selecting **File** ▸ **Examples** in the IDE; a new DS3231 option should be visible.

Enter the sketch from Listing 6-1 to test the library and the basic functions to set and retrieve time and date information, but don't upload it yet:

```
❶ #include <Wire.h>
  #include <DS3231.h>
  DS3231 RTC;

❷ bool century = false;
  bool h12Flag = false;
  bool pmFlag = false;
  // Change the following for your own time
  byte year = 22;
  byte month = 9;
  byte date = 2;
  byte hour = 11;
  byte minute = 8;
  byte second = 0;

  void setTimeData()
```

```
{
    RTC.setYear(year);
    RTC.setMonth(month);
    RTC.setDate(date);
    RTC.setHour(hour);
    RTC.setMinute(minute);
    RTC.setSecond(second);
    RTC.setClockMode(false); // Set clock to 24 h
}

void setup()
{
    Wire.begin();
 ❸ //setTimeData(); // Set time and date
    Serial.begin(9600);
}

void loop()
{
    // Display data on Serial Monitor
    Serial.print(RTC.getDate());
    Serial.print("/");
    Serial.print(RTC.getMonth(century));
    Serial.print("/20");
    Serial.print(RTC.getYear());
    Serial.print(" - ");
    Serial.print(RTC.getHour(h12Flag, pmFlag));
    Serial.print(":");
```

```
    ❹ if (RTC.getMinute() < 10)
      {
          Serial.print("0");
      }
      Serial.print(RTC.getMinute());
      Serial.print(":");
    ❺ if (RTC.getSecond() < 10)
      {
          Serial.print("0");
      }
      Serial.println(RTC.getSecond());
      delay(1000);
  }
```

Listing 6-1: Testing the DS3231 RTC library

The sketch first includes the I^2C and RTC libraries and then creates an instance of the real-time clock to reference ❶. It then declares the required variables to hold the time and data information ❷. These include three Boolean variables, which are left as false by default, as you're using 24-hour time.

To set the time, the custom function `setTimeData()` contains the functions to set all the time and date parameters. The variables declared earlier are placed in each matching function. This function needs to be called only once ❸, as you'll initially set

the time and date and then comment out the function before reuploading the sketch. Otherwise, the clock will reset to the value of the variables ❷.

To retrieve the time, the sketch uses a series of functions that begin with `RTC`, such as `RTC.getdate()`, which return their matching data. For a natural-looking display of the minutes and seconds, the sketch shows a 0 before the values for minutes under 10 ❹ and seconds under 10 ❺.

To change the time and date values to match your own time zone, enter your own values for time and date ❷, then uncomment the function `setTimeData()`❸, and finally upload the sketch. Now re-comment the `setTimeData()` function and save and re-upload the sketch. Once completed, open the Serial Monitor at 9,600 bps, and you should see the date and time being updated every second.

Let's combine these methods of triggering power to the Arduino and logging the events in a final project.

Project #21: Building an Event Logger

For this project, imagine that someone needs to prove their arrival at a certain place on a regular basis. This might be a

child arriving at school or activities, a security guard checking into work for the night, or an employee who must open a store at a particular time in the morning. To create a device to record the time and date of these events, this project uses a soft on/off circuit with an Arduino and a DS1307 or DS3231 real-time clock IC and SD card socket for logging the data to memory card. The trigger will be a simple button wired a distance away from the Arduino, like that included with a wired doorbell.

As an experienced Arduino user, you might build a circuit like the one in this project with slightly different products than those used here. For example, you could build a similar circuit with either an SD card module or an SD card shield (such as PMD Way part 668046, shown in Figure 6-13). You can use the framework you learn in this project for any situation where you want to trigger your Arduino to do something just once and then switch off until the next event.

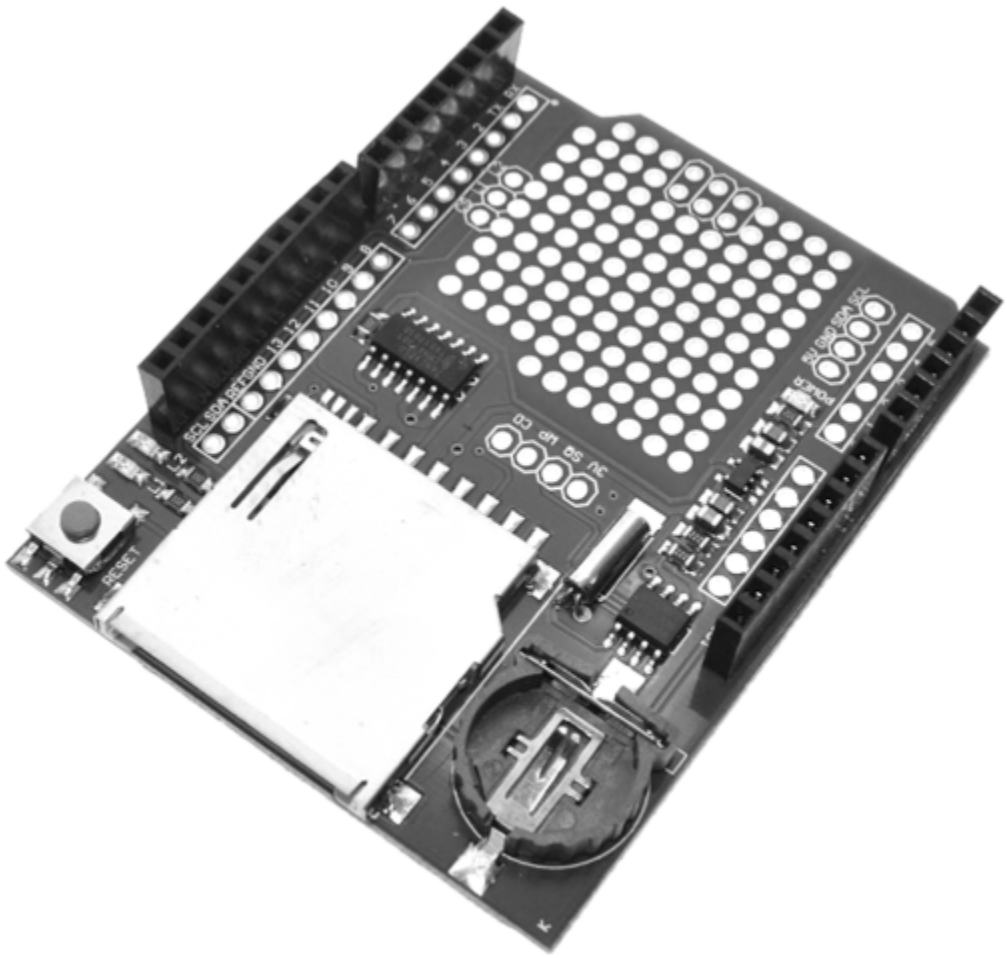


Figure 6-13: An SD and RTC shield for Arduino

You'll use the following hardware:

- An Arduino Uno or compatible board and USB cable
- A 9 to 12 V DC, 1 A power supply, wall wart, or plugpack
- The completed PCB or hardware from Project #20
- A DS1307 or DS3231 real-time clock module and SD or microSD card module, or SD card and RTC shield for Arduino
- A microSD or SD memory card
- A remote button and suitable twin-core wire

Assemble the circuit as shown in Figure 6-14. If you're using the solderless breadboard, you can omit the tactile switches if you don't want local control.

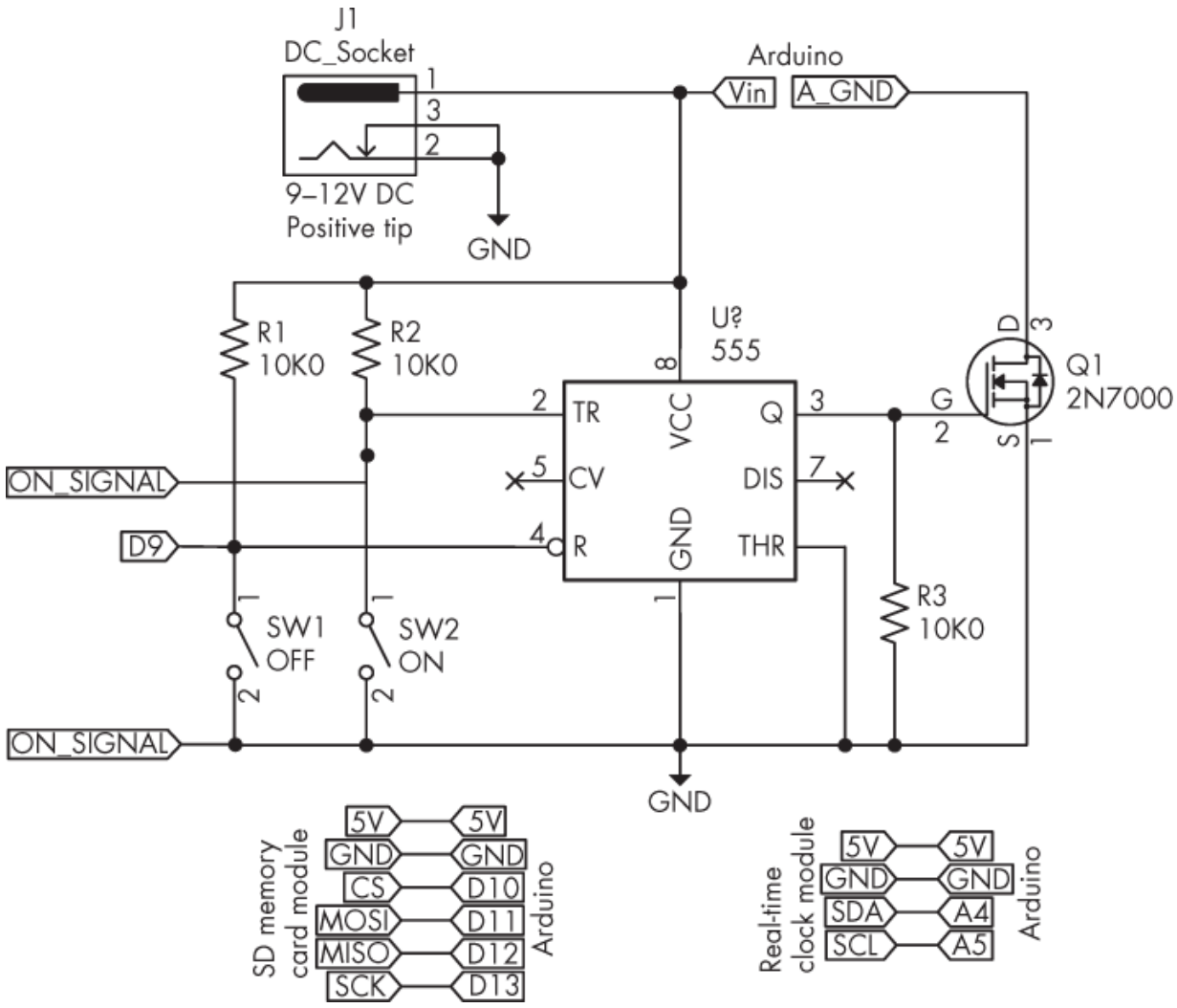


Figure 6-14: The schematic for Project #21

Don't forget to format your memory card correctly.

NOTE

If you're unfamiliar with the use of the SD card or real-time clock module, review Chapters 7 and 20 of Arduino Workshop*, 2nd edition.*

Once you've assembled the hardware, set the current time and date in the Project #21 sketch using the function `setDS3231time()` in `void setup()`. To preserve the time and date in the real-time clock, upload the sketch and then comment out the line and re-upload the sketch, as you did in the previous section.

Let's see how this works:

```
  // Project #21 - Arduino event logger

❶ #include <Wire.h>
  #include <DS3231.h>
  DS3231 RTC;
  #include <SD.h>
  #define offPin 9

❷ bool century = false;
  bool h12Flag = false;
  bool pmFlag = false;
  byte year = 22;
  byte month = 9;
  byte date = 2;
  byte hour = 11;
```

```
byte minute = 8;
byte second = 0;

void setTimeData()
{
    RTC.setYear(year);
    RTC.setMonth(month);
    RTC.setDate(date);
    RTC.setHour(hour);
    RTC.setMinute(minute);
    RTC.setSecond(second);
    RTC.setClockMode(false); // Set clock to 24 h
}

void turnOff()
{
    digitalWrite(offPin, LOW);
}

void logEvent()
{
    // Create the file for writing
    File dataFile = SD.open("DATA.TXT", FILE_WRI
    // If the file is ready, write to it:
    if (dataFile)
    {
        dataFile.print("Event occurred on: ");
        dataFile.print(RTC.getDate());
```

```
        dataFile.print("/");
        dataFile.print(RTC.getMonth(century));
        dataFile.print("/20");
        dataFile.print(RTC.getYear());
        dataFile.print(" - ");
        dataFile.print(RTC.getHour(h12Flag, pmFl
        dataFile.print(":");
        if (RTC.getMinute() < 10) {
            dataFile.print("0");
        }
        dataFile.print(RTC.getMinute());
        dataFile.print(":");
        if (RTC.getSecond() < 10) {
            dataFile.print("0");
        }
    dataFile.println(RTC.getSecond());
    dataFile.close(); // Close the file once the
    }
}

void setup()
{
    Wire.begin();
    //setTimeData(); // Set time and date
    pinMode(offPin, OUTPUT);
    digitalWrite(offPin, HIGH);
    pinMode(10, OUTPUT);
    // Check that the memory card exists and is
```

```
    if (!SD.begin(10))
    {
        // Stop sketch:
        return;
    }
}

❸ void loop()
{
    logEvent(); // Log event to SD card
    delay(1000); // Wait a moment
    turnOff();
}
```

The sketch is a combination of basic real-time clock and SD-card-writing functions, along with self-turn-off once the event has been recorded to the memory card. It first includes the required libraries and then sets the RTC I^2C bus address and the pin number that triggers the 555 timer to turn the circuit off ❶. Next, it sets the required variables for the date and time, along with the function to write the data ❷.

When called, the `turnOff()` function sets the digital pin connected to the 555 to LOW, turning off the circuit. The `logEvent()` function opens the text file on the SD card, retrieves the time and date from the real-time clock, and writes it in a

neat line. The `void setup()` function initializes the real-time clock and allows for setting the time and date. It also initializes the digital pins required for triggering the external circuit and using the SD card and checks that the SD card is ready.

The final loop ❸ operates when the external device circuit triggers the Arduino and turns it on. The `logEvent()` function logs the event, writing the time and date to the SD card. There's a short delay to ensure the data file has closed, after which `turnOff()` turns the Arduino off.

To review the data logged by the project, disconnect the power from the Arduino and insert the SD card into your computer. Open the *DATA.TXT* file to see the events listed in order.

The project has recorded a new line of text containing the date and time for each event.

When operating at 9 V DC, this project draws around 5 mA of current when off and peaks at 70 mA during operation. If you powered it with six AA cells for portable use and had one or two events an hour, the project could easily run for more than seven days.

Moving On

This chapter showed you how to add professional power control to your projects, reducing power consumption by allowing external actions to turn the power on and having your Arduino turn itself off.

The next chapter demonstrates another advanced way to turn projects on and off: safely controlling mains AC current with a wireless remote-controlled outlet.

7

CONTROLLING AC MAINS POWER OUTLETS



In this chapter, you'll learn to safely control mains AC power outlets using various forms of automation and remote control. This technique lets you easily manipulate devices such as lamps, fans, and pumps from a distance, without any direct connection to the mains power wiring.

You'll learn to:

- Use optocouplers to isolate electrical signals
- Hack a wireless AC mains remote-control transmitter and connect it to an Arduino circuit
- Create a timer-controlled wireless AC mains outlet

- Control AC mains outlets remotely with SMS text messages

Optocouplers

An *optocoupler* is a small device that allows the transmission of a signal between parts of a circuit while keeping those two parts electrically isolated. Inside a typical optocoupler are an LED and a phototransistor, a device that responds to light by amplifying an electric current. When a current runs through the LED, it switches on; the phototransistor detects the light and allows another current to flow through it. When the LED is off, current cannot flow through the phototransistor. The two currents are completely electrically isolated throughout the process.

Figure 7-1 shows the schematic symbol for an optocoupler.

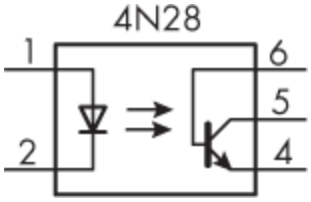


Figure 7-1: The schematic symbol for a typical optocoupler

In the schematic, an LED is connected to pins 1 (the anode) and 2 (the cathode). Pins 4, 5, and 6 are the phototransistor, while

pin 3 is not used. The optocoupler model you'll use, the 4N28, is in a six-pin dual in-line (DIL) package, as shown in Figure 7-2.

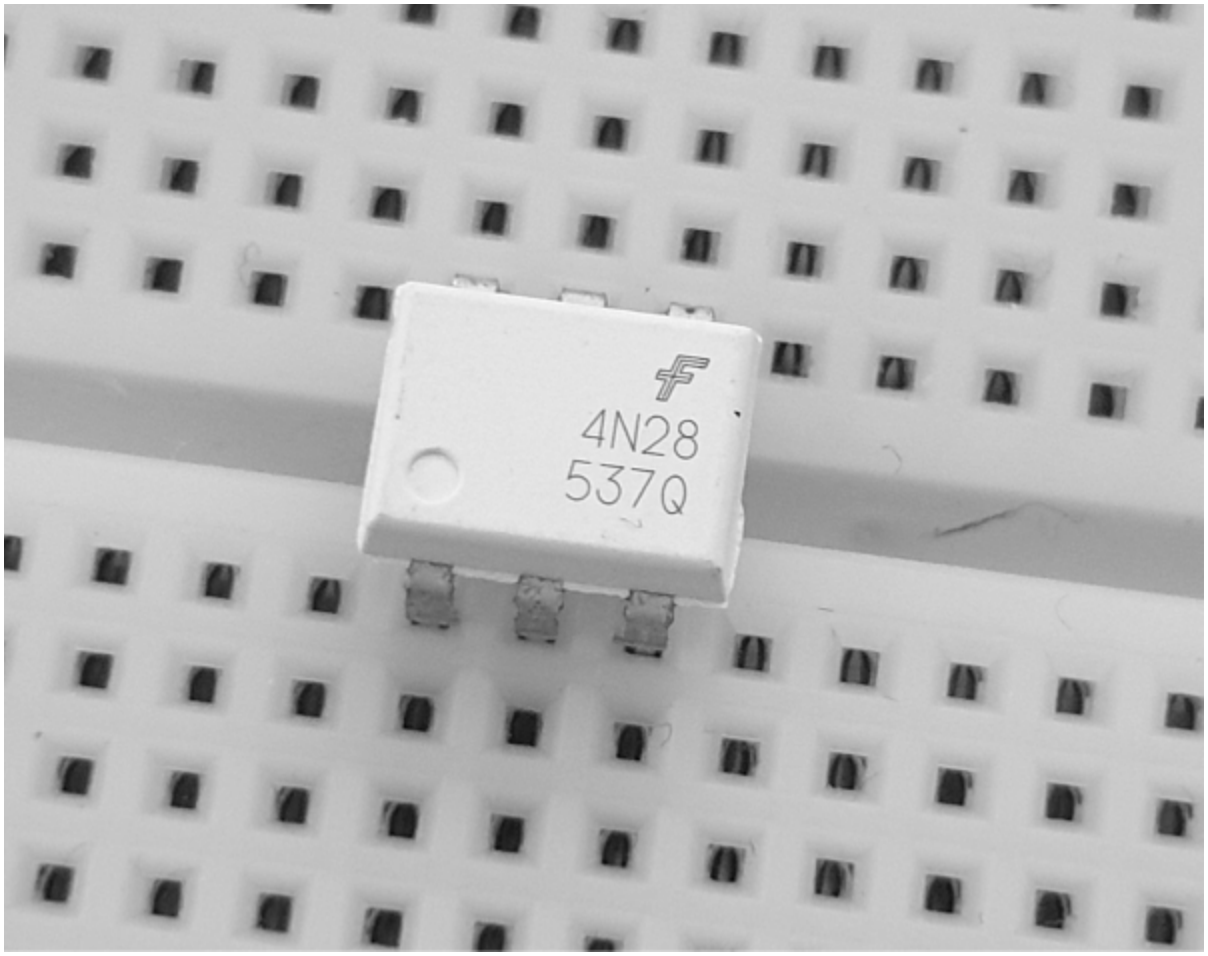


Figure 7-2: A 4N28 optocoupler inserted into a solderless breadboard

To use the optocoupler, you'll need a current-limiting resistor for the LED. You'll use 1 kΩ for this chapter. Connect the circuit to switch on or off across pins 4 and 5. If you'd like to experiment with an optocoupler, build the circuit shown in Figure 7-3.

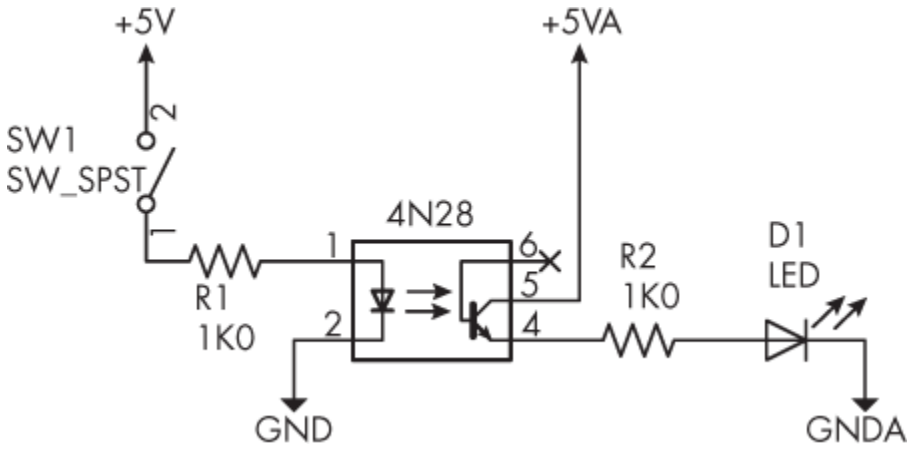


Figure 7-3: An optocoupler demonstration circuit

When SW_1 is closed, current runs through the resistor R_1, and the LED inside the optocoupler illuminates. This in turn triggers the photoresistor, which allows current to flow through from pin 5 to pin 4, turning on LED D_1. Pin 3 on the optocoupler is not electrically connected. This makes the optocoupler a switch of sorts, without any electrical or mechanical contact between the two sides of the switch.

You'll use optocouplers to interface with the transmitter of the wireless remote AC outlet in the following projects.

Remote-Control AC Outlets

Inexpensive wireless remote-control AC outlets, like the one shown in Figure 7-4, allow you to safely control mains power. In

Figure 7-4, the transmitter is on the right, and the receiver outlet is on the left.



Figure 7-4: A typical wireless remote-control AC outlet

The transmitter needs separate on and off buttons for control. Try to find one with as much space as possible between the buttons; the farther apart they are, the easier it will be to hack into the transmitter unit in Project #22, allowing you to connect it to an Arduino circuit to control the power outlet. You can find these units from the usual retailers, such as Amazon, Walmart, and so on.

WARNING

Under no circumstances are you to open or modify the receiver unit containing the mains outlet, as exposing yourself to mains wiring can be lethal.

To work through this chapter, I recommend buying two sets of controls: one to experiment with (so you don't need to worry too much about breaking it) and another to use for the project.

Hacking the Outlet Transmitter

This section explains how to hack into your wireless transmitter to control it via an Arduino. I'll show you with a series of photographs how I hacked into my remote, and you can use these guidelines to experiment with your remote to achieve the same goal. This requires some soldering and de-soldering, so be sure you're equipped with a hobbyist 20 to 50 W soldering iron, hand tools such as side-cutters, and some de-soldering wick.

Transmitters will be slightly different in shape, size, and so on, but make sure the one you purchase runs from a little 12 V A23-style battery and has separate on and off buttons, as noted earlier. Before doing any damage to the unit, insert the battery and test it to get an idea of how far between the transmitter and

receiver in which you can use the outlet. Figure 7-5 shows the battery in my transmitter.



Figure 7-5: The battery in a wireless remote-control transmitter

Check for any screws or tabs you can use to pop the transmitter enclosure apart, and open it, as shown in Figure 7-6.

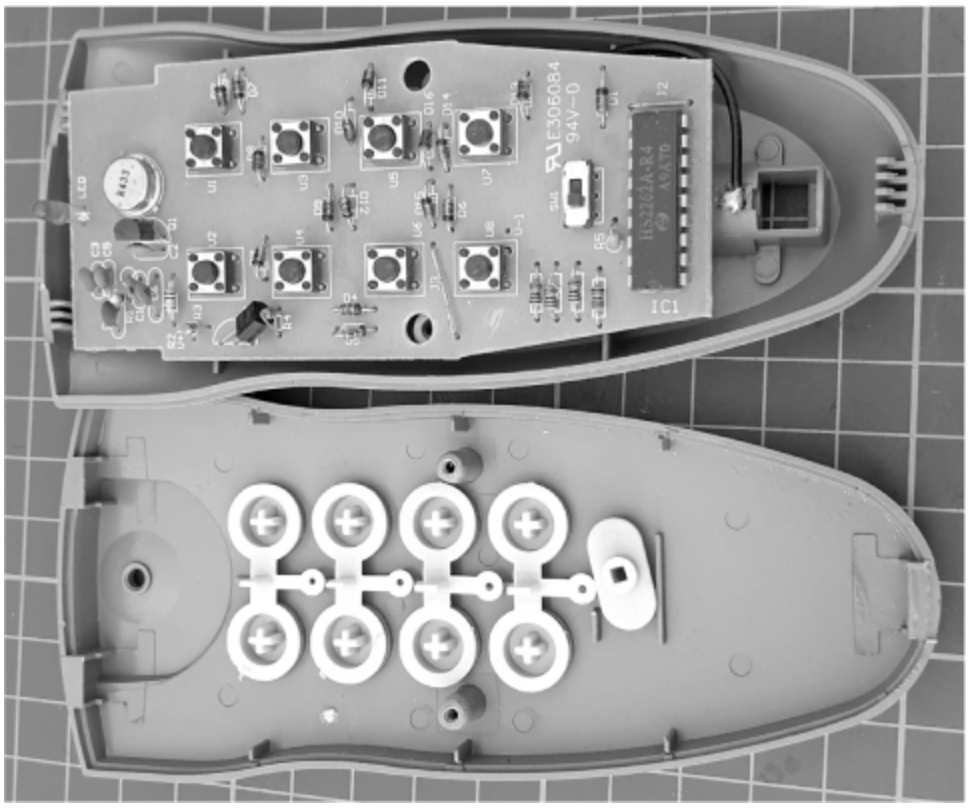


Figure 7-6: The wiring inside the wireless remote-control transmitter

This should reveal the PCB for the circuit, along with the buttons used to control the outputs. Carefully remove the PCB to check whether it looks possible to de-solder some of the buttons; if not, reassemble the unit and return it to your retailer. You should be able to de-solder buttons on any transmitters that fit the specs described earlier in this section, but it's best to make sure, for example, that they're the type that have buttons with through-hole pins.

You'll also need to connect power. If there are already wires running from the battery holder to the PCB, like those shown in

Figure 7-7, carefully cut off or de-solder them using the solder wick from the battery holder ends.

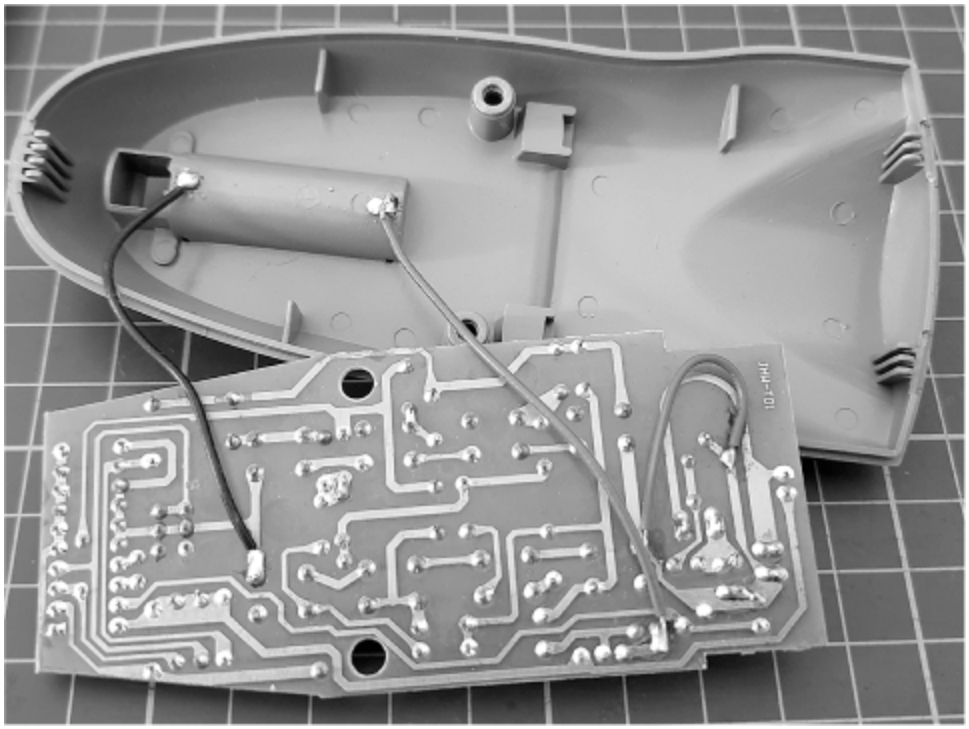


Figure 7-7: The power wires in the PCB

Next, identify the PCB pads that contain the pins of the on and off buttons for the first (or only) outlet in the system by locating the buttons and then finding their matching pins on the other side of the PCB. Viewing the buttons from the bottom of the PCB (the side with the copper tracks), match the buttons' pins to the PCB pads.

In our example, the PCB uses standard tactile buttons with four legs, two for each contact. In Figure 7-8, I've drawn lines

between the pads of each side of the tactile switch to note which PCB pads I needed to de-solder.

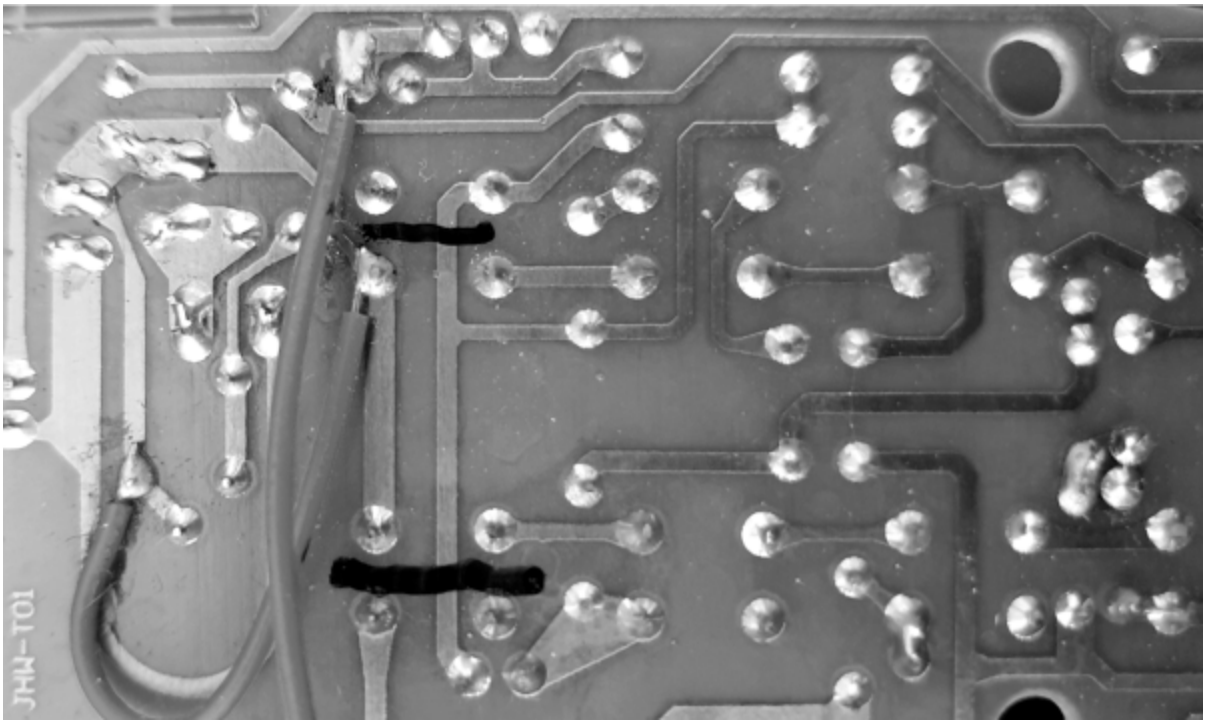


Figure 7-8: The transmitter PCB with button pins separated out with a black marker

Once you've located the PCB pads for the on and off buttons, de-solder the pins, ideally using some solder wick, as shown in Figure 7-9. Be careful not to overheat and destroy the pads.

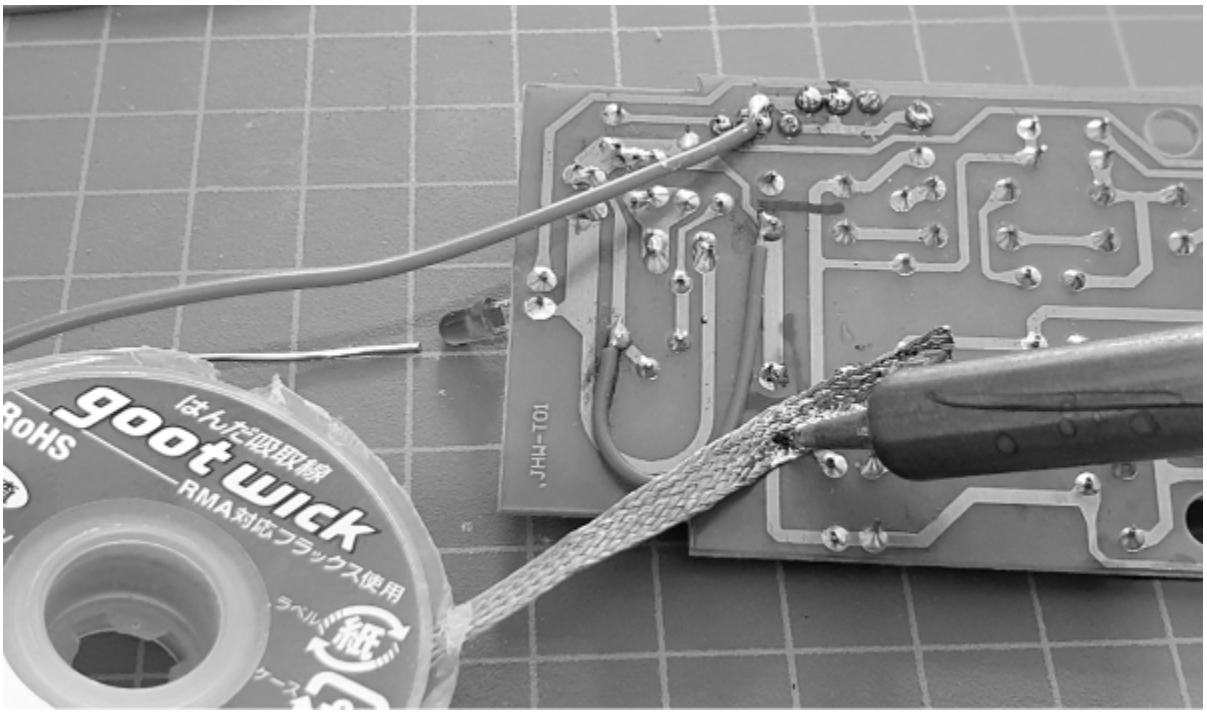


Figure 7-9: De-soldering buttons from the transmitter

Now gently pry the buttons out from the PCB. This will leave you with some blank button spaces, as shown in Figure 7-10.

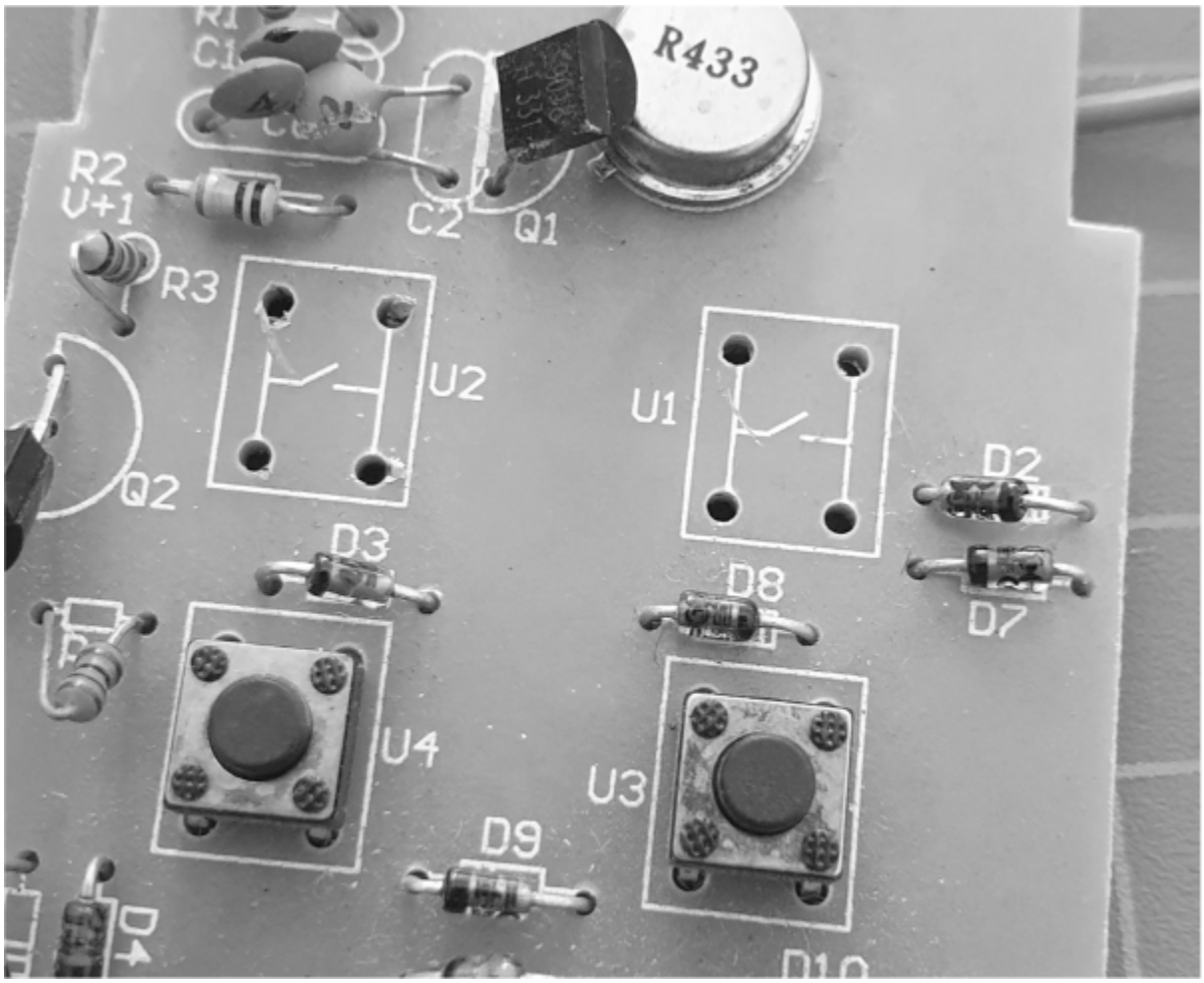


Figure 7-10: The transmitter PCB with control buttons removed

In these empty holes, solder in some jumper wires long enough to reach from the transmitter PCB to a solderless breadboard. I've chopped the ends from some preformed jumpers for easy connections to external circuitry, as shown in Figure 7-11.

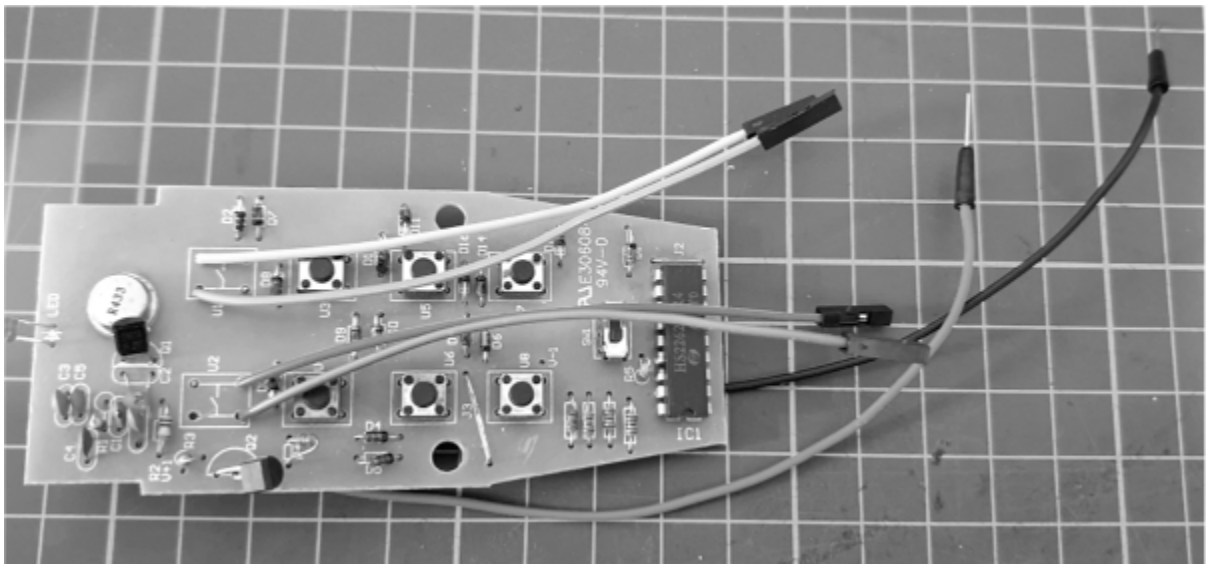


Figure 7-11: Modified wiring on the transmitter PCB

It’s time to check the operation of the transmitter PCB. Connect 12 V to the power leads and short out the on and off wire pairs in turn to check the receiver still switches on and off. If it doesn’t, the (now removed) buttons may have formed part of the circuit—that is, both button contacts for each side were used as a PCB track connection. This happened with my example buttons, so I placed a short wire across the side of the button and soldered it into the PCB, as shown in Figure 7-12.

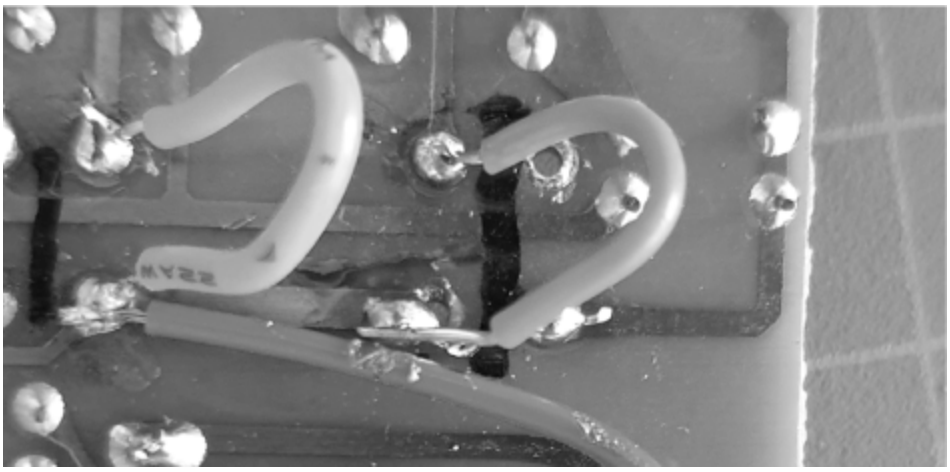


Figure 7-12: Adding links to the transmitter's button pads

Give the system another test if necessary. If it's successful, cover the link wires' joints with some hot glue so they don't come off by accident later, as shown in Figure 7-13.

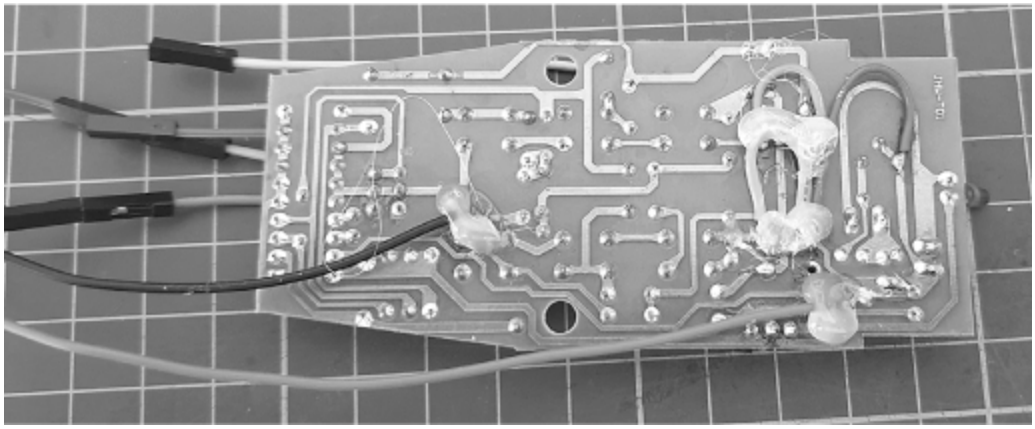


Figure 7-13: Protecting the extra soldered links with hot glue

You now have a transmitter PCB that can be controlled by an Arduino and an external optocoupler circuit. Let's put it to work.

Project #22: Controlling the Transmitter Board

This project briefly demonstrates controlling a hacked transmitter board, which will give you the framework to control the outlet in your own projects. You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- A 12 V DC, 1A power supply, wall wart, or plugpack (if you like, you can use the DC socket breakout described for Project #19 in Chapter 6)
- A hacked wireless outlet transmitter and receiver from the previous section
- Two 1 kΩ, 0.25 W, 1 percent resistors
- Two 4N28 optocouplers
- A solderless breadboard
- Various jumper wires

Assemble the circuit as shown in Figure 7-14.

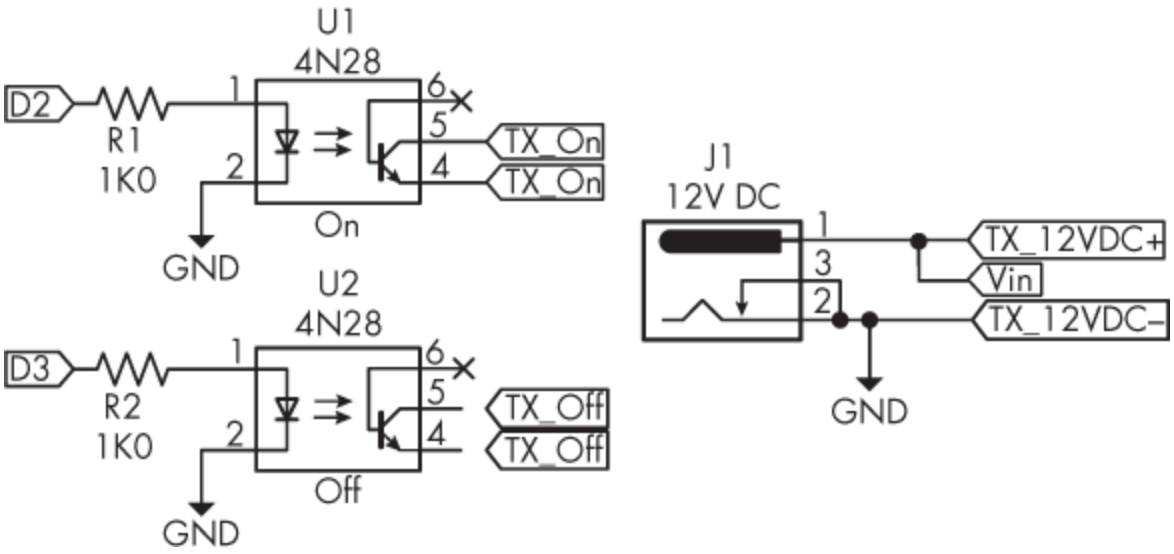


Figure 7-14: The schematic for Project #22

The TX_On and TX_Off pairs in the schematic represent the wiring to the transmitter PCB control buttons, and TX_12VDC+ and – represent the 12 V power leads to the transmitter PCB. As the 12 V DC is supplying the entire circuit, the power to the Arduino is via the Vin pin. Arduino pins D2 and D3 are used to turn the transmitter on and off.

Now upload the Project #22 sketch. After a few moments, the mains outlet should repeatedly switch on and off around every five seconds.

Let's see how this works:

```
// Project #22 - Mains outlet control with Ardui
```

```
❶ void mainsOff()
  {
      digitalWrite(3, HIGH);
      delay(1000);
      digitalWrite(3, LOW);
  }

❷ void mainsOn()
  {
      digitalWrite(2, HIGH);
      delay(1000);
      digitalWrite(2, LOW);
  }

  void setup()
  {
      delay(1000);
      pinMode(2, OUTPUT);
      pinMode(3, OUTPUT);
  }

  void loop()
  {
    ❸ mainsOn(); // AC mains on
      delay(5000);
    ❹ mainsOff(); // AC mains off
      delay(5000);
  }
```

To control the transmitter, the sketch simply closes the contacts for the on and off switches, now represented by the optocouplers, with the functions at ❶ and ❷. The delay in these functions ensures the transmitter has activated long enough to trigger the receiver unit.

The delay in `void setup()` helps to test the circuit, as otherwise the transmitter would sometimes turn on immediately when power was applied to the Arduino. You should leave this delay in your own projects to avoid unnecessary or unwanted outlet activation.

Finally, the code demonstrates the control by turning the AC on and off slowly ❸ ❹. You might connect a lamp or a small fan to the AC outlet to see the effects.

Keep this project's hardware together, as you'll expand upon it in the next project.

Project #23: Controlling the Mains Outlet with a Timer

In this project, you'll use the DS1307 or DS3231 real-time clock IC to build an AC mains outlet that turns on or off at the

required point(s) in time. This is a great way to control small irrigation pumps, night lights, alarms, or similar devices.

To build this project, you'll need the same hardware setup used in Project #22, along with a DS3231 or DS1307 RTC module or the logging shield used with Project #21 in Chapter 6. Assemble the circuit as shown in Figure 7-15.

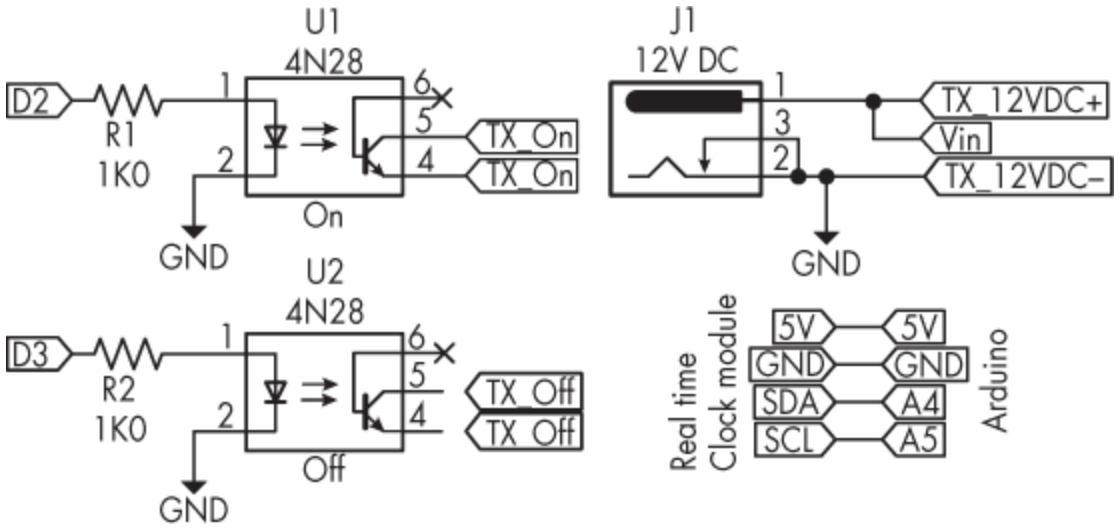


Figure 7-15: The schematic for Project #23

Now upload the Project #23 sketch (which should work for either RTC IC you've used). Don't forget to set the time with your current time data using the `setDS3231time()` function in `void setup()`. Once you've updated the time values, uncomment the function, upload the sketch, and then re-comment the function and upload the sketch again. I've used the DS3231 library as demonstrated with Project #21 in Chapter 6.

Let’s see how this works:

```
// Project #23 - Timer-controlled mains outlet
#include <Wire.h> ❶
#include <DS3231.h>

DS3231 RTC;
bool century = false; ❷
bool h12Flag = false;
bool pmFlag = false;
byte year = 22;
byte month = 9;
byte date = 19;
byte hour = 21;
byte minute = 53;
byte second = 0;

void setTimeData()
{
    RTC.setYear(year);
    RTC.setMonth(month);
    RTC.setDate(date);
    RTC.setHour(hour);
    RTC.setMinute(minute);
    RTC.setSecond(second);
    RTC.setClockMode(false); // Set clock to 24 h
}
```

```
void mainsOff()
{
    digitalWrite(3, HIGH);
    delay(1000);
    digitalWrite(3, LOW);
    digitalWrite(13, LOW);
}

void mainsOn()
{
    digitalWrite(2, HIGH);
    delay(1000);
    digitalWrite(2, LOW);
    digitalWrite(13, HIGH);
}

void setup()
{
    mainsOff();
    //setTimeData(); ❸ // Set time and date
    Wire.begin();
    pinMode(2, OUTPUT);
    pinMode(3, OUTPUT);
    pinMode(13, OUTPUT);
    digitalWrite(13, LOW);
}

void turnOn(int onHour, int onMinute)
```

```
{
    if ((RTC.getHour(h12Flag, pmFlag) == hour) &
    {
        mainsOn();
        delay(59100);
    }
}

void turnOff(int offHour, int offMinute)
{
    if ((RTC.getHour(h12Flag, pmFlag) == hour) &
    {
        mainsOff();
        delay(59100);
    }
}

void loop()
{
    turnOn(17, 02);
    turnOff(17, 03);
    turnOn(17, 04);
    turnOff(17, 05);
    turnOn(17, 06);
    turnOff(17, 07);
}
```

The sketch first includes the I^2C and RTC libraries and then creates an instance of the real-time clock to reference ❶. It then declares the required variables to hold the time and data information ❷. These include three Boolean variables, which are left as false by default, since the sketch uses 24-hour time.

The custom `setTimeData()` function contains the functions to set all the time and date parameters. The variables declared earlier are placed in each matching function. This function needs to be called only once ❸, as you initially set the time and date and then comment out the function before re-uploading the sketch; otherwise, the clock will reset to the value of the variables.

The `mainsOff()` and `mainsOn()` functions also use the Arduino's LED on D13 to indicate the system's on or off status. For outlet control at a certain time, the two `turnOn()` and `turnoff()` functions each accept an hour and minute and compare them against the current time.

If there's a match, the outlet is turned on or off, respectively, as shown with the examples in `void loop()`. You can include as many on or off functions as required; the sketch will continually loop about the functions checking to turn on or off. There is a long delay in both `turnOn()` and `turnOff()` to stop

multiple activations of the transmitter during the times when there is a match of time for on or off.

For a challenge, you can modify the sketch to allow the user to select the date or day of the week along with the time to turn on or off.

Project #24: Controlling the Mains Outlet with SMS

This project takes wireless control to the next level, using the Project #22 hardware in conjunction with a 3G cellular shield to create an AC mains outlet controllable by text message (wherever there's cell phone service). Doing so is surprisingly simple.

NOTE

You may have built similar projects to this in Chapter 22 of Arduino Workshop*, 2nd edition.*

For this project to work, you'll need access to a cellular network that operates at UMTS (3G) 850 MHz, 900 MHz, 1900 MHz, or 2100 MHz and allows the use of devices not supplied by the network provider. This information should be available from your cellular provider. You will also need a SIM card for your

network—prepaid or otherwise—for this shield, in addition to the one already in your phone. Also make sure the requirement to enter a PIN to use the SIM card is turned off. (You should be able to do this by inserting the SIM card into a regular cell phone and changing the setting in the Security menu.)

This project uses the SIM5320-type 3G GSM shield and antenna shown in Figure 7-16. 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 latter uses the UMTS/HSDPA 900/2100MHz frequency bands (mainly for European users), and the former uses the UMTS/HSDPA 850/1900MHz frequency band (mainly for US-based users and Australians using the Telstra network).



Figure 7-16: A 3G shield with an antenna

Since these shields require 12 V external power, begin by connecting the power supply you used in earlier projects in this chapter via the Vin and GND pins. To configure pins D2 and D3, which the shield uses with SoftwareSerial to communicate with the Arduino, connect the jumpers over the RX 3 and TX 2 pins, as shown in Figure 7-17.

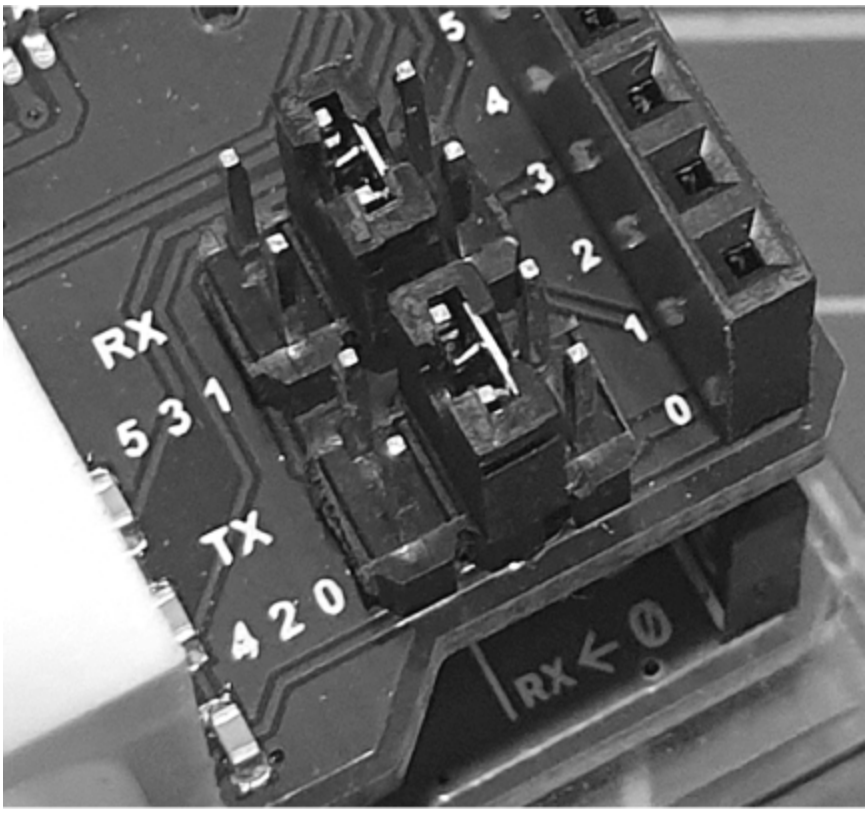


Figure 7-17: Shield serial configuration jumpers

Next, turn the shield over and insert your carrier SIM card into the holder, as shown in Figure 7-18.

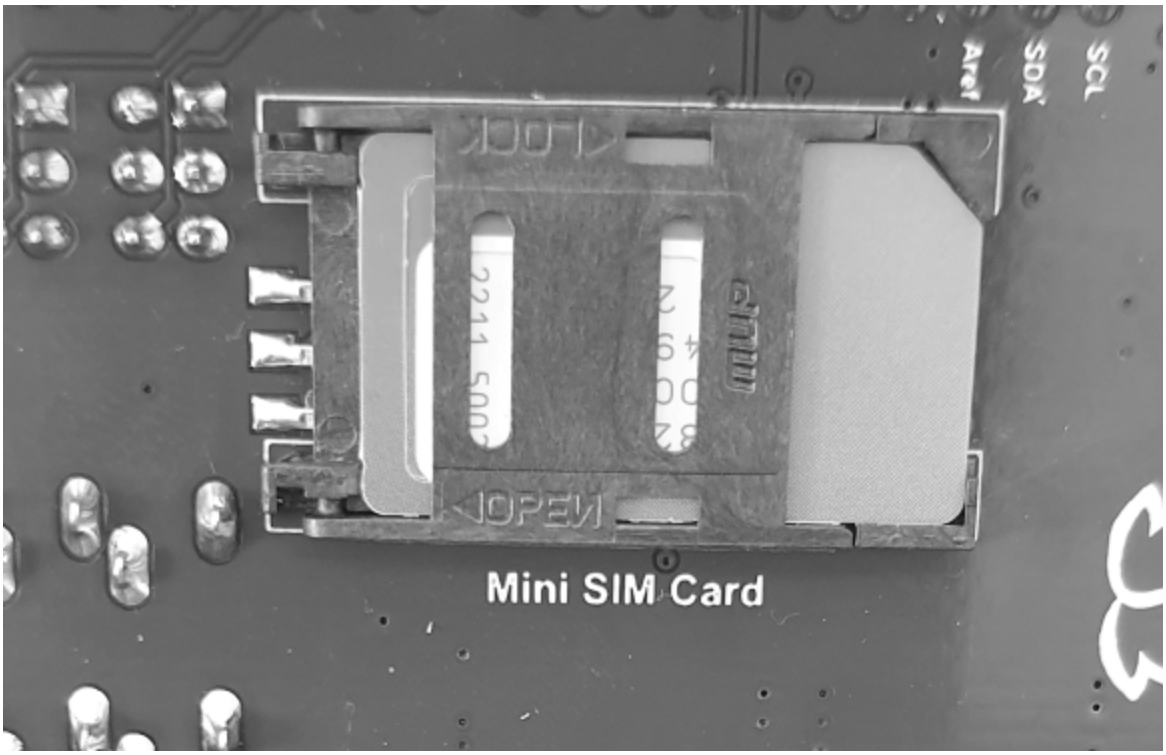


Figure 7-18: A SIM card in its holder

Gently insert the 3G shield into the Arduino. Connect the external power and the USB cable between the Arduino and the PC, then screw in the external antenna. Finally, turn the SIM module on using the power button on the top left of the shield, as shown in Figure 7-19. Press the button for two seconds, then let go.

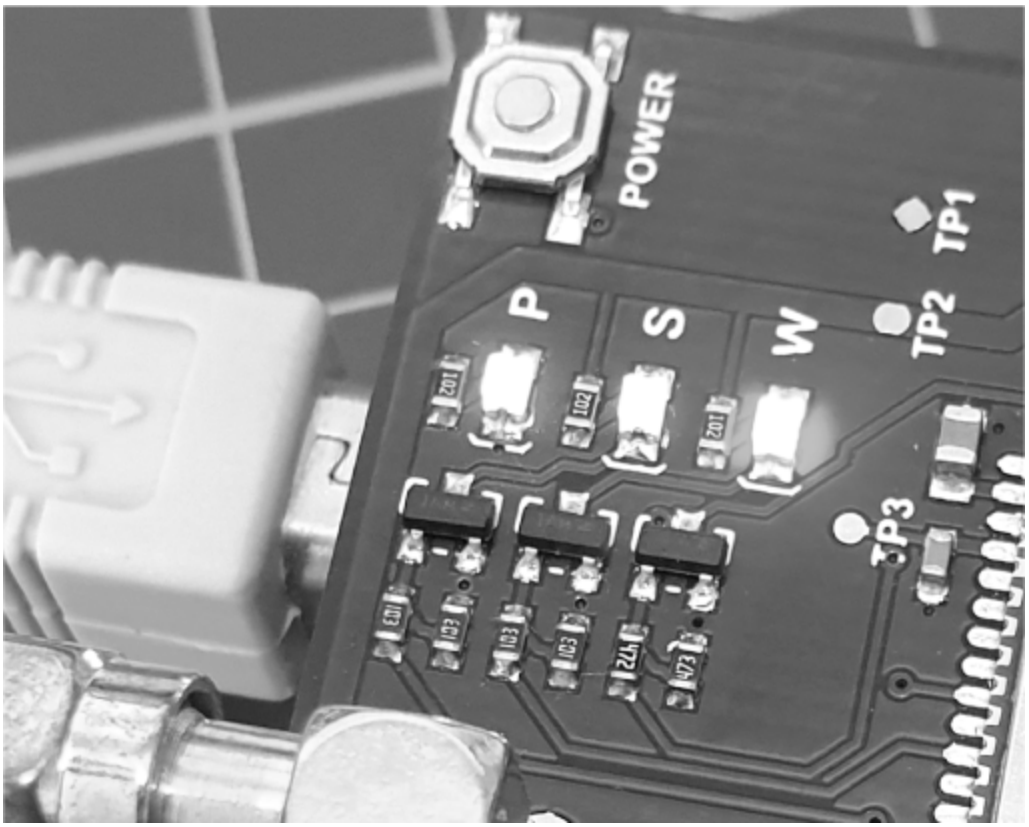


Figure 7-19: The 3G shield power button and status LEDs, which are on for power, status, and network activity

The P (for power) and S (for status) LEDs should come on. The blue W (for network activity) LED should start blinking once the 3G shield has registered with the cellular network, meaning you're ready to use the shield and all is well.

To build this project, you'll need the 3G shield you've just put together, along with the hardware setup you built in Project #22. Assemble the circuit as shown in Figure 7-20. Note that the optocoupler inputs have changed from the pins used in Project

#22 to Arduino pins D4 and D5, as we use D2 and D3 for the 3G shield's serial communications.

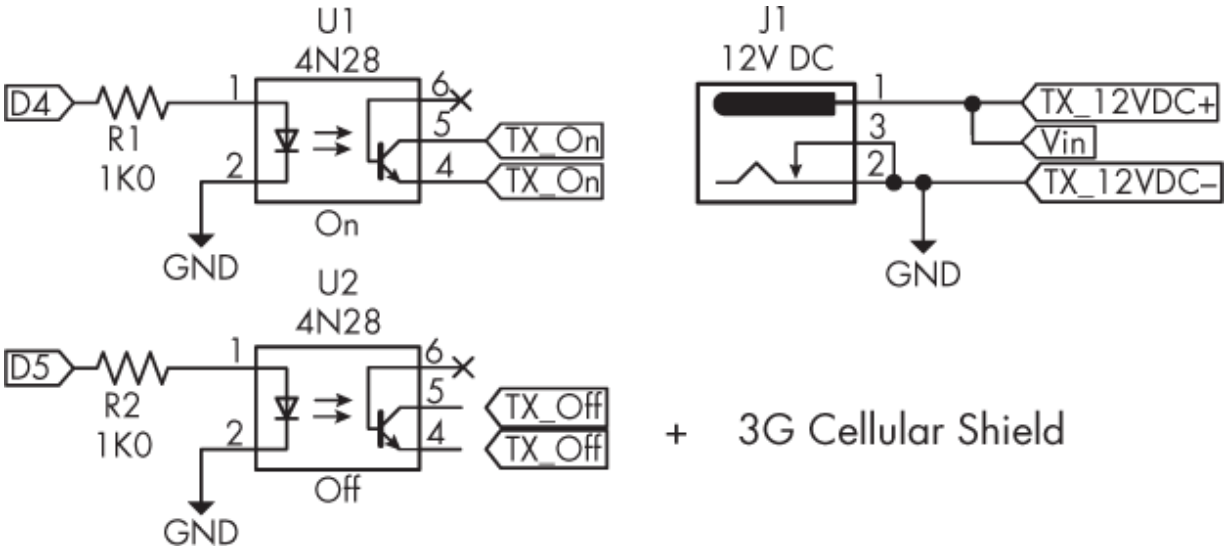


Figure 7-20: The schematic for Project #24

Enter and upload the following sketch. Once the shield turns on and the blue LED starts blinking, send **`#1`** as an SMS to the SIM card's cellular number. The mains outlet should turn on; send **`#0`** to turn it off again. I hope this inspires a brief sense of wonder—these moments make learning and building projects quite enjoyable.

Let's see how this works:

```
// Project #24 - Setting up an SMS remote contro

#include <SoftwareSerial.h>
```

```
❶ SoftwareSerial cell(2, 3);

❷ void mainsOff()
  {
      digitalWrite(5, HIGH);
      delay(1000);
      digitalWrite(5, LOW);
      digitalWrite(13, LOW);
  }

❸ void mainsOn()
  {
      digitalWrite(4, HIGH);
      delay(1000);
      digitalWrite(4, LOW);
      digitalWrite(13, HIGH);
  }

  void setup()
  {
      pinMode(5, OUTPUT);
      pinMode(4, OUTPUT);
      pinMode(13, OUTPUT);
    ❹ pinMode(8, OUTPUT);
      mainsOff();
    ❺ digitalWrite(8, HIGH);
      delay(2000);
      digitalWrite(8, LOW);
```

```
    cell.begin(4800);
    delay(30000);
 ❻ cell.println("AT+CMGF=1");
    delay(200);
 ❼ cell.println("AT+CNMI=3,3,0,0");
    delay(200);
}

void loop()
{
    char inchar;
    if (cell.available() > 0)
    {
        inchar = cell.read();
     ❽ if (inchar == '#')
        {
            delay(10);
            inchar = cell.read();
            if (inchar == '0')
            {
              ❾ mainsOff();
            }
            else if (inchar == '1')
            {
              ❿ mainsOn();
            }
            delay(10);
            cell.println("AT+CMGD=1,4"); // Dele
```

```
            }
        }
    }
```

The sketch initializes the software serial port for communication with the 3G shield ❶ and then declares the variable used to interrogate incoming data from the shield. The transmitter control functions appear at ❷ and ❸, and the code configures the digital pins in `void setup()` ❹.

Starting at ❺, the sketch turns on and configures the 3G shield for use, using the AT command `AT+CMGF=1` to convert incoming SMS messages to text and send them to the software serial port ❻. Whenever the 3G shield receives a text message, the details are sent to the Arduino via the software serial port, one character at a time ❼. The sketch tests each incoming character to see if it's a `#` ❽; if so, it checks for a `0` or `1` and then turns the transmitter off ❾ or on ❿ respectively.

The shield's power button is connected to digital pin 8, so you can also control the power from the project sketch, instead of manually turning the button on or off.

Moving On

In this chapter, you learned to safely control mains AC power outlets using various forms of automation and remote control. You can use the knowledge gained to control the AC mains outlet using various sensors, switches, or other inputs or code in your own projects. As long as your input device can control an LED, it can control the AC mains outlet.

In the next chapter, you'll use high-power shift registers to build even more interesting control applications.

8

CONTROLLING HIGHER-POWERED DEVICES



Many Arduino users rely on the 74HC595 shift register for their projects, as it's popular and easy to use. However, the 74HC595 can handle only a relatively low amount of current, especially when all pins are active: while you can draw 20 mA of current from one output continuously, the entire IC is rated for a draw of only 70 mA through the V_{CC} or GND pin.

If you need 20 mA per output to drive, for instance, eight separate LEDs, you could use only three of the eight pins under recommended operating conditions. While it may be possible to exceed the manufacturer's direction, good electronic designs

should consider safety and reliability. A shift register designed to handle higher currents is a better choice.

This chapter shows you how to control higher-powered devices with your Arduino with the TPIC6B595 shift register IC. You'll learn to:

- Experiment with a binary number display
- Use multiple TPIC6B595s to control more than eight high-powered outputs
- Use bright Piranha-style LEDs that are more powerful than regular LEDs

You'll also build a PC-controlled eight-relay board and control giant seven-segment numeric displays.

The TPIC6B595

The TPIC6B595 is controlled in the same way as the 74HC595 but offers up to 150 mA per output and a 500 mA IC total current draw—just over 60 mA per pin when all pins are used. It can also switch voltages up to 50 V DC. This allows for control of eight higher-current items such as powerful LEDs, relay coils, or mechanical switchgear.

The TPIC6B595 is a *latching shift register*, meaning that as long as power is connected, it will maintain the output status. For example, if you upload a new sketch, the outputs are not affected. If you've powered your circuit from a power supply and not the Arduino, you can reset the Arduino without altering the outputs.

Figure 8-1 shows a TPIC6B595 in a dual in-line package, through-hole format in a solderless breadboard.

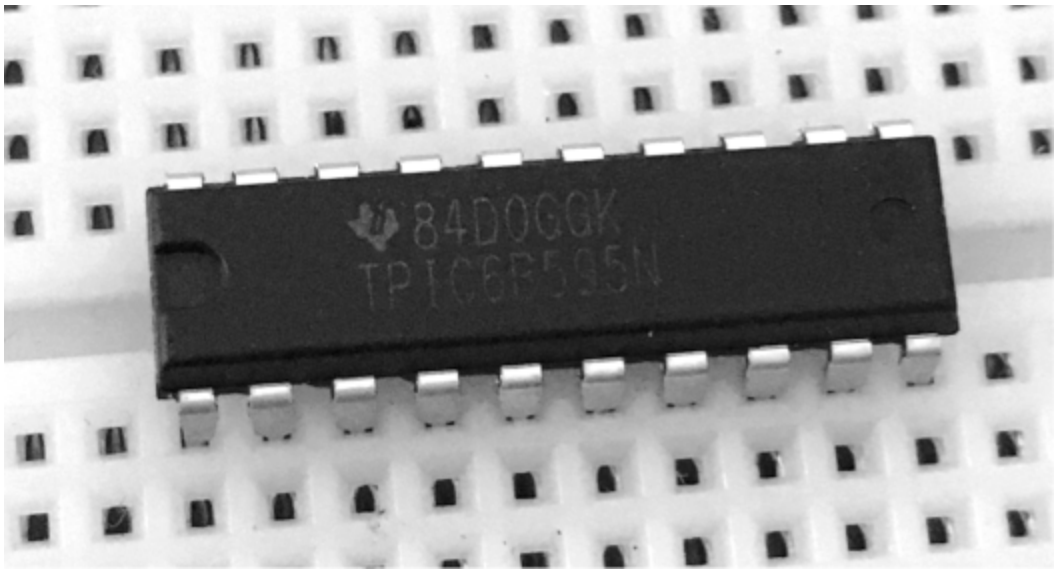


Figure 8-1: A TPIC6B595 shift register

Figure 8-2 shows the TPIC6B595's schematic symbol.

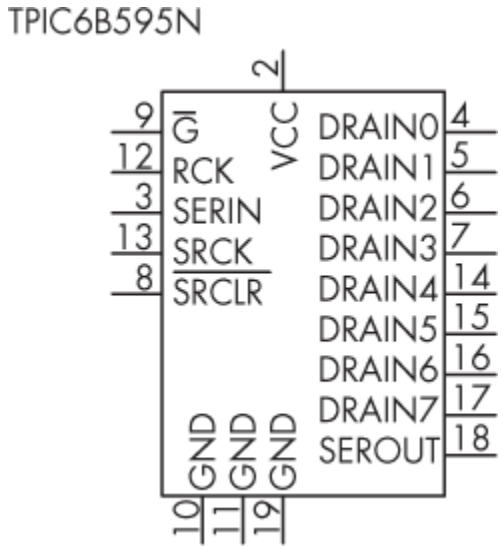


Figure 8-2: The schematic symbol for the TPIC6B595

The eight output pins in the schematic are labeled DRAIN*x*, since the TPIC6B595 has low-side outputs. Like the 2N7000 N-MOSFETs used in previous chapters, each of these outputs controls current entering the pin (as opposed to the 74HC595's high-side outputs, for example, where currents flow out from the eight control pins). This means the devices to be controlled are connected between the power supply and the control pins on the TPIC6B595, which switches the currents' ability to flow from the device to GND.

Consider the schematic in Figure 8-3. Current flows from the 5 V source, through the resistor and LED, and into the TPIC6B595's output pin. When that pin is activated, the current continues to GND, completing the circuit.

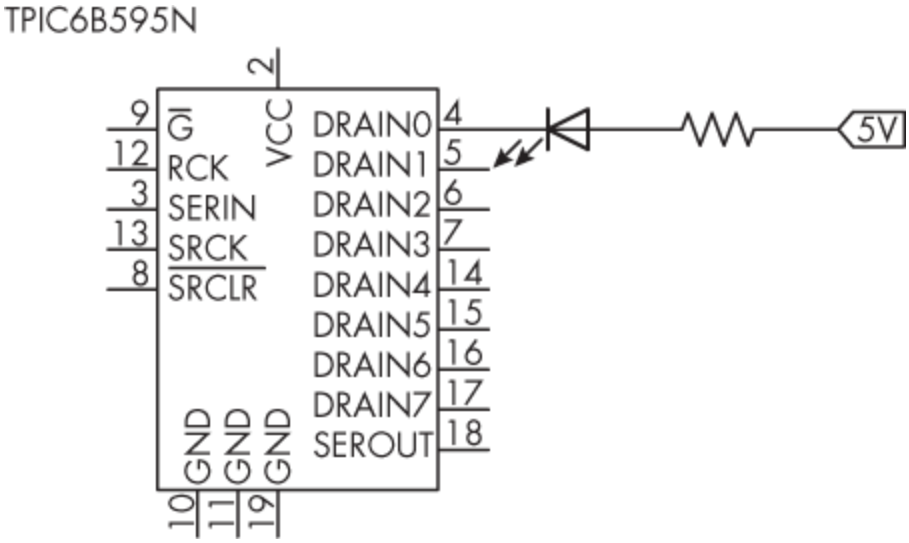


Figure 8-3: An example of LED control with TPIC6B595

Given the TPIC6B595's method of controlling current, the voltage of the devices it controls can be up to 50 V, while the shift register is still operating on 5 V. Conveniently, that means you can control 12 V devices or higher without worrying about level conversion back to the Arduino.

Let's put the TPIC6B595 to the test with a simple project that demonstrates shift register operation.

Project #25: Creating a TPIC6B595 Binary Number Display

This project demonstrates the use of TPIC6B595 outputs while refreshing your knowledge of binary numbers and how they

relate to shift register output control. You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- A solderless breadboard
- Various jumper wires
- A TPIC6B595 shift register IC
- One 0.1 µF capacitor
- Eight LEDs
- Eight 1 kΩ, 0.25 W, 1 percent resistors

Assemble the circuit as shown in Figure 8-4.

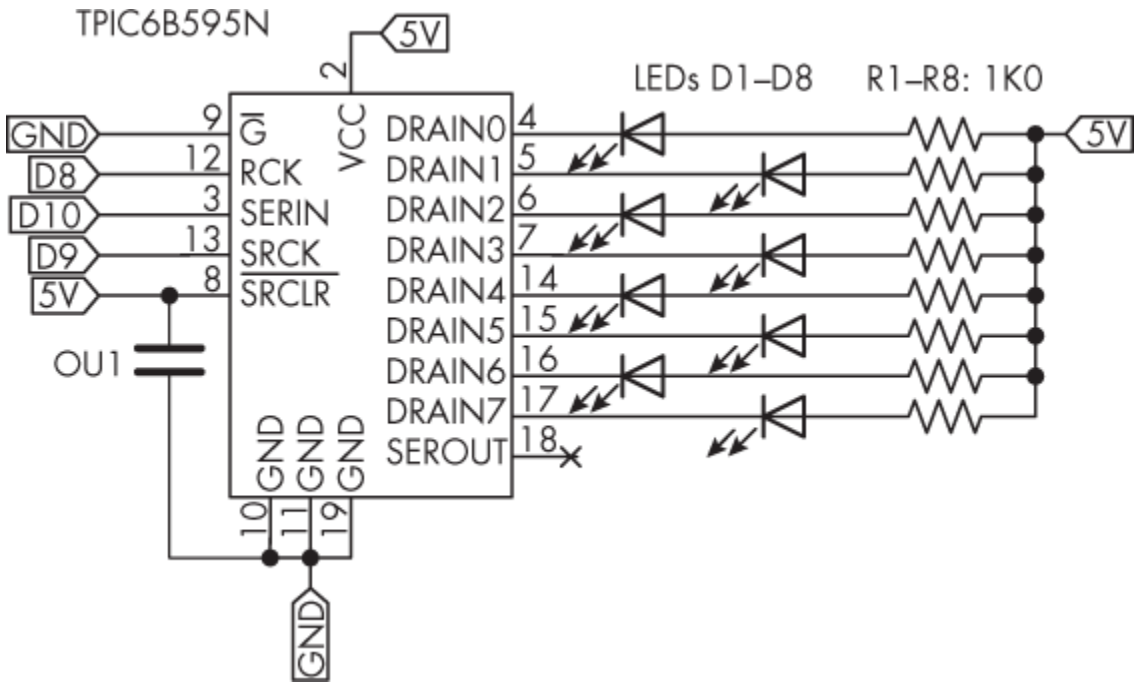


Figure 8-4: The schematic for Project #25

Enter and upload the Project #25 sketch. Open the Serial Monitor in the IDE and enter a number between 0 and 255, inclusive, then press CTRL-ENTER. The Arduino should respond by displaying the number in binary using the LEDs, as well as in the Serial Monitor, as shown in Figure 8-5. LED 1 will be the least significant bit of the number, which represents 1, and LED 8 will be the most significant bit, which represents 255.

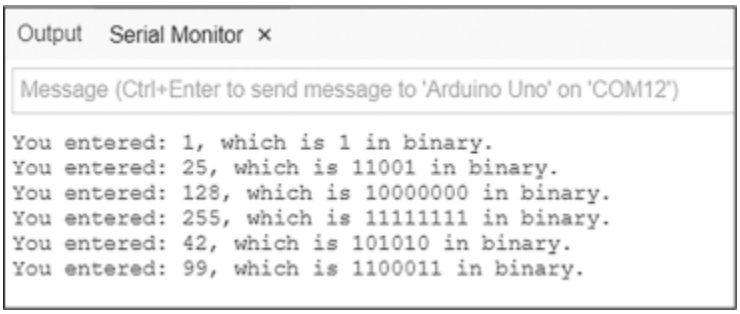


Figure 8-5: Example output from Project #25

Let's see how this works:

```
  // Project #25 - TPIC6B595 binary number display

❶ #define latch 8 // Latch RCK pin
  #define clock 9 // Clock SRCK pin
  #define data 10 // Data SERIN pin

  void displayBinary(int displayNumber)
  {
      digitalWrite(latch, LOW);
```

```
    shiftOut(data, clock, MSBFIRST, displayNumbe
    digitalWrite(latch, HIGH);
}

void setup()
{
 ❷ Serial.begin(9600);
    pinMode(latch, OUTPUT);
    pinMode(clock, OUTPUT);
    pinMode(data, OUTPUT);
}

voi d  loop()
{
    long number = 0;
    long entry = 0;
 ❸ Serial.flush();
    while (Serial.available() == 0) {}
 ❹ while (Serial.available() > 0)
    {
        number = number * 10;
        entry = Serial.read() - '0';
        number = number + entry;
        delay(5);
    }
    displayBinary(number);
    Serial.print("You entered: ");
    Serial.print(number);
```

```
      Serial.print(", which is ");
      Serial.print(number, BIN);
      Serial.println(" in binary.");
  }
```

The sketch begins by defining the Arduino digital pins used for connecting to the shift register's Latch, Clock, and Data pins, respectively ❶. The custom `displayBinary()` function accepts an integer and sends it to the shift register for output control, using the same method as the 74HC595 shift register mentioned earlier. To send the bits representing the number to the shift register in binary and to activate the pins in the shift register to control the LEDs that will match the number to be displayed in binary, the function uses `MSBFIRST` (most significant bit first).

You can turn shift register outputs on and off with the 8 bits of the number sent to the shift register, as each bit matches an output and a status (1 for HIGH, 0 for LOW). You can also change `MSBFIRST` to `LSBFIRST`, standing for least significant bit first, to see the number "reversed" in binary.

The sketch initializes the Serial Monitor and digital output pins ❷ and then flushes the serial input and waits for the user to enter a number into the Serial Monitor ❸. It then combines the digits of the number entered in the Serial Monitor to make the

final number to display ❹. The custom `displayBinary()` function sends that number to the shift register and the Serial Monitor.

You'll put this framework for shift register control to use in the next project.

Project #26: Building a PC-Controlled Relay Board

In this project, you'll build a relay control board with eight single-pole, double-throw (SPDT) relays that you control via a PC or another device with an Arduino-compatible UART. In the future, you might use the technique introduced in this project to control low-voltage lighting or electric door locks, turn speakers on and off, and more.

Each relay in this project is capable of controlling up to 30 V DC at 2 A of current, if you're using the project PCB. If you're using the solderless breadboard, the relays should be used to switch only up to 100 mA or so.

While it's possible to build this project using a solderless breadboard, this requires soldering jumper wires to the relay pins for remotely wiring the relays back to the circuit, as shown in Figure 8-6, as the relay pins don't sit in the breadboard very well. I strongly recommend you use the PCB instead.

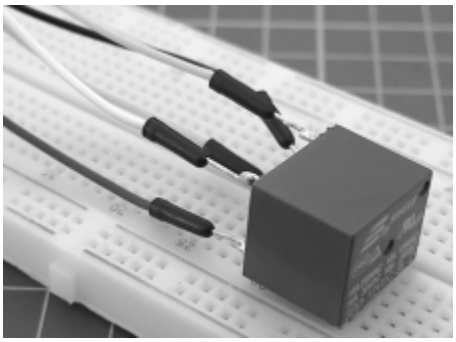


Figure 8-6: A relay with remote wiring for breadboard use

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- Assorted jumper wires
- A 12 V power supply or wall wart with DC plug
- A TPIC6B595 shift register
- A 20-pin IC socket
- One 0.1 µF capacitor
- Eight LEDs
- Eight 1 kΩ, 0.25 W, 1 percent resistors
- Eight 1N4001 power diodes
- Eight SRD-12VDC-SL-C SPDT relays
- A solderless breadboard or Project #26 PCB

If you're using the PCB, you will also need the following:

- Ten three-way 5.08 mm terminal blocks
- A 20-pin IC socket

- A PCB mount DC socket

Assemble the circuit as shown in Figure 8-7.

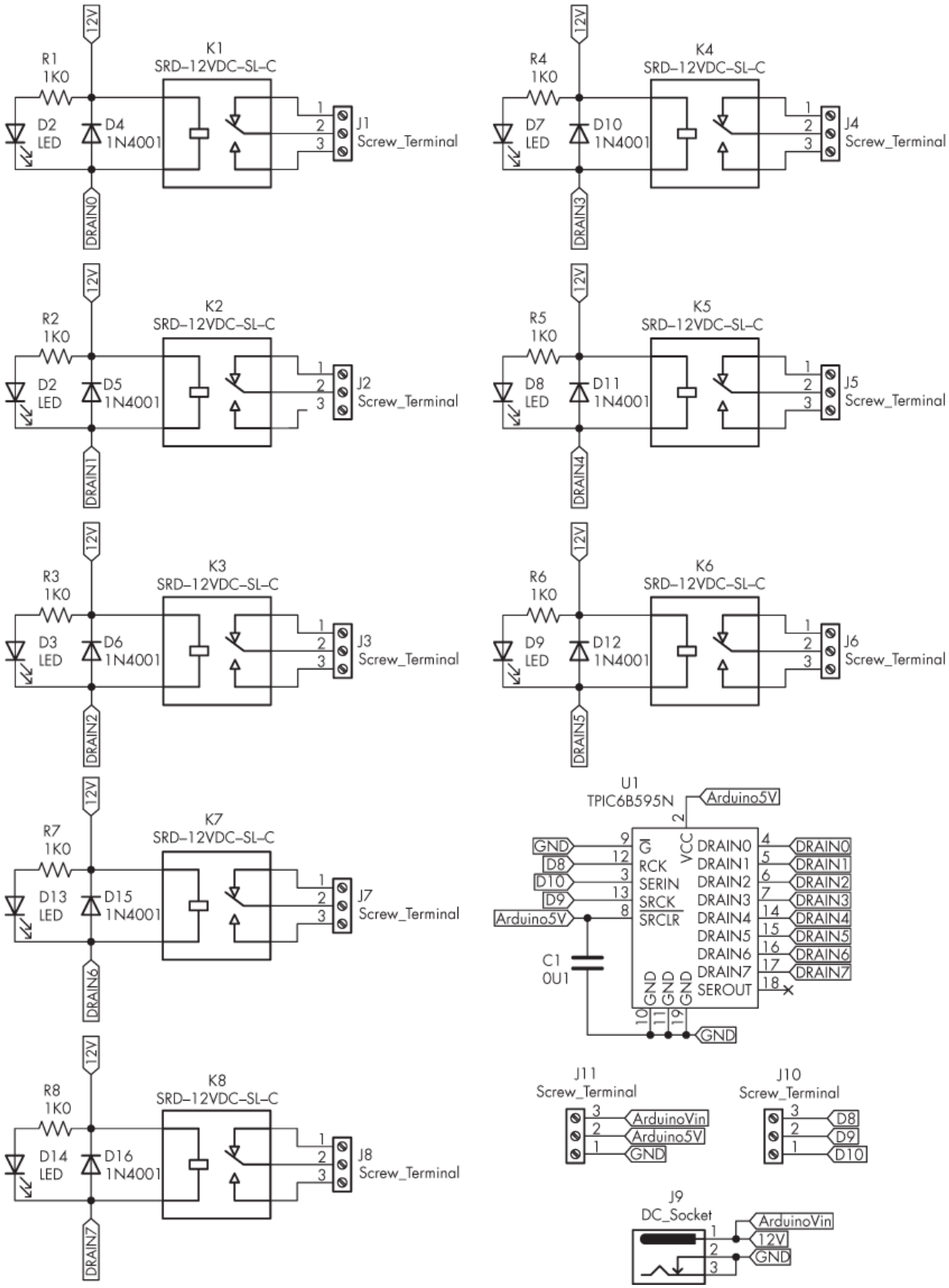

12V
R1
1K0
K1
SRD–12VDC–SL–C
D2
LED
D4
1N4001
J1
Screw_Terminal
DRAIN0
R2
1K0
K2
SRD–12VDC–SL–C
D2
LED
D5
1N4001
J2
Screw_Terminal
DRAIN1
R3
1K0
K3
SRD–12VDC–SL–C
D3
LED
D6
1N4001
J3
Screw_Terminal
DRAIN2
R7
1K0
K7
SRD–12VDC–SL–C
D13
LED
D15
1N4001
J7
Screw_Terminal
DRAIN6
R8
1K0
K8
SRD–12VDC–SL–C
D14
LED
D16
1N4001
J8
Screw_Terminal
DRAIN7
R4
1K0
K4
SRD–12VDC–SL–C
D7
LED
D10
1N4001
J4
Screw_Terminal
DRAIN3
R5
1K0
K5
SRD–12VDC–SL–C
D8
LED
D11
1N4001
J5
Screw_Terminal
DRAIN4
R6
1K0
K6
SRD–12VDC–SL–C
D9
LED
D12
1N4001
J6
Screw_Terminal
DRAIN5
U1
TPIC6B595N
Arduino5V
GND
D8
D10
D9
Arduino5V
C1
0U1
GND
RCK
SERIN
SRCK
SRCLR
VCC
DRAIN0
DRAIN1
DRAIN2
DRAIN3
DRAIN4
DRAIN5
DRAIN6
DRAIN7
SEROUT
GND
J11
Screw_Terminal
ArduinoVin
Arduino5V
GND
J10
Screw_Terminal
D8
D9
D10
J9
DC_Socket
ArduinoVin
12V
GND

Figure 8-7: The schematic for Project #26

If you're using the PCB, the layout is simple, as shown in Figure 8-8.

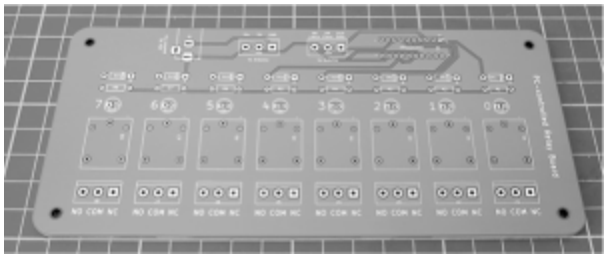


Figure 8-8: The PCB for Project #26

Start by inserting the resistors; then insert the diodes, the LEDs, the IC and DC sockets, the terminal blocks, and end with the relays. Be sure to insert the shift register correctly—pin 1 is marked on the PCB. Connect to the Arduino via the two terminal blocks next to the shift register, as shown in the schematic and Figure 8-9. The Arduino is powered by the relay board's 12 V supply and returns 5 V to the board to power the shift register.

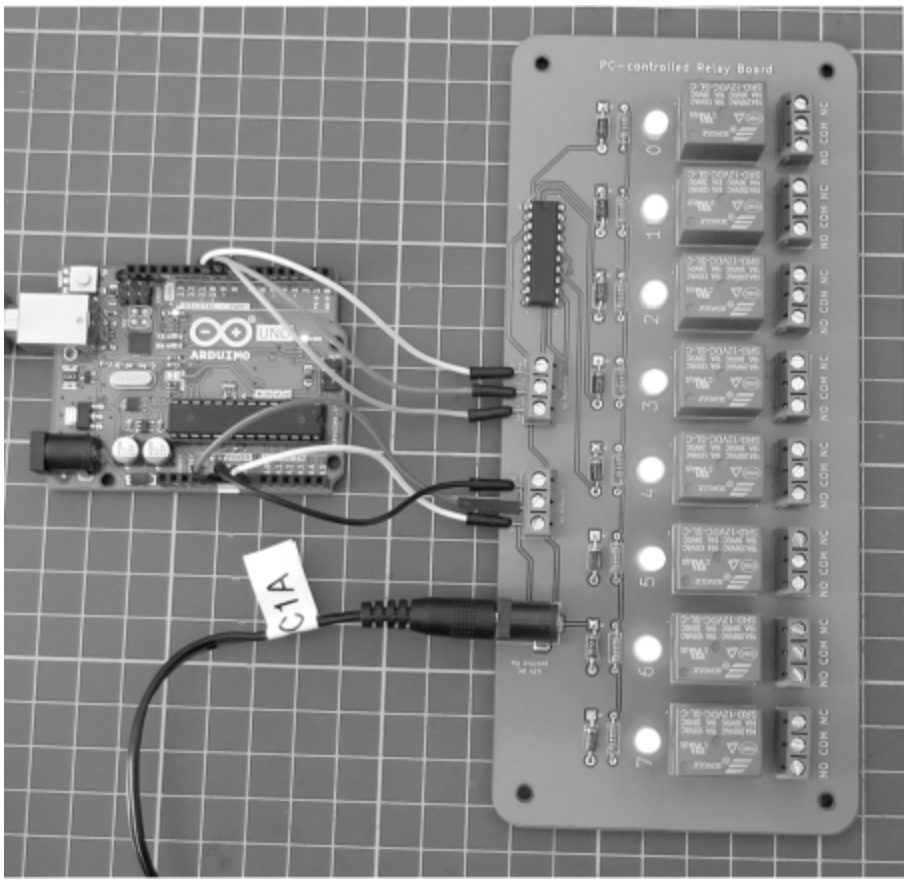


Figure 8-9: The completed hardware for Project #26

Once you've set up the hardware, enter and upload the Project #26 sketch, which controls the relays with various commands. Entering the numbers 0 through 7 using the Serial Monitor or terminal software turns on relays 0 through 7, respectively; entering 8 through F turns off relays 0 to 7; G turns all relays on; and H turns all relays off. Enter **?** to check which relays are on and off based on the commands being entered. The result is returned as a binary number matching the relay order. If the user enters an unrecognized character, the Arduino returns a list of valid commands.

For example, Figure 8-10 shows the output for the commands `G`, then `Q` (which caused the incorrect command message to display), then `H`, and then `0`, `3`, `5`, `7`, and `?`.

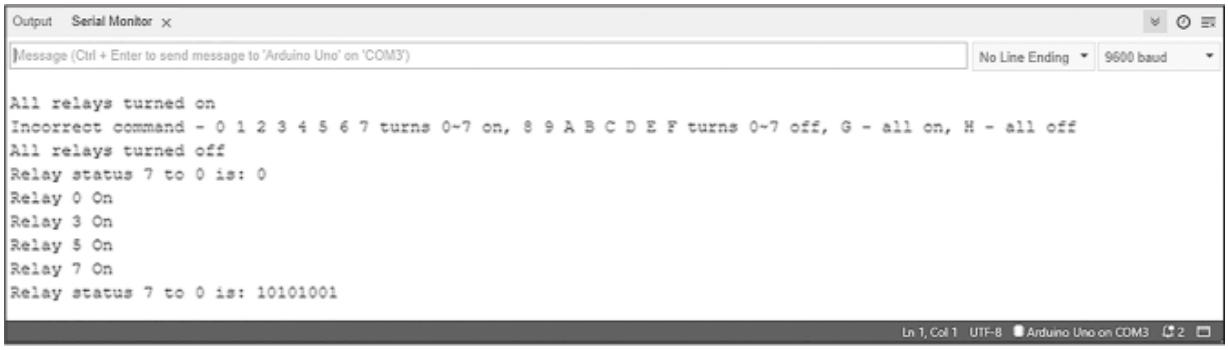


Figure 8-10: An example of operations in the Serial Monitor

If you're sharing the relay board and Arduino with others, they don't need to run the Arduino IDE for control; instead they can use any terminal software on their PC, Mac, or other computer that supports USB serial. For example, the same operation is possible with Roger Meier's CoolTerm application, available from *http://freeware.the-meiers.org*, as shown in Figure 8-11.

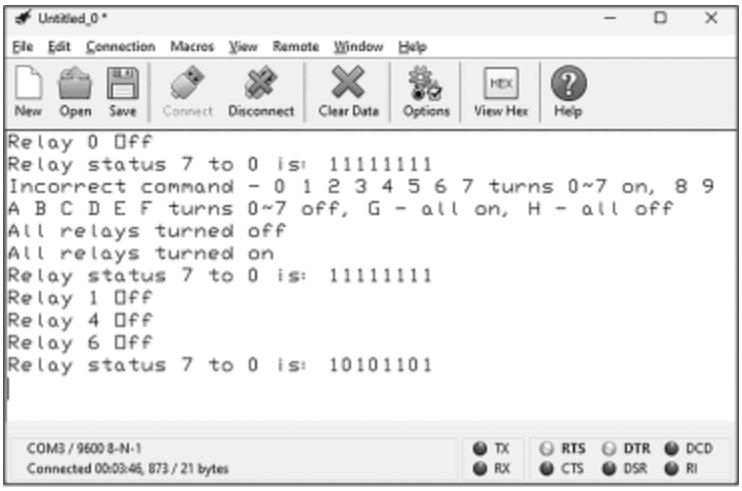


Figure 8-11: Controlling the relays using the CoolTerm application

The control Arduino expects only single characters from the host computer (or other UART), so you can write software for your computer in many environments to control the relays. Search your preferred environment's resources for "plaintext serial over USB" or similar to learn more.

Let's see how this works:

```
// Project #26 - PC-controlled relay board

#define latch 8 // Latch RCK pin
#define clock 9 // Clock SRCK pin
#define data 10 // Data SERIN pin

int relayStatus;

void showStatus()
```

```
{
    Serial.print("Relay status 7 to 0 is: ");
    Serial.println(relayStatus,BIN);
}

void waveHello()
{
    int d = 250;
    Serial.println("Hello!");
    allOff();
    for (int a = 0; a < 8; a++)
    {
        relayOn(a);
        delay(d);
        relayOff(a);
    }
    for (int a = 6; a >= 0; a--)
    {
        relayOn(a);
        delay(d);
        relayOff(a);
    }
}

void relayOn(int a)
{
    relayStatus = relayStatus|(1<<a);
    sendStatus(relayStatus);
```

```
    Serial.print("Relay "); Serial.print(a); Ser
}

void relayOff(int a)
{
    relayStatus = relayStatus^(1<<a);
    sendStatus(relayStatus);
    Serial.print("Relay "); Serial.print(a); Ser
}

void allOn()
{
    relayStatus = 255;
    sendStatus(relayStatus);
    Serial.println("All relays turned on");
}

void allOff()
{
    relayStatus = 0;
    sendStatus(relayStatus);
    Serial.println("All relays turned off");
}

void sendStatus(int a)
{
    digitalWrite(latch, LOW);
    shiftOut(data, clock, MSBFIRST, a);
```

```
    digitalWrite(latch, HIGH);
}

void setup()
{
    Serial.begin(9600);
    pinMode(latch, OUTPUT);
    pinMode(clock, OUTPUT);
    pinMode(data, OUTPUT);
}

void loop()
{
    char a = 0;
    Serial.flush();
    while (Serial.available() == 0) {}
    while (Serial.available() > 0) ❶
    {
        a = Serial.read();
    }
    switch (a) ❷
    {
        case '0' : relayOn(0); break;
        case '1' : relayOn(1); break;
        case '2' : relayOn(2); break;
        case '3' : relayOn(3); break;
        case '4' : relayOn(4); break;
        case '5' : relayOn(5); break;
```

```
          case '6' : relayOn(6); break;
          case '7' : relayOn(7); break;
          case '8' : relayOff(0); break;
          case '9' : relayOff(1); break;
          case 'A' : relayOff(2); break;
          case 'B' : relayOff(3); break;
          case 'C' : relayOff(4); break;
          case 'D' : relayOff(5); break;
          case 'E' : relayOff(6); break;
          case 'F' : relayOff(7); break;
          case 'G' : allOn(); break;
          case 'H' : allOff(); break;
          case 'Z' : waveHello(); break; ❸
          case '?' : showStatus(); break;
          default : Serial.print("Incorrect comman
          Serial.println("8 9 A B C D E F turns 0~
      }
  }
```

The sketch first declares the integer `relayStatus`, which holds the status of the relays. Think of this number in binary, with the least significant bit representing relay 0: if this bit is 1, the relay is on, and if it is 0, the relay is off. The custom `showStatus()` function sends the value of `relayStatus` in binary back to the serial interface, so the receiver can see which relays are on or off in their terminal, Serial Monitor, or other software. All the

functions that control a relay also send feedback via serial to the user, describing the completed action.

The `relayOn(int a)` function turns on relays, using the bitwise arithmetic operator OR (`|`) to activate the desired relay without interrupting other relays. The function receives the value `a` and performs a bitwise OR with the `relayStatus` variable, then it updates the relays using the new value of `relayStatus`. For example, if relays 0, 1, 2, and 3 are on, then the value of `relayStatus` in binary is currently B00001111. If the user then enters **`5`** in the Serial Monitor (or terminal software) to turn on relay 5, the program would switch bit 5 to 1, as follows:

```
B00001111 | // Current value of relayStatus
B00100000 = // Perform OR with 1 << 5—received b
B00101111   // New value of relayStatus
```

The `sendStatus()` function changes the relays accordingly by updating the shift register outputs.

The `relayOff(int a)` function turns relays off, using the bitwise arithmetic operator XOR (^) to deactivate the desired relay without interrupting other relays. For example, if relays 0, 1, 2, 3, and 5 are on, the value of `relayStatus` in binary is currently B00101111. If you enter **`A`** in the Serial Monitor or terminal

software to turn off relay 3, the program should switch bit 3 to 0, as follows:

```
B00101111 ^ // Current value of relayStatus
B00001000 = // Perform XOR with 1 << 3
B00100111   // New value of relayStatus
```

Again, the `sendStatus()` function changes the relays accordingly by updating the shift register outputs. Two additional functions, `allOn()` and `allOff()`, turn all relays on and off, respectively, by sending 255 (binary `B11111111`) and 0 (binary `B0000000`) to the shift register.

General operation is simple. The Arduino awaits a single character from the serial line ❶. Once a character has arrived, it is matched to a command ❷. The user can activate the simple `waveHello()` function ❸ by pressing **z** to turn the relays on and off one at a time, for testing and amusement. Finally, if the character isn't a command, the program sends a quick reference to serial ❹ so the user can learn the command set.

For a challenge, you can modify the sketch so the relay status is saved to the internal EEPROM when changed and the system sets the relays from the EEPROM data after a reset.

Now that you know how to control eight higher-current devices with a single TPIC6B595, I'll show you how to use two or more TPIC6B595s at once.

Using Multiple TPIC6B595s

You can easily use two or more TPIC6B595 shift registers at once to control 16 or more devices in the same way you would with the 74HC595, but with the ability to handle higher currents. Start by connecting each TPIC6B595's Clock lines together, connect their Latch lines together, then connect the Serial Out from the first shift register to the Serial In on the second shift register, repeating as required. For example, the schematic in Figure 8-12 shows double the number of LEDs controlled by Project #25.

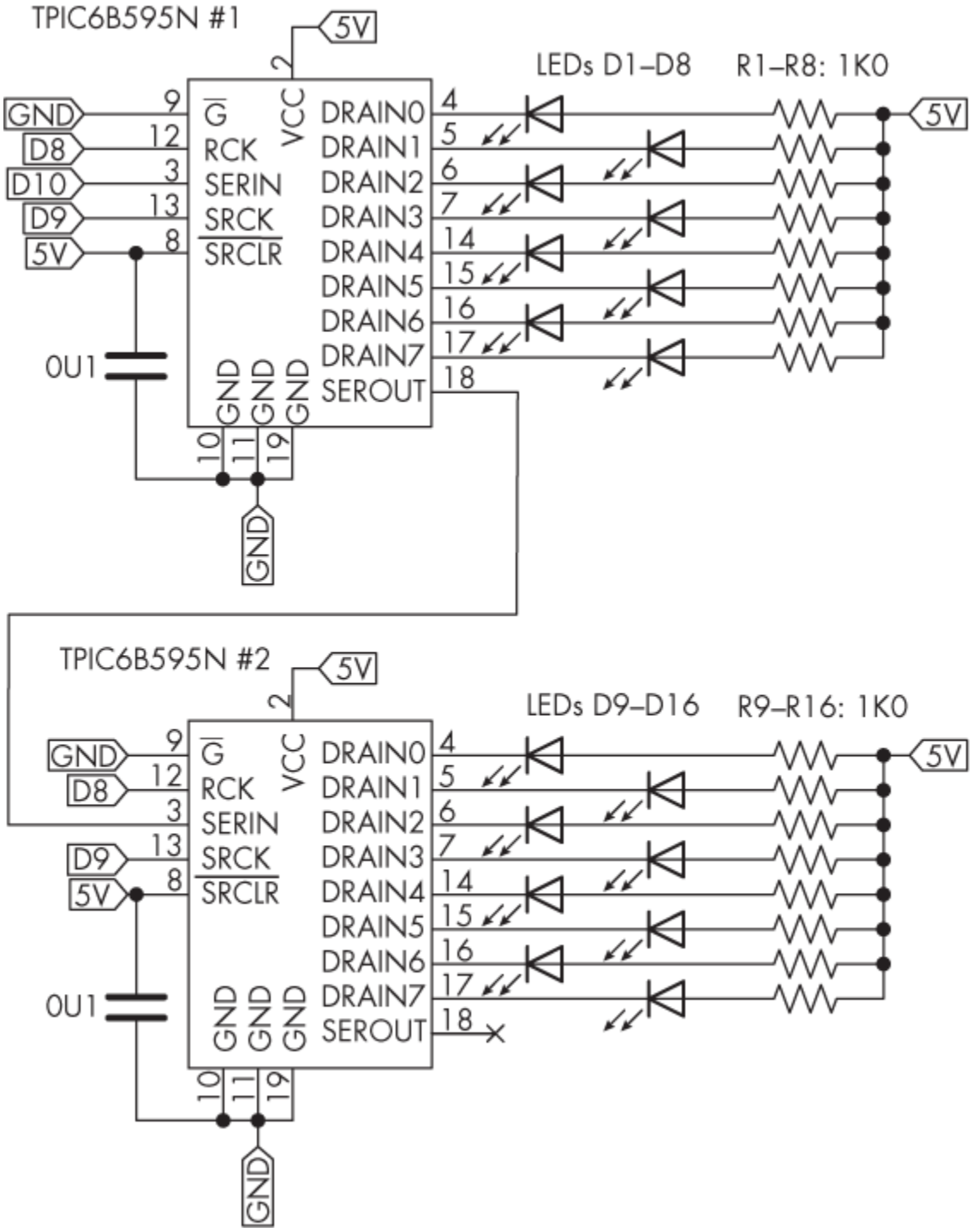


Figure 8-12: The schematic for controlling 16 LEDs with shift registers

Next, send out 2 bytes of data while the latch is low, instead of 1 byte. You'll need to send the byte for the last shift register in the

chain first. For example, to send out 2 bytes to the shift registers in Figure 8-12, you would use the following function:

```
void sendData(int a, int b)
{
digitalWrite(latch, LOW);
shiftOut(data, clock, MSBFIRST, b); // For TPIC
shiftOut(data, clock, MSBFIRST, a); // For TPIC
digitalWrite(latch, HIGH);
}
```

To add one or more additional shift registers, just add more parameters and `shiftOut()` functions, one for each extra register. You'll use multiple TPIC6B595s in the following project, along with a new type of LED.

Piranha-Style LEDs

There's an incredible range of LEDs available on the market, from tiny surface-mount LEDs to those large enough to serve as part of an automotive headlight. One example in between these two is the compact yet very bright Piranha-style LED; its through-hole packaging makes it easy to use. Figure 8-13 shows a pair of Piranha LEDs.

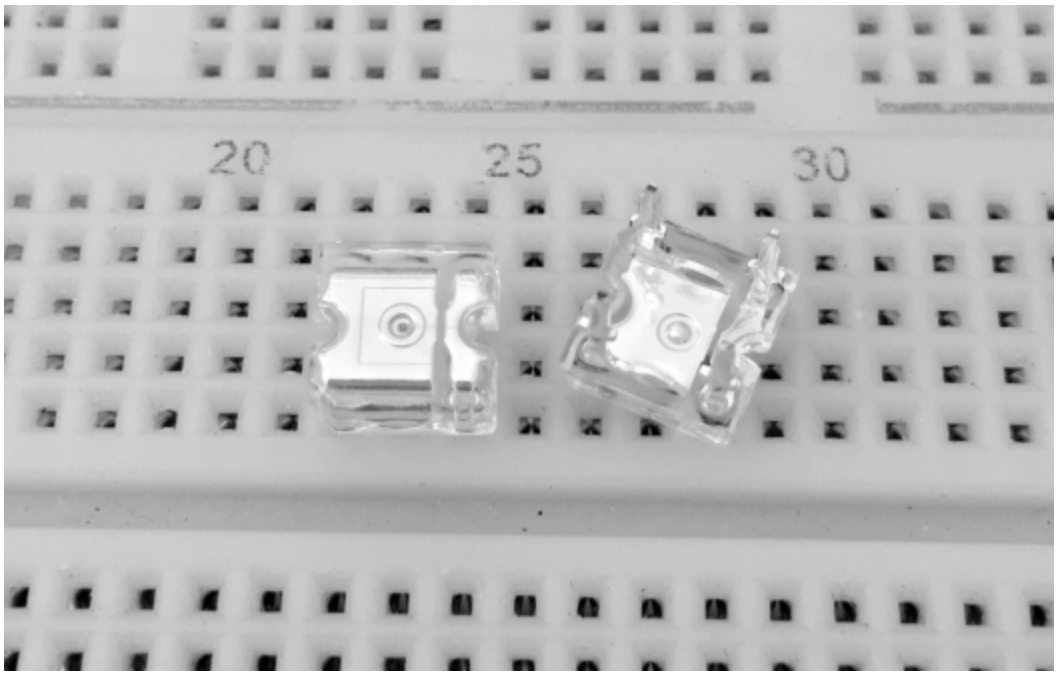


Figure 8-13: Two Piranha LEDs

Each LED has four legs, two anodes, and two cathodes. The two anode pins are electrically connected to each other, as are the two cathode pins. The anode side has the corner cut off at the bottom left, as well as a larger metallic surface compared to the cathode side. Figure 8-14 shows the schematic symbol for Piranha LEDs.



Figure 8-14: The schematic symbol for Piranha-style LEDs

When used alone, these LEDs operate safely at 20 mA of current at around 2 to 2.2 V DC. For the next project, you'll use them in

groups of four each, organized in series (rather than parallel to each other). In this configuration, each group of four LEDs will require 9 V DC power and a 47 Ω resistor to maintain the required current and a high brightness.

To calculate the required resistor to use with LEDs in series, you use formula $R = (V_s - V_f) / I_f$, where V_s is the power supply voltage, V_f is the forward voltage (recommended operating voltage) of the LEDs, and I_f is the recommended operating current for the LEDs. For the following project, you'd use 8 V at 20 mA as the LED forward voltage and operating current, with a 9 V supply. This gives you $R = (9\ V - 8\ V) / .02\ A$, which results in 50 Ω. We don't have a 50 Ω resistor, so 47 Ω is the nearest choice.

Project #27: Creating a Giant Seven-Segment LED Display

Multiple TPIC6B595s work well to drive large numbers of LEDs. In this project, you'll build one or more seven-segment LED displays to create large numerical displays you can use for various purposes. For instance, you might use them to indicate various data generated with an Arduino project, such as temperature, an event count, or time.

To construct a single-digit display, you'll need the following parts. For a larger display, multiply all parts other than the Arduino by the number of digits you'd like to create. This project shows you how to use four digits at once, so multiply by four if you'd like to follow the example exactly. For this project, use the PCB rather than a breadboard.

- An Arduino Uno or compatible board and USB cable
- Assorted jumper wires
- A 9 V power supply or wall wart with DC plug
- A TPIC6B595 shift register
- A 20-pin IC socket
- A 0.1 µF capacitor
- Thirty-two 5 mm Piranha-style LEDs
- Eight 47 Ω, 0.25 W, 1 percent resistors
- Four 3-contact 5.08 mm terminal blocks
- A PCB mount DC socket
- The Project #27 PCB

Figure 8-15 shows the project's schematic.

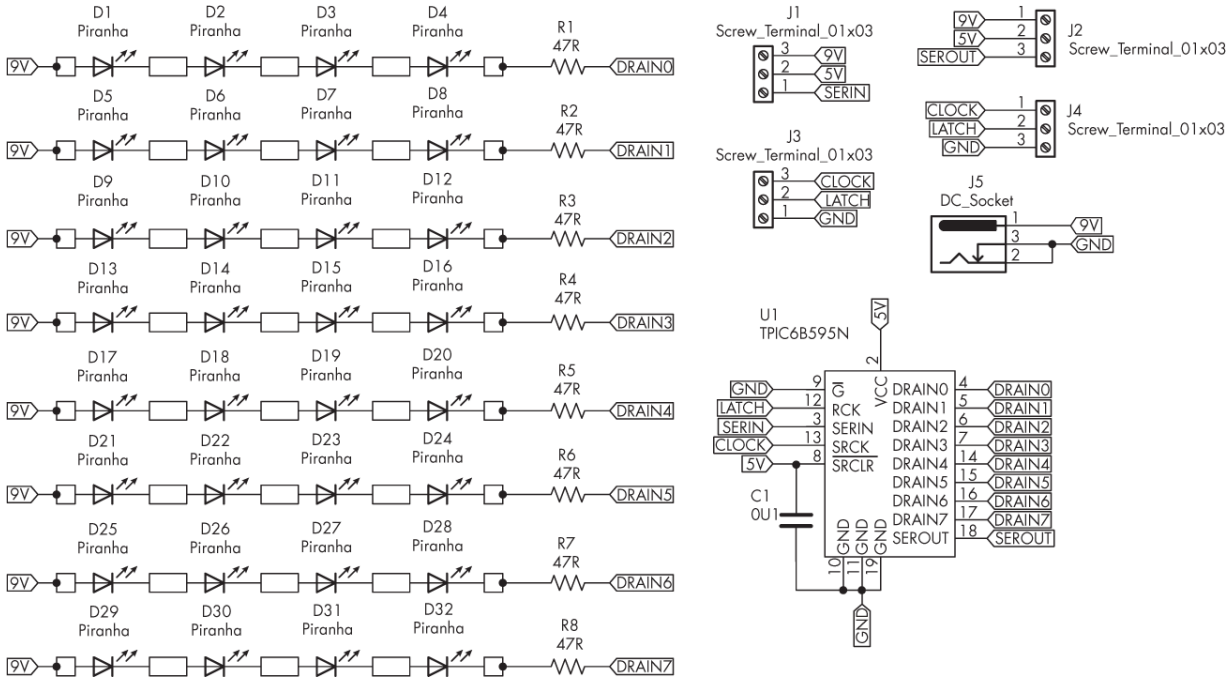


Figure 8-15: The schematic for Project #27

As you can see, the TPIC6B595 has “low-side” outputs. When an output is activated, the current starts from the power supply, goes through the items to be controlled, and then passes through the shift register DRAIN pins and out via GND.

Single-Digit Display

Figure 8-16 shows the simple PCB layout.

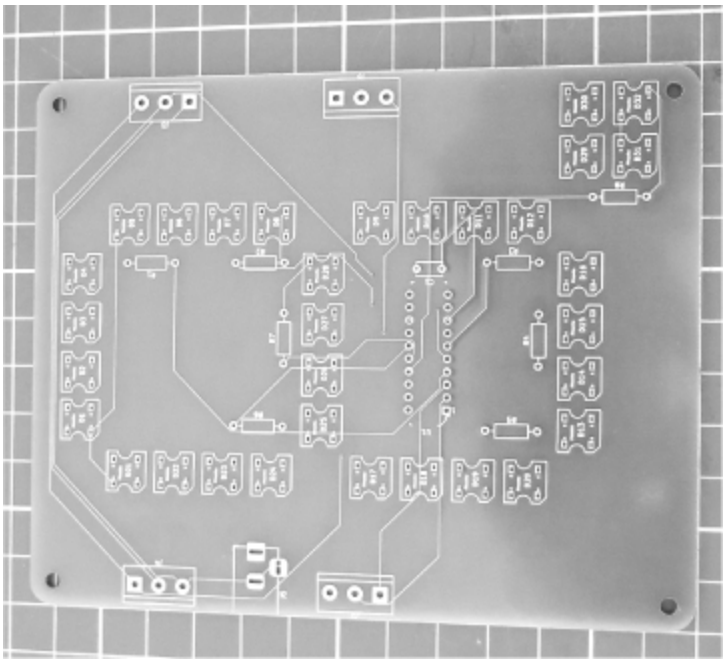


Figure 8-16: The unpopulated PCB for one digit

Always start by connecting the lowest-profile components: the resistors, the IC socket, the LEDs, and, finally, the DC socket and terminal blocks. Don’t be tempted to insert all the LEDs at once and then turn the PCB over, as some of the LEDs may come loose or fall out. Instead, solder them in one at a time. Orient all the LEDs with the anodes on the left and cathodes on the right (the left side of the PCB is the one with the DC socket). Once completed, your board should resemble the example shown in Figure 8-17.

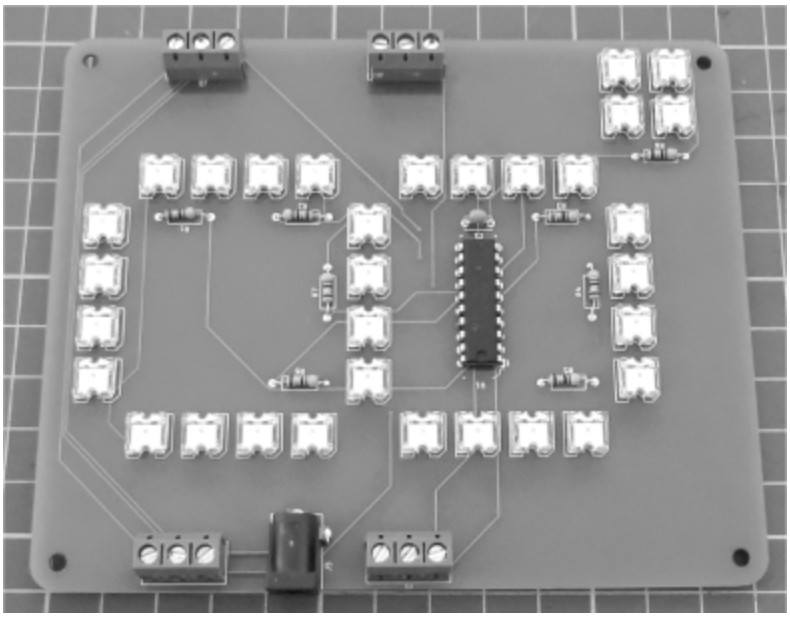


Figure 8-17: The completed PCB for one digit

Once you've set up the hardware, enter and upload the Project #27a sketch that follows, which demonstrates one digit (you can choose to build three more digits in the section "Four-Digit Display"). After uploading the sketch, you'll need to connect your Arduino and power supply to the PCB. With the PCB facing upward and the DC socket on your left, make the connections as described in Table 8-1.

Table 8-1: PCB to Arduino Connections

PCB left side	Arduino
Clock	D9

PCB left side	Arduino
Latch	D8
GND	GND
9V	Vin
5V	5V
Serial In	D10

Finally, connect a 9 V DC power supply to the DC socket on the PCB. The Arduino is powered by the 9 V supply via the Vin pin and feeds back 5 V to the display board to power the shift register. The display should count from zero to nine; then again with a decimal point beside each numeral, as shown in Figure 8-18; and finally repeat.

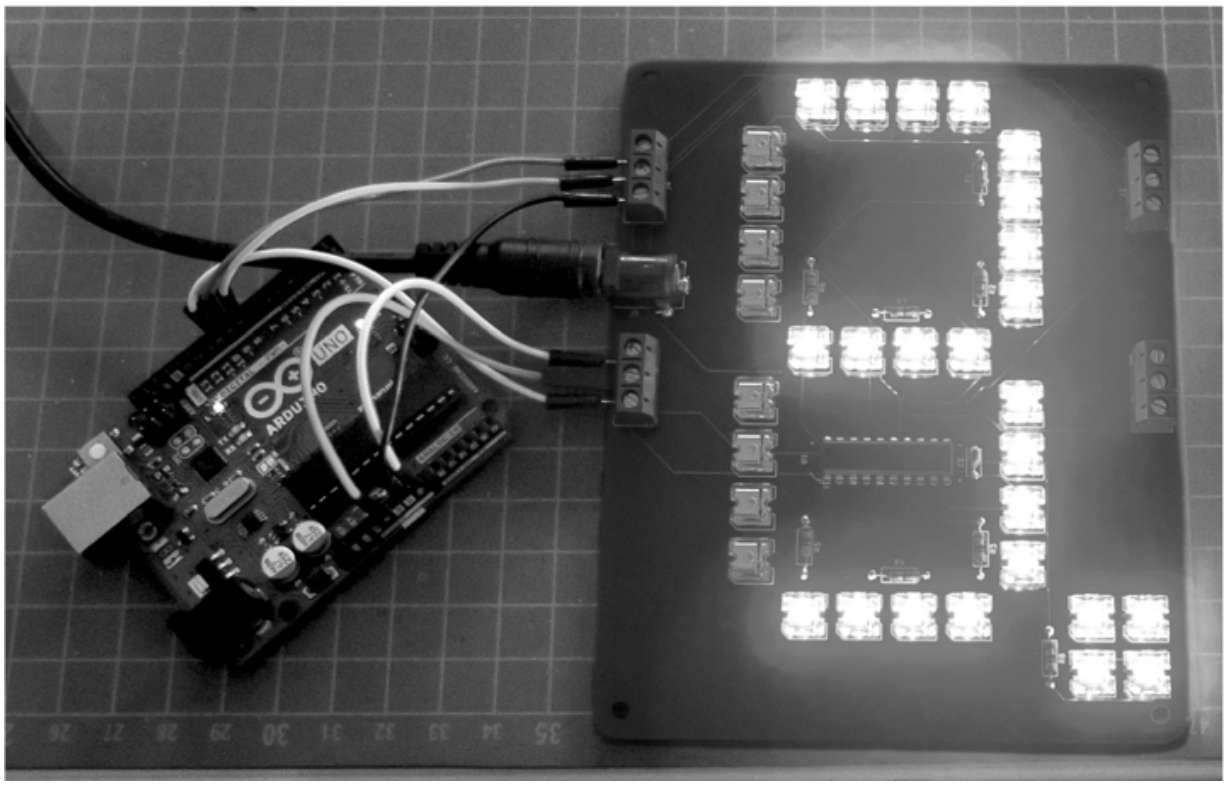


Figure 8-18: Example display board output

Because of the incredible brightness of the LEDs, I set the supply voltage to 7 V DC before taking the photo in Figure 8-18 for a clearer picture. Your display should be much brighter when operating at its designated 9 V DC.

Each segment of the digit consists of 28 Piranha LEDs, plus 4 more for the decimal point, driven by an output on the TPIC6B595. Therefore, you could consider the digit to be made up of seven LEDs, plus one more for the decimal point. These are controlled via the TPIC6B595's outputs in the same method as the relay board described in Project #26.

Let's see how this works:

```
  // Project #27a - Single giant seven-segment LED

❶ #define latch 8 // Latch RCK pin
  #define clock 9 // Clock SRCK pin
  #define data 10 // Data SERIN pin

  int digits[] = {B00111111,  // 0
                ❷ B00000110,  // 1
                  B01011011,  // 2
                  B01001111,  // 3
                  B01100110,  // 4
                  B01101101,  // 5
                  B01111101,  // 6
                  B00000111,  // 7
                  B01111111,  // 8
                  B01100111}; // 9

  void sendNumber(int a, boolean point)
  {
      if (point == false)
      {
          digitalWrite(latch, LOW);
          shiftOut(data, clock, MSBFIRST, digits[a
          digitalWrite(latch, HIGH);
      } else if (point == true)
      {
```

```
        digitalWrite(latch, LOW);
     ❸ shiftOut(data, clock, MSBFIRST, digits[a]
        digitalWrite(latch, HIGH);
    }
}

void setup()
{
    pinMode(latch, OUTPUT);
    pinMode(clock, OUTPUT);
    pinMode(data, OUTPUT);
}

void loop()
{
  ❹ for (int a = 0; a < 10; a++)
    {
        sendNumber(a,false);
        delay(1000);
    }
  ❺ for (int a = 0; a < 10; a++)
    {
        sendNumber(a,true);
        delay(1000);
    }
}
```

The sketch first defines the Arduino digital pins used for connecting to the shift register's Latch, Clock, and Data pins ❶. For the array of 8 bytes ❷, each byte represents the eight outputs that are used to control the segments on the display. For example, the number 1 is represented in binary as `B00000110`, as you want to turn on the second and third segments of the display to display 1.

The custom `sendNumber()` function sends the requisite data to display each digit to the shift register, which then sets the appropriate outputs. This function accepts the digit to display and accepts `true` or `false` as the second parameter, which is used to turn the decimal point on or off. If the decimal point is required, the sketch uses the OR function `|` ❸ (as in Project #7 in Chapter 2) to turn on the most significant bit of the shift register (bit 7), which controls DRAIN7.

The `void setup()` function sets the required digital output pins. Finally, the sketch demonstrates the display by counting from zero to nine and back again ❹ and then again with the decimal point turned on ❺.

You now have a large, impressive numerical LED display that can be seen from quite a distance. If the LEDs are too bright for

your liking, you can increase the value of the resistors, perhaps to 180 or 270 Ω. Be sure to use 0.25 W (1/4 W) resistor types.

Four-Digit Display

You can build and use multiple display boards for larger numerical projects. In this example, you'll use four boards to display numbers of up to four digits. You might use this to display data or as a large, bright clock.

Construct a second, third, and fourth board just as you did in the previous section, with one difference: these new boards won't need the DC socket, as the first display board will act as your power source. Once the extra boards are assembled, connect them by bridging the terminal blocks on each side to each other. The matching labels on the rear of the display boards, as shown in Figure 8-19, will help you do so.

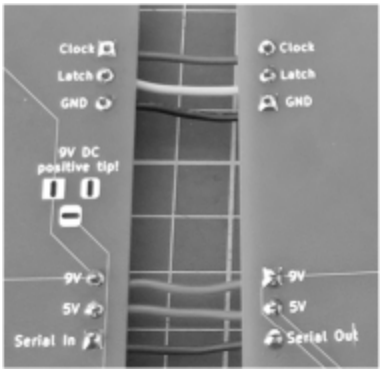


Figure 8-19: The rears of two display boards

Next, enter and upload the Project #27b demonstration sketch. Once the upload completes, connect the Arduino to the leftmost display board and connect the 9 V power supply. The display boards should show random four-digit numbers, with random placement of the decimal point, as shown in Figure 8-20.

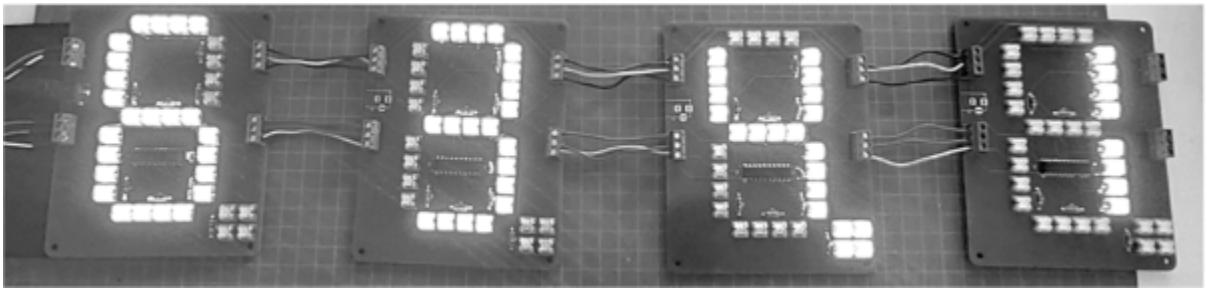


Figure 8-20: The four display boards operating at 7 V DC

Let's see how this works:

```
// Project #27b - Four giant seven-segment LED d

#define latch 8 // Latch RCK pin ❶
#define clock 9 // Clock SRCK pin
#define data 10 // Data SERIN pin

int digits[] = {B00111111,   // 0 ❷
                B00000110,   // 1
                B01011011,   // 2
                B01001111,   // 3
                B01100110,   // 4
                B01101101,   // 5
```

```
                B01111101,   // 6
                B00000111,   // 7
                B01111111,   // 8
                B01100111};  // 9

void sendNumbers(int numbers[], int dp)
{
    digitalWrite(latch, LOW);
    for (int i = 0; i < 4; i++)
    {
        int dig_idx = numbers[i]; ❸
        if (dp == i) {
        // Display the digit
           shiftOut(data, clock, MSBFIRST, digit
        } else {
            shiftOut(data, clock, MSBFIRST, digi
        }
    }
    digitalWrite(latch, HIGH);
}

void setup()
{
    randomSeed(analogRead(0)); ❺
    pinMode(latch, OUTPUT);
    pinMode(clock, OUTPUT);
    pinMode(data, OUTPUT);
}
```

```
void loop()
{
    int numbers[4] = {random(0, 10), random(0, 1
    sendNumbers(numbers, random(0, 3)); ❼
    delay(1000);
}
```

The sketch for four boards has a few differences from the Project #27a sketch to make using multiple displays easier. Once again, it defines the Arduino digital pins used for connecting to the shift register's Latch, Clock, and Data pins ❶ and defines the array of 8 bytes ❷ that represents the eight outputs used to control the segments on the display.

The `void sendNumbers()` function accepts two parameters: an array of four numbers (one for each display board) and an integer that represents on which boards to display the decimal point. The sketch checks for this ❸. If the parameter is 1, the required bit for the decimal point display is included in the byte sent to the shift register ❹.

As you're displaying random numbers, the generator is seeded with the analog input data ❺, and then the array for numbers to be displayed is also filled with random numbers ❻ and sent

to the display boards ❼. Finally, the sketch includes a delay before displaying more random numbers.

You can display all sorts of numerical data with these boards. You might also add your own characters by adding more elements to the digit arrays. For example, a degree symbol for temperature would be as follows:

```
B01100011 // Binary representation of °
```

For a challenge, pair up four display boards with an Arduino and a real-time clock module to make a large, bright clock. You might add a thermometer as well. I hope you enjoy using these display boards as much as I enjoyed designing them.

Moving On

In this chapter, you learned how to control currents and voltages using the TPIC6B595 shift register, a more capable alternative to the popular 74HC595. You can now control items that draw more current than an Arduino's digital I/O pin can safely handle. You also learned how to use bright Piranha-style LEDs for excellent indicators.

In the next chapter, you'll use MP3 player modules to create digital music players and sound boards for various purposes.

9

BUILDING A DIGITAL MUSIC PLAYER AND SOUND BOARD



Digital audio players became popular during the 2000s but faded away as smartphones began to dominate the market. However, there are still plenty of uses for discrete MP3 players, such as audio playback in toys, as announcement systems, or as simple audio players. Thanks to the low cost of the bare MP3 player modules, you can easily build fun, interesting MP3 player projects.

In this chapter, you'll learn to:

- Build a simple MP3 player just for listening to music, with none of the distractions of a smartphone
- Make an MP3 audio sound board for direct playback control

You can use the devices you build in this chapter to play any audio you like and can modify them for the applications mentioned previously in your own future projects.

The YX6300 MP3 Module

For the MP3 projects in this chapter, you'll use a YX6300-type compact module, such as PMD Way part 725600, shown in Figure 9-1.

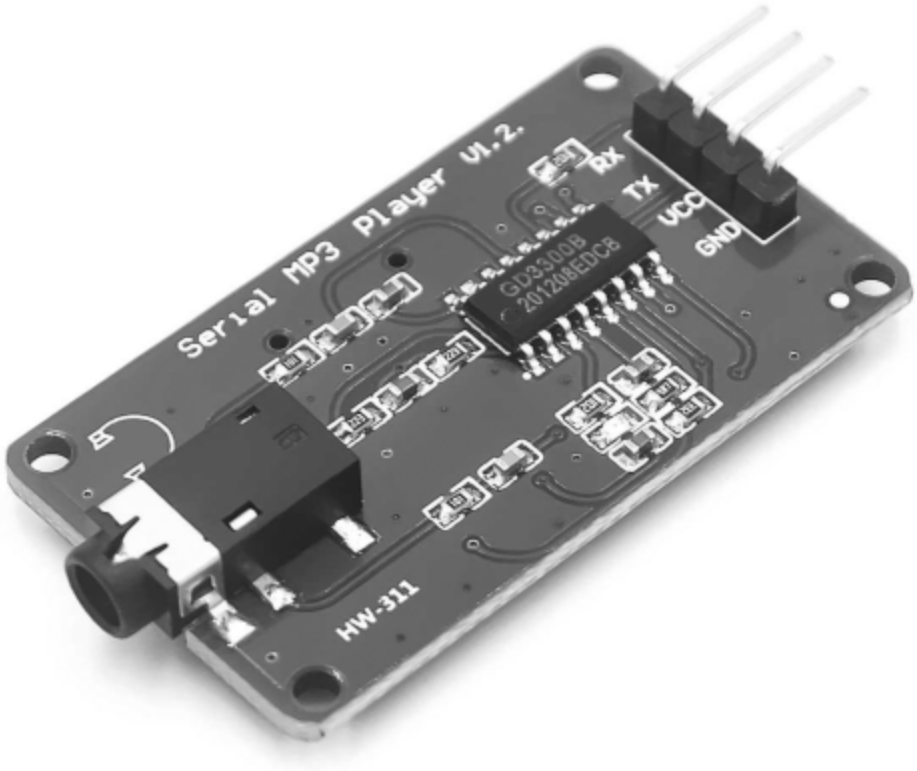


Figure 9-1: An MP3 player module (front)

On the reverse side of the module is a microSD flash memory card socket, which can be used with cards up to 32GB, as shown in Figure 9-2.

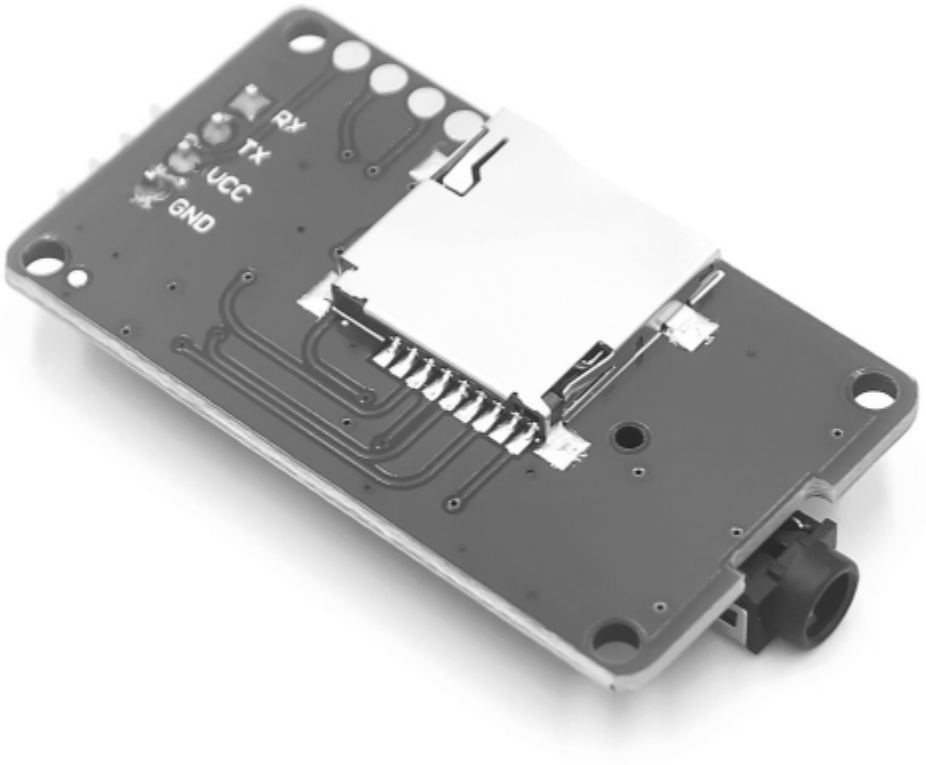


Figure 9-2: The back of an MP3 player module

When buying a flash memory card to use with the MP3 module, be sure to purchase an adapter as well so you can insert it into normal SD memory card sockets on PCs for file transfer.

I chose to base this chapter on the YX6300-type MP3 module for a few reasons. First, you can connect headphones, amplifiers, or speakers to the audio output via the module's 3.5 mm stereo audio jack socket, with no extra circuitry required. Second, commands and information are sent via serial data (a UART) using an Arduino software serial port, so we need to connect

only four pins to the Arduino circuit. Finally, the unit is inexpensive and widely available.

As a test before you start the first project, let's use the module to play some audio. First, prepare three or four MP3 audio files of your choosing for playback on your PC and copy them to the microSD card. Next, connect the MP3 module to the Arduino, following the schematic in Figure 9-3.

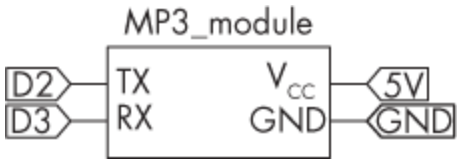


Figure 9-3: The MP3 player test circuit

Carefully insert the memory card into the module, with the label facing upward, as shown in Figure 9-4. The card will slide in and then bump up against a spring lock—push it in a little farther until the lock clicks. (To remove the card, gently push against the card, which should spring in a little and then pop out.)

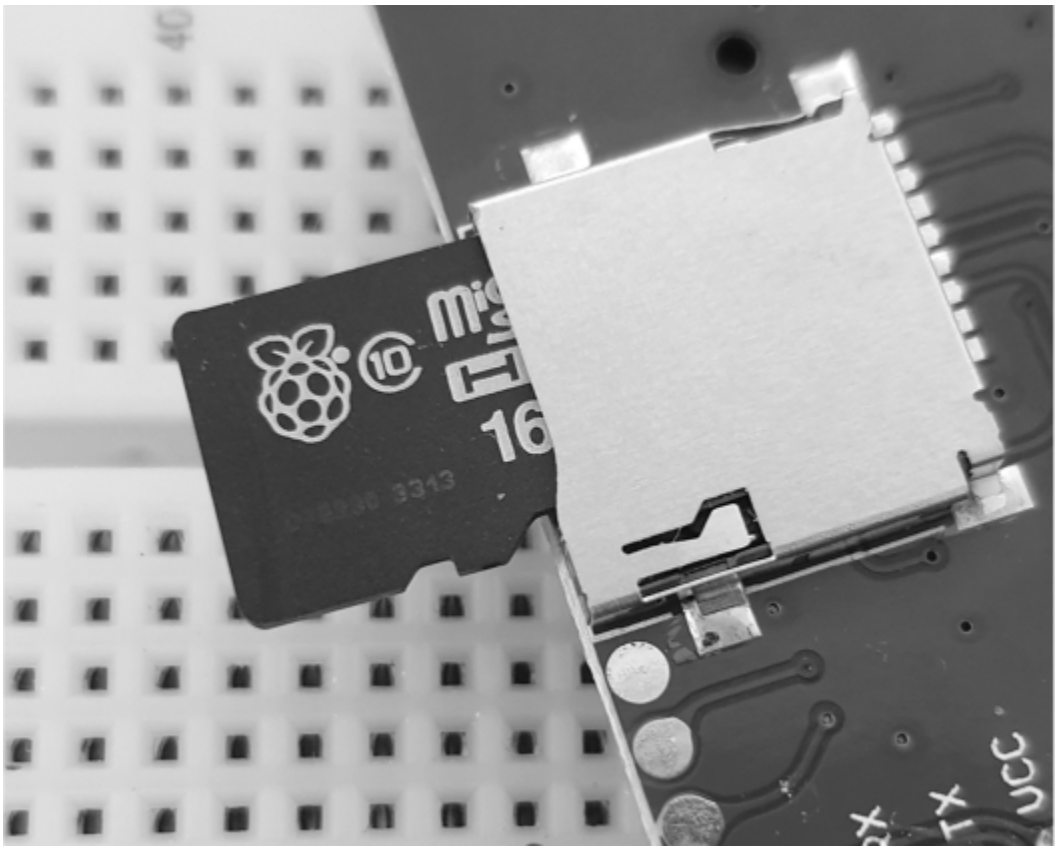


Figure 9-4: Inserting the memory card into an MP3 player module

Finally, plug some headphones or amplified speakers into the MP3 module and then enter and upload the Listing 9-1 sketch. After a few moments, the module should play the first 10 seconds of the first audio file on the memory card and then repeatedly skip to the next and play it for 10 seconds. The LED on the rear of the module should stay on while the memory card is inserted and should blink when playing audio. If yours doesn't work, check the wiring, including the TX/RX back to the Arduino.

Listing 9-1 shows how this works.

```
❶ #define HEADER_START   0x7E
  #define HEADER_VERSION 0xFF
  #define TAIL_END       0xEF
  #define CMD_PLAY_NEXT  0x01
  #define CMD_PLAY_PREV  0x02
  #define CMD_PLAY_IDX   0x03

  int8_t commands[8] = {0, 0, 0, 0, 0, 0, 0, 0};
  // Holds command set to send to MP3

  #include <SoftwareSerial.h>
  SoftwareSerial MP3(2, 3); // Module TX to D2; mo

  void controlMP3(int8_t command, int16_t data)
  {
      commands[0] = HEADER_START;        // Header
      commands[1] = HEADER_VERSION;      // Header
      commands[2] = 0x06;                // Header
      commands[3] = command;             // Comman
      commands[4] = 0x00;                // Feedba
      commands[5] = (int8_t)(data >> 8); // Data b
      commands[6] = (int8_t)(data);      // Data b
      commands[7] = 0xEF;                // End by
    ❷ for (int i = 0; i < 8; i++)
      {
          MP3.write(commands[i]);
```

```
    }
}

void setup()
{
    MP3.begin(9600);
 ❸ controlMP3(0x09, 0x02); // Select flash card
    delay(500);
 ❹ controlMP3(0x0D, 0); // Resume playback, no t
    delay(500);
}

void loop()
{
 ❺ controlMP3(0x01, 0); // Next track, zero data
 ❻ delay(10000);
}
```

Listing 9-1: Testing a serial MP3 player

This sketch sends commands and data to the MP3 module via a serial UART connection with the Arduino. Each data packet, containing both the command and the associated data, consists of 8 hexadecimal bytes of data.

To save time, the sketch defines some useful command values as variables ❶. These commands are assembled into the `commands[]` array and then sent out via a software serial port ❷. This uses the first command listed in Table 9-1 to initialize the MP3 player and select the microSD card socket as the source of audio files ❸ and then starts playing the audio files ❹. The sketch plays the next audio file found on the microSD card ❺ for 10 seconds ❻, and then the process repeats.

Table 9-1 describes the commands you can use with your player.

Table 9-1: MP3 Player Commands

Command set	Purpose
`7E FF 06 09 00 00 02 EF`	Initializes MP3 player, uses the microSD card
`7E FF 06 0D 00 00 00 EF`	Resumes/starts playback

Command set	Purpose
`7E FF 06 **0E** 00 **00** **00** EF`	Pauses playback
`7E FF 06 **16** 00 **00** **00** EF`	Stops playback, resets to first track
`7E FF 06 **02** 00 **00** **00** EF`	Plays previous track
`7E FF 06 **01** 00 **00** **00** EF`	Plays next track
`7E FF 06 **05** 00 **00** **00** EF`	Turns volume down
`7E FF 06 **04** 00 **00** **00** EF`	Turns volume up

Of the two bold bytes in each command group, the first is the command byte, and the second is the data bytes. These are the two parameters used in the sketch's `controlMP3()` function to

instruct the MP3 player. For basic playback use, you don't need to send data bytes.

In the next project, you'll test these commands by building your own MP3 player with some control functions.

Project #28: Building a Simple MP3 Player

In this project, you'll create an MP3 player with typical play, pause, volume, and track forward/backward controls. You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- An MP3 player module with a microSD card of MP3 audio files
- A solderless breadboard
- Assorted jumper wires
- Headphones or cables to audio amplifier and speakers
- An analog 6- or 12-button keypad circuit from Chapter 1

To build this project, you can either reuse the 12-button keypad you built in Project #3, using only 6 of the buttons, or Project #2. Don't forget to update the analog values on the analog keypad library if you're using the 12-button keypad. Assemble the circuit as shown in Figure 9-5.

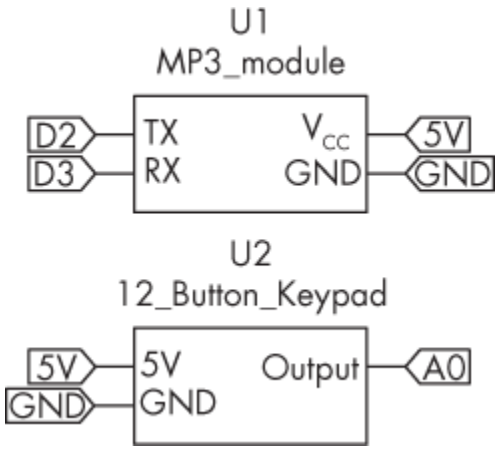


Figure 9-5: The schematic for Project #28

Enter and upload the Project #28 sketch. After a few moments, press button 7; the music in the MP3 files you arranged on the memory card should start playing. Test the other buttons to ensure their functions work. If some or all do not work, check that the button analog values match the actual keypresses using the process described in Project #2, and update the values in the analogkeypad library as described in Project #3.

Let's see how this works:

```
  // Project #28 - Simple MP3 player

❶ #define HEADER_START   0x7E
  #define HEADER_VERSION 0xFF
  #define TAIL_END       0xEF
  #define CMD_PLAY_NEXT  0x01
  #define CMD_PLAY_PREV  0x02
```

```
  #define CMD_PLAY_IDX   0x03

❷ int8_t commands[8] = {0, 0, 0, 0, 0, 0, 0, 0};
  // Holds command set to send to MP3
  #include <SoftwareSerial.h>
  SoftwareSerial MP3(2, 3); // module TX to D2; mo
  #include <analogkeypad.h>
  analogkeypad keypad;

  void controlMP3(int8_t command, int16_t data)
  {
      commands[0] = HEADER_START;
      commands[1] = HEADER_VERSION;
      commands[2] = 0x06;
    ❸ commands[3] = command;8
      commands[4] = 0x00;
    ❹ commands[5] = (int8_t)(data >> 8);
      commands[6] = (int8_t)(data);
      commands[7] = TAIL_END;
      for (int i = 0; i < 8; i++)
      {
        ❺ MP3.write(commands[i]);
      }
      delay(100);
  }

  void setup()
  {
```

```
  ❻ MP3.begin(9600);
    controlMP3(0x09, 0x02); // Select uSD card f
    delay(500);
}

void loop()
{
    switch (keypad.readKeypad()) // Read button
    {
        case 7 : controlMP3(0x0D, 0); break; //
        case 8 : controlMP3(0x0E, 0); break; //
        case 9 : controlMP3(0x02, 0); break; //
        case 10 : controlMP3(0x01, 0); break; //
        case 11 : controlMP3(0x05, 0); break; //
        case 12 : controlMP3(0x04, 0); break; //
    }
    delay(250);
}
```

The sketch first declares the array used for storing the commands to control the MP3 module ❶ and then initializes the software serial port for Arduino-to-MP3-player control, as well as the library for the analog button keypad ❷. The custom `void controlMP3()` function accepts the command and data parameters required to control the MP3 player, inserts them into the array ❸ ❹, and sends the command to the MP3 player

via the software serial port ❺. The sketch then starts the software serial port and sends the initialize command to instruct the MP3 player to use the microSD card ❻.

Once operation has begun, the sketch loops, awaiting a response from the analog button circuit in `void loop()`, where each button is assigned a `controlMP3()` function along with the appropriate command data from Table 9-1.

You now have a working MP3 audio player that you can embed, modify, or make portable for your own distraction-free source of audio (as opposed to smartphones, whose games, messages, and other content can be distracting). The USB cable connected to the Arduino is powering the project, but you could instead use a power bank or AC-to-USB adapter and place everything inside an enclosure of your own design. You might also draw a small map to explain which buttons have which effects, as shown in Figure 9-6, if you're demonstrating the project to your friends or family.

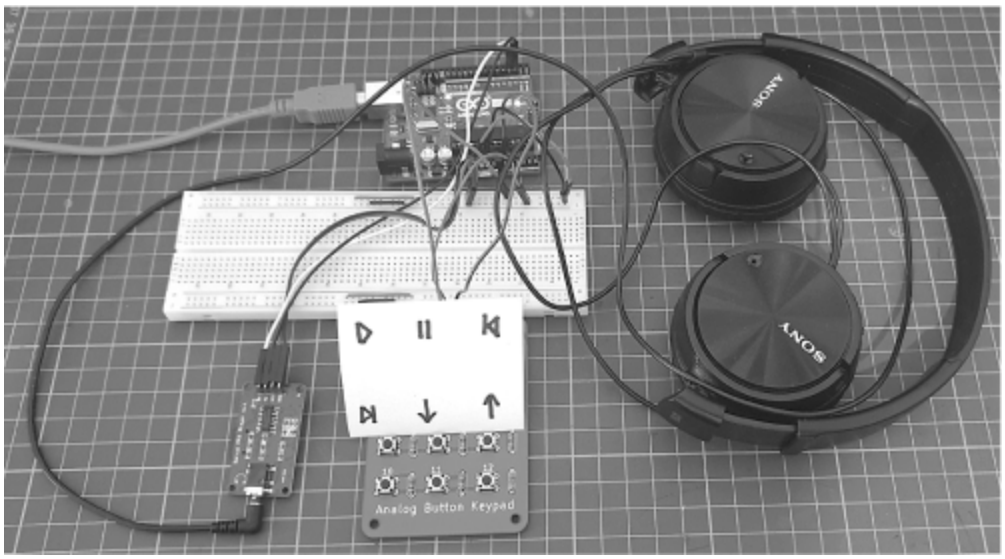


Figure 9-6: Example of Project #28 with the 12-button keypad and a user reference for the playback controls

In the next project, you'll use the MP3 player module you've just built to create a sound board.

Project #29: Building an MP3 Player Sound Board

In this project, you'll use the hardware from Project #28 with the 12-button board to create a *sound board*, a device that plays preset audio tracks at the press of a button. Sound boards are often used in broadcasting, in toys, or as part of exhibits in museums to offer information for the visually impaired. This project uses buttons to initiate audio playback, but you can use it as a framework for a device that plays audio upon other actions the Arduino detects, such as triggers in your code or outputs from sensors, switches, and so on.

Before uploading the Project #29 sketch, rename the MP3 audio files on your microSD card, using the numbering convention shown in Figure 9-7, so the MP3 player can search for the exact audio file to play at the press of a button. For example, when the player searches for audio file 1, the MP3 will play the *001001.mp3* file. Accordingly, you should name file 4 *004004.mp3*, file 12 *012012.mp3*, and so on. Store these files in a folder named *01*.

```
D:\01>ls -la
total 4354
drwxr-xr-x    2 john     Administ        0 Sep 21 13:57 .
drwxr-xr-x    2 john     Administ        0 Jan  1  1980 ..
-rw-r--r--    1 john     Administ    15396 Sep 21 10:46 001001.mp3
-rw-r--r--    1 john     Administ     7680 Sep 21 10:47 002002.mp3
-rw-r--r--    1 john     Administ    21986 Sep 21 10:47 003003.mp3
-rw-r--r--    1 john     Administ   739032 Sep 21 11:34 004004.mp3
-rw-r--r--    1 john     Administ  1857384 Sep 21 11:33 005005.mp3
-rw-r--r--    1 john     Administ   106076 Sep 21 11:32 006006.mp3
-rw-r--r--    1 john     Administ   391296 Sep 21 13:02 007007.mp3
-rw-r--r--    1 john     Administ   614664 Sep 21 13:01 008008.mp3
-rw-r--r--    1 john     Administ   580394 Sep 21 13:01 009009.mp3
-rw-r--r--    1 john     Administ   711722 Sep 21 14:09 010010.mp3
-rw-r--r--    1 john     Administ  3642525 Sep 21 14:09 011011.mp3
-rw-r--r--    1 john     Administ   221201 Sep 21 14:10 012012.mp3
```

Figure 9-7: An example of MP3 files in the 01 *folder, with filenames structured for sound board use*

These MP3 files can contain any audio you like. Just for fun, you might try downloading some sound effects such as animal sounds or ambient daily noise from websites such as *https://www.zapsplat.com*. I've included a folder of sounds from Zapsplat with the book download files at *https://nostarch.com*

/arduino-arduinians. Alternately, you could load the MP3 with your 12 favorite songs.

Once you've gathered and organized your MP3 files and inserted the microSD card into the player, enter and upload the following sketch:

```
// Project #29 - MP3 player sound board

#define HEADER_START   0x7E
#define HEADER_VERSION 0xFF
#define TAIL_END       0xEF
#define CMD_PLAY_NEXT  0x01
#define CMD_PLAY_PREV  0x02
#define CMD_PLAY_IDX   0x03

int8_t commands[8] = {0, 0, 0, 0, 0, 0, 0, 0};
// Holds command set to send to MP3
#include <SoftwareSerial.h>
SoftwareSerial MP3(2, 3); // Module TX to D2; mo
#include <analogkeypad.h>
analogkeypad keypad;

void controlMP3(int8_t command, int16_t data)
{
    commands[0] = 0x7E;
    commands[1] = 0xFF;
```

```
    commands[2] = 0x06;
    commands[3] = command;
    commands[4] = 0x00;
    commands[5] = 0x01;
    commands[6] = (int8_t)(data);
    commands[7] = 0xEF;
    for (int i = 0; i < 8; i++)
    {
        MP3.write(commands[i]);
    }
    delay(100);
}

void setup()
{
    MP3.begin(9600);
    controlMP3(0x09, 0x02); // Select uSD card f
    delay(500);
}

void loop()
{
    switch (keypad.readKeypad()) // Read button
    {
     ❶ case 1 : controlMP3(0x0F, 1); break; // P
        case 2 : controlMP3(0x0F, 2); break; //
        case 3 : controlMP3(0x0F, 3); break; //
        case 4 : controlMP3(0x0F, 4); break; //
```

```
            case 5 : controlMP3(0x0F, 5); break;
            case 6 : controlMP3(0x0F, 6); break;
            case 7 : controlMP3(0x0F, 7); break;
            case 8 : controlMP3(0x0F, 8); break;
            case 9 : controlMP3(0x0F, 9); break;
            case 10 : controlMP3(0x0F, 10); break;
            case 11 : controlMP3(0x0F, 11); break;
            case 12 : controlMP3(0x0F, 12); break;
        }
        delay(250);
    }
```

This sketch operates in the same manner as that of the previous project, except that different types of commands and data are sent to the player when you press the buttons. These new commands play whatever track number matches the value of the data sent in the command. For example, to play file *001001.mp3*, the sketch sends

```
7E FF 06 0F 00 01 01 EF
```

using `controlMP3(0x0F, 1)` ❶. The command for direct track playback is `0x0F`, and the data value (1) is the number of the track's filename (*001001.mp3*). The rest of the buttons follow the

same command format, with data values that match the other MP3 filenames.

Moving On

In this chapter, you learned how to build your own MP3 player and use it to listen to music and play sound effects. There are plenty of other fun ways to use your MP3 module, such as creating audible outputs for projects instead of LEDs or displays. For a final challenge, you might write your own MP3 player Arduino library.

In the next chapter, you'll learn how to use a new type of temperature sensor, OLED displays, and multiple I^2C buses with your Arduino.

10

USING MULTIPLE I²C DEVICES WITH THE SAME ADDRESS



In your Arduino journey, you've likely come across devices that interface via the I²C bus, a simple two-wire data bus for which the Arduino is considered the *primary* device, and each additional device or IC on the bus is a *secondary* device. Each secondary device has its own address that allows the Arduino to communicate with it. If two or more devices have the same bus address, however, you simply can't use them together on the same bus.

This chapter shows you how to use two or more devices with the same bus address with the help of the TCA9548A I²C switch IC. You'll learn to:

- Build an I²C address scanner to determine device bus addresses
- Use compact graphic organic light-emitting diode (OLED) displays
- Use the BMP180 temperature and air pressure sensor
- Combine the BMP180 and OLED devices

To prepare you to work with multiple I²C devices at once, I'll first introduce a simple tool that reports the bus address of a device connected to the Arduino's I²C bus: the I²C bus scanner.

NOTE

If you're not familiar with the I²C bus, you can review Chapter 19 of Arduino Workshop*, 2nd edition.*

Project #30: Using an I²C Bus Scanner

In this project, you'll learn to use an I²C bus scanner. This tool helps you gather information on parts or devices whose documentation you can't find, such as the part's bus address. It's

also ideal for quick I^2C bus testing once you've connected a device to the bus.

You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- A solderless breadboard or mounting for the I^2C device
- Various jumper wires

The hardware connection will depend on the I^2C device under test. As an example, I'll use the BMP180 sensor for temperature, barometric pressure, and altitude, as shown in Figure 10-1.

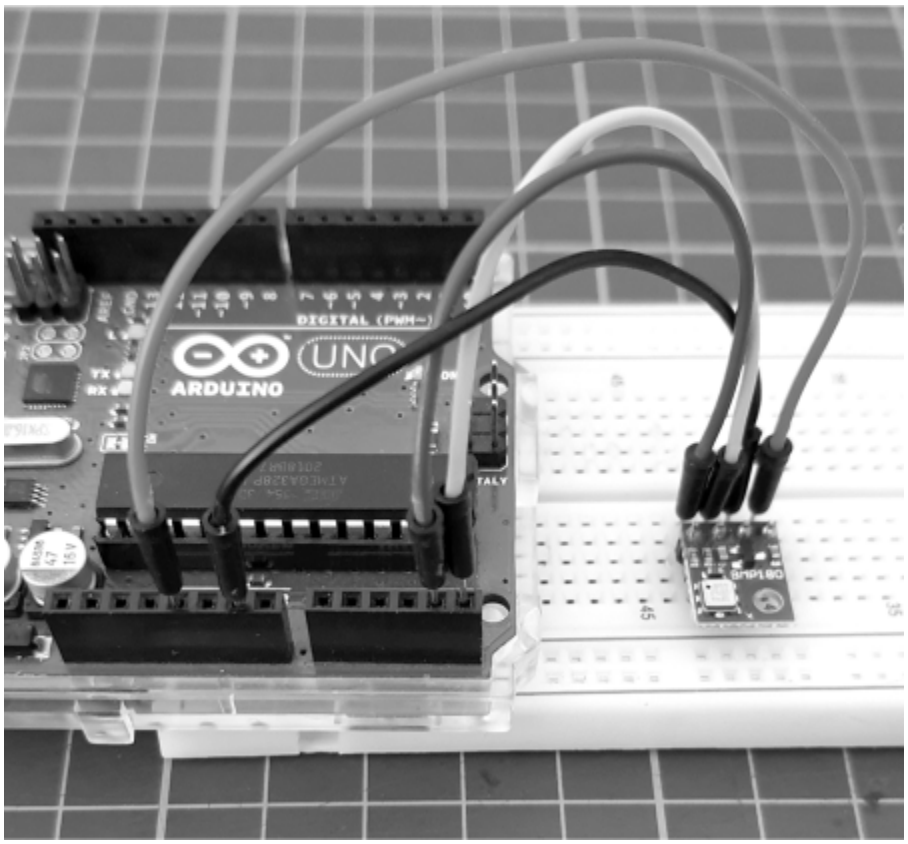


Figure 10-1: The BMP180 sensor board

You'll also use the BMP180 later in the chapter and elsewhere in this book, but for this project, you can use any I^2C device that's 5 V tolerant. If your device isn't a module, don't forget to use pull-up resistors as you normally would.

The sketch for this project is provided with the Arduino IDE. Open and upload the i2c_scanner sketch included with the Wire library examples, by selecting **File ▸ Examples ▸ Wire ▸ i2c_scanner**.

The sketch should interrogate the bus and report the address (or addresses) found in the Serial Monitor, as shown in Figure 10-2.



Figure 10-2: The results of the I²C scanner sketch

If you have two or more devices on the bus, the scanner will report both addresses, but you won't know which device belongs to which address. For example, Figure 10-3 shows the bus addresses for the two devices connected to the bus. When you're using two or more devices at once with the scanner, it's okay to remove one or more devices even when the power is on, as the I^2C bus is capable of "hot swapping." That is, it's designed to allow for devices to be connected or removed while operating.



Figure 10-3: The results of the I²C scanner sketch with two devices

The sketch is quite simple. Let's see how it works:

```
// Project #30 - I2C bus address scanner

void loop()
{
    int nDevices = 0;
    Serial.println("Scanning…");
 ❶ for (byte address = 1; address < 127; ++addre
    {
        Wire.beginTransmission(address);
        byte error = Wire.endTransmission();
     ❷ if (error == 0)
        {
            Serial.print("I2C device found at ad
            if (address < 16)
            {
                Serial.print("0");
            }
            Serial.print(address, HEX);
```

```
            Serial.println(" !");
            ++nDevices;
        }
      ❸ else if (error == 4)
        {
            Serial.print("Unknown error at addre
            if (address < 16)
            {
                Serial.print("0");
            }
            Serial.println(address, HEX);
        }
    }
    if (nDevices == 0)
    {
      ❹ Serial.println("No I2C devices found\n")
    } else
    {
        Serial.println("done\n");
    }
    delay(5000); // Wait 5 seconds for next scan
}
```

The sketch starts and finishes an I^2C connection to every possible I^2C address location (0 through 127) ❶. If a device exists on the address being tested, the sketch returns an error

code of 0 ❷; if an error occurs, it returns a 4 ❸. If it doesn't find any devices, it prints the message `No I2C devices found` ❹.

Now that you've seen how to connect an I²C device or two to the bus, let's get started with the TCA9548A.

The TCA9548A Breakout Board

The TCA9548A, shown on a breakout board in Figure 10-4, allows you to connect the single I²C bus from your Arduino to one of eight separate buses that emanate from the TCA9548A, each of which operates in the same manner as the Arduino's original I²C bus.



Figure 10-4: The TCA9548A breakout board

The TCA9548A also handles voltage translation, enabling you to use devices with the host Arduino that operate on different

voltages—a convenient benefit, since an increasing number of devices operate only at 3.3 V or 1.8 V. Unfortunately, the TCA9548A is available in surface-mount format only, so it can't be used with a solderless breadboard or prototyping board. Instead, you'll need to use the module.

The TCA9548A breakout board should include a pair of inline header pins that are not yet soldered to the board. Let's solder them in now. To line up the pins properly, you can drop them into a solderless breadboard, as shown in Figure 10-5.

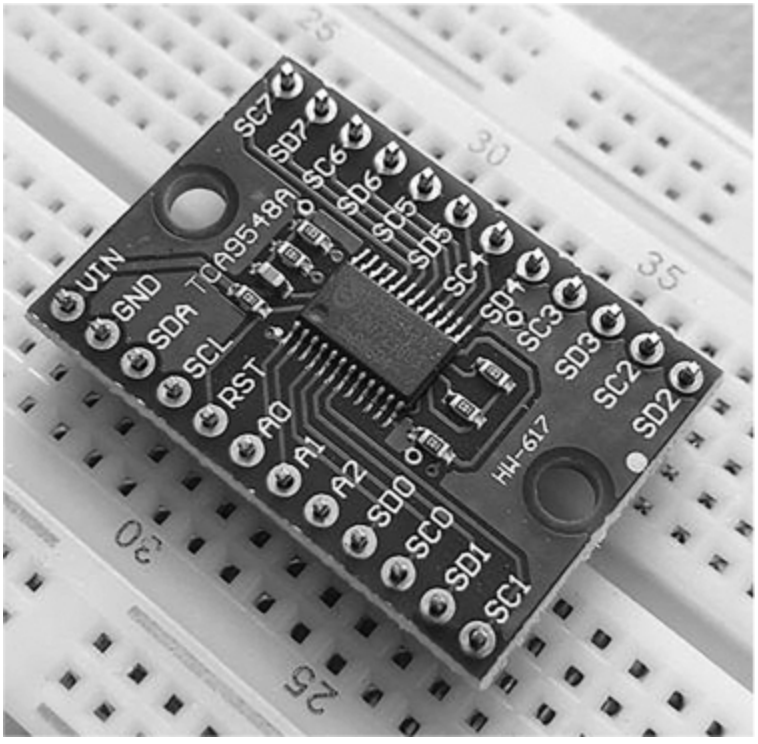


Figure 10-5: The TCA9548A breakout board with inline header pins before soldering

After you solder in the pins, your board is ready for use in solderless breadboards and other hardware solutions, as shown in Figure 10-6.

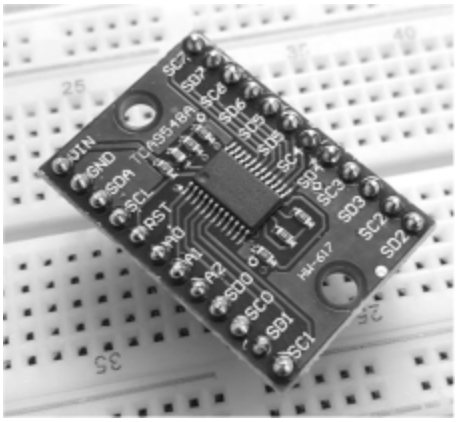


Figure 10-6: The TCA9548A breakout board with inline header pins soldered to PCB pads

The pinouts on the board match the schematic diagram shown in Figure 10-7, which also includes the connections you should make to your Arduino Uno or compatible board.

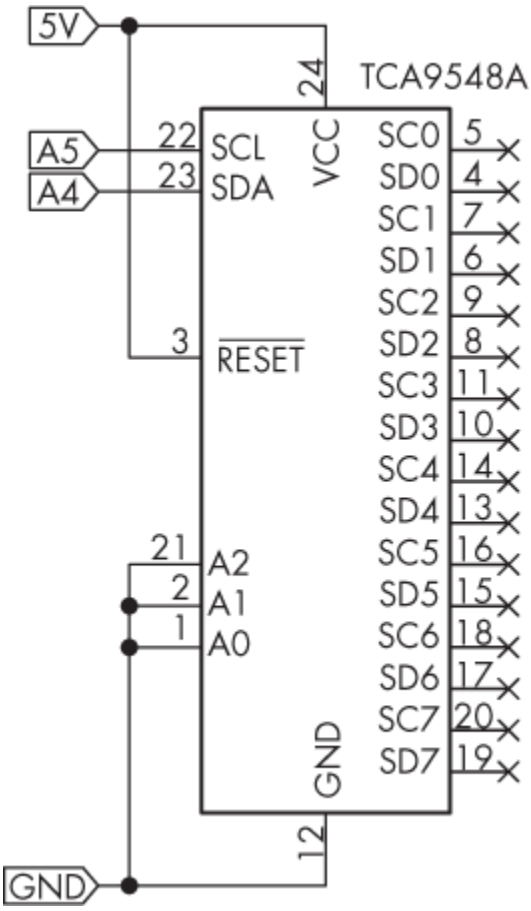


Figure 10-7: The TCA9548A schematic

The TCA9548A requires pull-up resistors between the I^2C bus and the positive power rail; these resistors are included on the breakout board.

The pins labeled A0 to A2 in the schematic are used to define the bus address for the TCA9548A. As you can see, the schematic sets them all to GND. If you run the port scanner, it will return an address of `0x70`. If necessary, you can change the address by connecting the pins A0 to A2 to combinations of GND or 5V, as listed in Table 10-1.

Table 10-1: TCA9548A Address Configuration

A0	A1	A2	Address
GND	GND	GND	0x70
GND	GND	5V	0x71
GND	5V	GND	0x72
GND	5V	5V	0x73
5V	GND	GND	0x74
5V	GND	5V	0x75
5V	5V	GND	0x76
5V	5V	5V	0x77

The eight I^2C bus pairs in the schematic are labeled as SC0/SD0 through SC7/SD7 (Clock and Data, respectively). The TCA9548A treats them each as a physically separate I^2C bus.

Using the TCA9548A in your sketch requires just one additional step before using your I^2C devices as normal: instructing the TCA9548A to use one of the eight buses it controls. To do this, you'll send a byte of data to the TCA9548A's I^2C bus register, which will determine which of the eight buses you want to use. Each bit of the byte is used to turn the bus on or off, with the most significant bit (MSB) for bus 7, and the least significant bit (LSB) for bus 0. For example, sending B00000001 (in binary) or 0 in decimal activates bus 0, while sending B00010000 activates bus 5. The TCA9548A then channels all data in and out of the bus to the Arduino on the selected bus; the TCA9548A itself is transparent to the Arduino.

Send the bus selection data only when you want to change buses. To simplify this process, you can use the following function to select the required bus:

```
void TCA9548A(uint8_t bus)
{
    Wire.beginTransmission(0x70); // TCA9548A ad
    Wire.write(1 << bus);         // Send byte t
    Wire.endTransmission();
}
```

This function accepts a bus number and places a 1 in the TCA9548A's bus register, matching our requirements. Insert this function right before you need to access a device on a particular I^2C bus. For example, you'd use this function to access a device on bus 0:

```
TCA9548A(0);
```

Here's the function to access a device on bus 7:

```
TCA9548A(7);
```

Now that you know how to select the required bus, I'll introduce a neat display that you'll use in your projects.

Graphic OLED Displays

You may be familiar with classic 16 × 2–character LCDs, as they're quite popular and inexpensive. However, they are also somewhat bulky and old fashioned. The OLED display is a sleeker, more contemporary option that can display both text and graphics in various sizes.

For the projects in this chapter, you'll use a compact 128 × 32–pixel OLED display with a diagonal display size of 0.91 inches, such as PMD Way part 35998841A, shown in Figure 10-8.

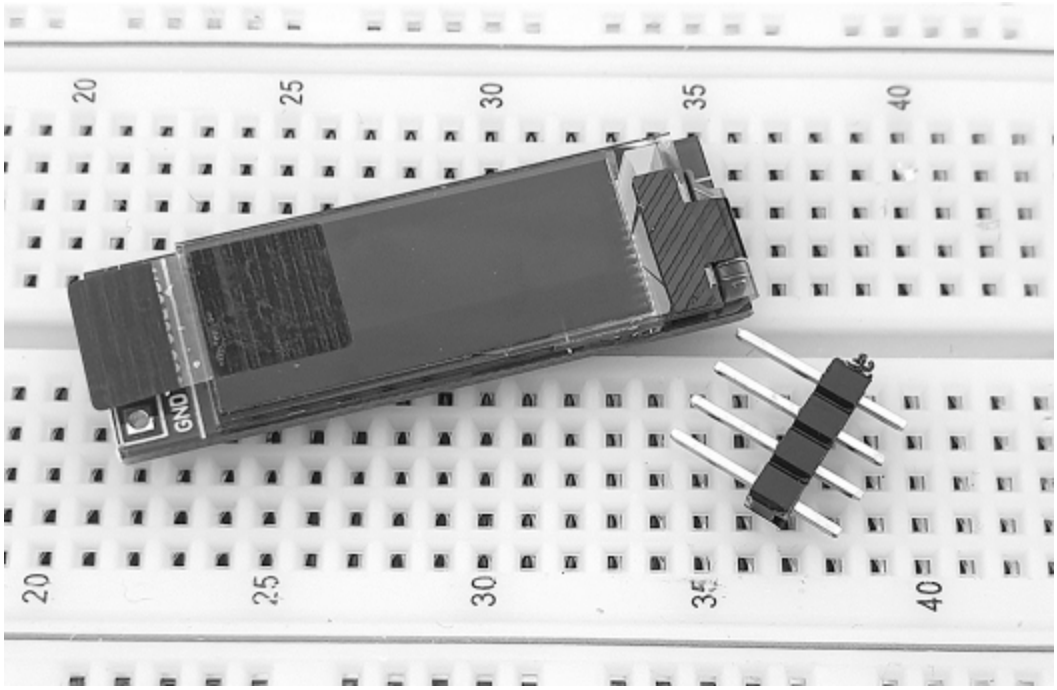


Figure 10-8: A compact OLED display with inline header pins

Inline headers for using with a solderless breadboard are included with the example board from PMD Way and are usually included with others, but check with the supplier. You can easily solder these to the breadboard by mounting the pins and the OLED in a breadboard, as shown in Figure 10-9. Supporting the other end of the OLED with something to keep the display level, such as another group of inline header pins, as shown in Figure 10-9, helps to ensure the pins are straight.

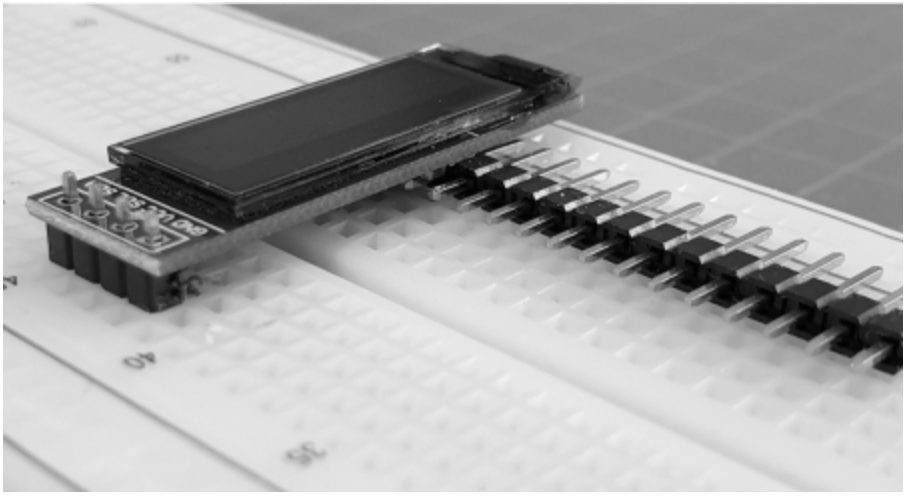


Figure 10-9: A compact OLED display sitting in inline header pins, ready for soldering

Once you're ready to use the OLED, remove the plastic screen protector by pulling the blue tag, as shown in Figure 10-10.

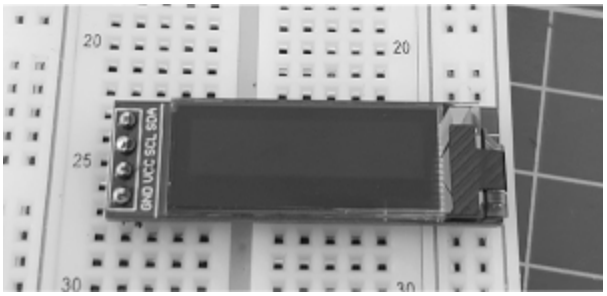


Figure 10-10: Ready-to-use OLED

To use the OLED, you'll need to install an Arduino library. Search for **u8g2** in the Library Manager, and then click **Install** at the bottom of the library description.

Once you've installed the library, test the OLED by running the I^2C scanner sketch. It should return the address 0x3C, but if you

get a different result, make a note of it for later. Next, connect the OLED to your Arduino Uno or compatible board using the connections listed in Table 10-2.

Table 10-2: OLED to Arduino Connections

OLED	Arduino Uno
SDA	A4
SCL	A5
VCC	5V
GND	GND

Now enter and upload the Listing 10-1 sketch, which you can find in the downloadable code files on this book's web page. After a few moments, the display should scroll between displaying "Count!" and an integer, as shown in Figure 10-11. If your display had a different bus address to 0x3C, insert the new address into the "address" field in the line `u8g2_2.setI2CAddress(0x3D);` in `void setup()`, then uncomment the line and save and re-upload the sketch.

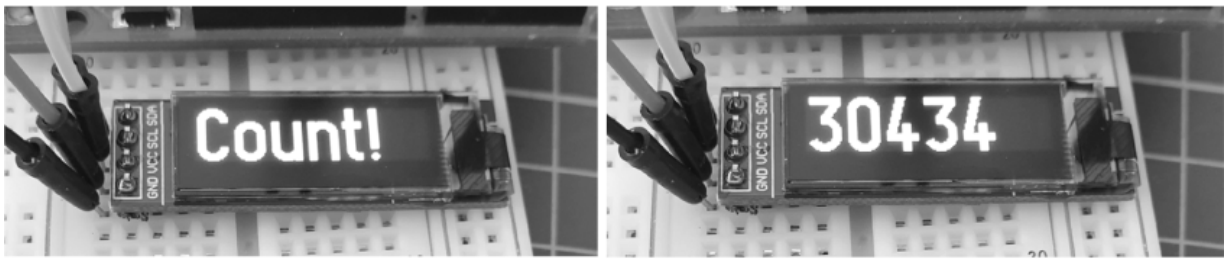


Figure 10-11: Example displays from Listing 10-1

Listing 10-1 shows how this works.

```
❶ #include <U8g2lib.h>
  #include <Wire.h>
  U8G2_SSD1306_128X32_UNIVISION_F_HW_I2C u8g2(U8G2_

  void setup()
  {
      Wire.begin();
    ❷ u8g2.begin();
      //u8g2_2.setI2CAddress(address);
    ❸ u8g2.setFont(u8g2_font_logisoso32_tr); // Select font
  }

  void loop()
  {
      for (int a = 29999; a < 32767; a++)
      {
          u8g2.clearBuffer(); // Clear OLED memory
        ❹ u8g2.drawStr(0, 32, "Count!"); // Write t
          u8g2.sendBuffer(); // Display contents o
```

```
            delay(1000);
            u8g2.clearBuffer(); // Clear OLED memory
            u8g2.setCursor(0, 32); // Position curso
            u8g2.print(a); // Write integer to memor
            u8g2.sendBuffer(); // Display contents o
            delay(1000);
        }
    }
```

Listing 10-1: Using graphic OLED displays

First, the sketch includes the u8g2 library and includes the OLED library ❶, after which it creates the instance of an OLED. The OLED's specifications are defined in this file so the library knows which type of OLED you're using.

The sketch then initializes the OLED ❷ and selects a font for the display ❸. The sketch uses a font that's 32 pixels high, which works well for this OLED, but you can replace it with any of the fonts available at *https://github.com/olikraus/u8g2/wiki/fntlistall*.

The sketch controls the OLED by drawing up the intended display in its memory and then sending the contents of the memory to the display. After the OLED's memory is cleared, the function ❹ accepts the lower x- and y-coordinates of the OLED

display and then starts drawing the text itself. The sketch then prints the contents of the OLED memory to the display.

The sketch clears the memory again after a short delay and displays a number by first setting the cursor position and then printing the number to the memory, followed by another `sendBuffer()` function to display the memory.

Now that you've mastered the OLED display, I'll introduce a sensor that you'll use along with an OLED in your final project.

BMP180 Sensors

The BMP180 sensor is an inexpensive, easy-to-use I^2C device that measures ambient temperature and air pressure. This chapter's final project uses a compact BMP180 module with pull-up resistors like the one used in Project #30, such as PMD Way part 18000001, shown in Figure 10-12.

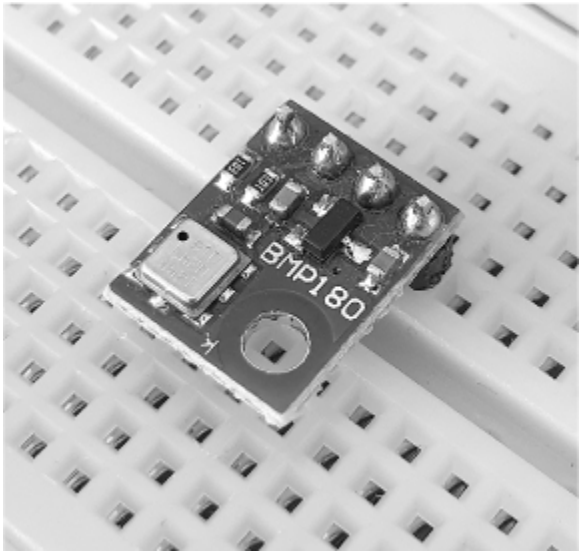


Figure 10-12: The BMP180 sensor module

As with the OLED display, begin by soldering the BMP180's header pins to the board. Next, find the **Adafruit BMP180** library in the Arduino Library Manager and install it.

During the process of installing the BMP180 library, you may be asked to install some missing dependencies. Click **Install All** if prompted to do so.

Now enter and upload the Listing 10-2 sketch. Once the upload is complete, open the Serial Monitor in the IDE, where you'll find the current temperature in degrees Celsius, along with the air pressure in hectopascals. In some regions, the unit of air pressure is the millibar. One hectopascal equals one millibar. Figure 10-13 shows an example of this output.



Figure 10-13: Example output from Listing 10-2

Listing 10-2 shows how this works.

```
❶ #include "Adafruit_BMP085.h"
  Adafruit_BMP085 bmp;

  int temperature;
  int pressure;

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

  void loop()
  {
      Serial.print("Temperature = ");
    ❸ temperature = bmp.readTemperature();
      Serial.print(temperature);
      Serial.println(" degrees C");
```

```
    Serial.print("Pressure = ");
  ❹ Serial.print(int(bmp.readPressure() / 100));
    Serial.println(" hPa");
    Serial.println();
    delay(500);
}
```

Listing 10-2: Using BMP180 temperature and air pressure sensors

To set up the BMP180, the sketch first includes the requisite library and creates an instance of the sensor ❶. Two variables, `temperature` and `pressure`, store the sensor data. This is followed by initialization of the sensor ❷.

The sketch can then retrieve the data from the sensor using the `read Temperature()` ❸ and `readPressure()` ❹ functions. The temperature and air pressure readings (with the latter divided by 100 to convert it to hectopascals) are then sent to the Serial Monitor.

Project #31: Creating a Temperature and Air Pressure Display

This project combines all the techniques you've learned in this chapter so far, using multiple I^2C buses to display the BMP180's temperature and air pressure readings with an OLED. You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- A 128 × 32–pixel, 0.91-inch OLED display
- A BMP180 temperature and air pressure sensor board
- A TCA9548A breakout board
- A solderless breadboard
- Various jumper wires

Assemble the circuit as shown in Figure 10-14.

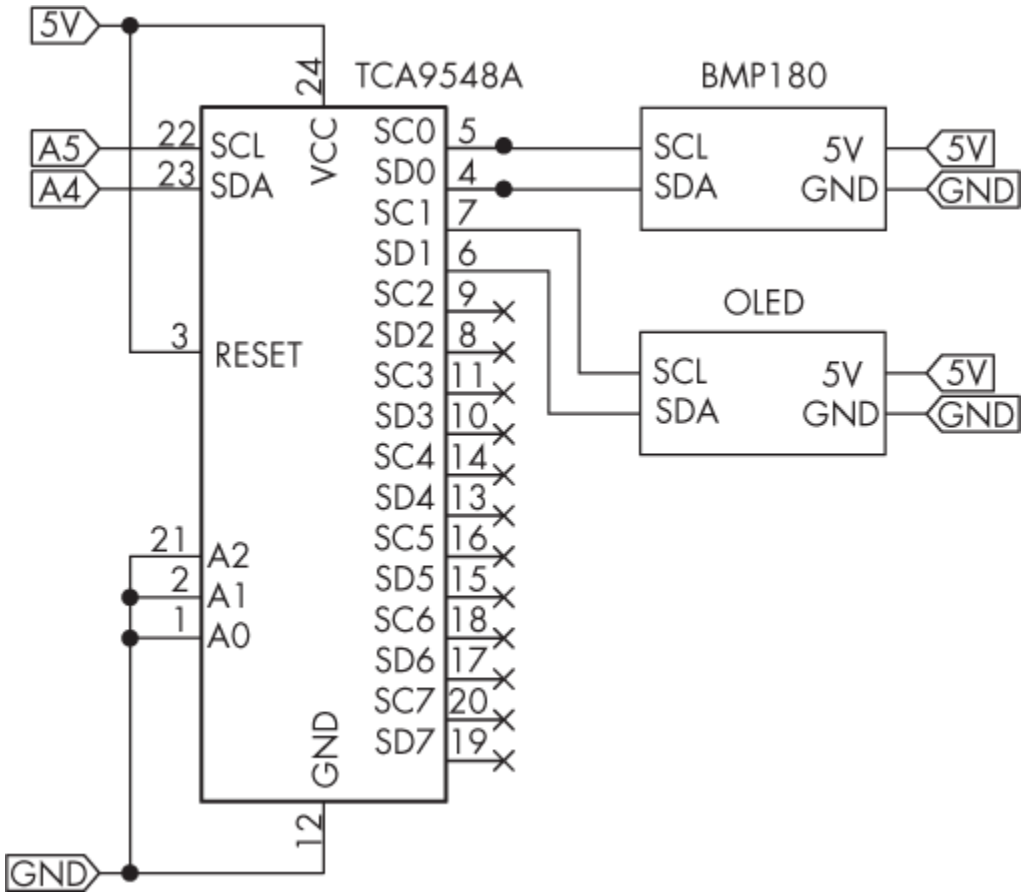


Figure 10-14: The schematic for Project #31

Next, enter and upload the Project #31 sketch. The temperature and air pressure should alternately display on the OLED, as shown in Figure 10-15.

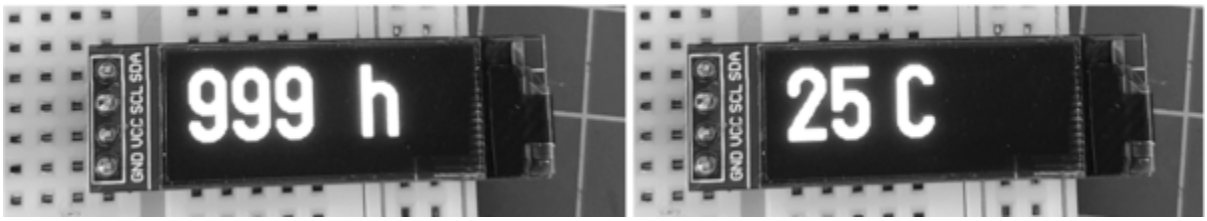


Figure 10-15: Example output from Project #31

Let’s see how this works:

```
  // Project #31 - Temperature display

❶ #include <U8g2lib.h>
  #include <Adafruit_BMP085.h>
  #include <Wire.h>

  U8G2_SSD1306_128X32_UNIVISION_F_HW_I2C u8g2(U8G2_
  Adafruit_BMP085 bmp;
  float temperature;
  float pressure;

❷ void TCA9548A(uint8_t bus)
  {
      Wire.beginTransmission(0x70); // TCA9548A ad
      Wire.write(1 << bus);         // Send byte to
      Wire.endTransmission();
  }

  void setup()
  {
   ❸ Wire.begin();
      u8g2.begin();
   ❹ u8g2.setFont(u8g2_font_logisoso32_tr); // Se
      TCA9548A(0); // Select I2C bus 0 for the BMP
   ❺ bmp.begin();
```

```
}

void loop()
{
    // First, get the temperature from the BMP18
 ❻ TCA9548A(0); // Select I2C bus 0 for the BMP
    temperature = bmp.readTemperature();
    pressure = bmp.readPressure();

    // Next, display the temperature on the OLED
 ❼ TCA9548A(1); // Select I2C bus 1 for the OLE
    u8g2.clearBuffer(); // Clear OLED memory
    u8g2.setCursor(0, 32);
    u8g2.print(int(temperature));
    u8g2.drawStr(50, 32, "C");
 ❽ u8g2.sendBuffer();
    delay(1000);

    u8g2.clearBuffer(); // Clear OLED memory
    u8g2.setCursor(0, 32);
    pressure = pressure / 100; // Convert to hec
    u8g2.print(int(pressure));
    u8g2.drawStr(80, 32, "h");
 ❾ u8g2.sendBuffer();
    delay(1000);
}
```

The sketch first includes the required libraries ❶ and configures the OLED type to use, temperature sensor, and required variables, as covered previously. It then defines a custom function to select the I²C bus from the TCA9548A ❷. It starts the I²C and OLED functions ❸, selects the OLED font ❹, and starts the temperature sensor ❺.

The sketch then switches the TCA9548A to the first of its I²C buses to retrieve data from the BMP180 ❻ and then to the second I²C bus ❼ so that the temperature and pressure data can be displayed on the OLED ❽ ❾.

Moving On

In this chapter, you learned to harness multiple devices on an I²C bus, alongside adding a new sensor and display to your tool kit. You might use these skills to make a large, multiple-time-zone clock with many OLEDs and a real-time clock IC, or you could use many temperature sensors in one area to determine the variations of temperature inside a refrigerator. No matter the project, you're no longer limited by inexpensive devices with a single I²C bus address.

In the next chapter, you'll use an Arduino Leonardo or compatible board to emulate a USB mouse and keyboard, which opens up many interesting and useful possibilities.

11

EMULATING USB MICE AND KEYBOARDS WITH THE LEONARDO



This chapter shows you how to convert information gathered by the Arduino into keypresses or mouse movements and then send them to a computer to simulate typing on a USB keyboard or to simulate manipulating an external mouse. To do so, you'll use the Arduino Leonardo development board, which has a few subtle differences from the usual Uno or compatible board.

Along with a micro-USB connector and a complete surface-mount design for all the main components, the Leonardo uses

Microchip Technology's ATmega32U4 microcontroller. This microcontroller has built-in USB communication, so the board doesn't need a USB interface chip, which allows the Leonardo to appear as a mouse or keyboard to a connected computer. You might use this keyboard emulation technique to build your own game controller, shortcut keyboard, rapid data capture and entry systems, input devices for the differently abled, and more.

In this chapter, you'll learn to:

- Emulate keypresses and typing on USB keyboards, as well as USB mouse movement and button control
- Build a USB data logger that writes directly to a spreadsheet
- Use a keypad to make a keyboard of shortcuts

USB Keyboards

To emulate a keyboard with the Leonardo (shown in Figure 11-1), you don't need any extra hardware apart from a USB cable long enough to reach between your board and the host PC. If you're ordering a longer cable, don't forget that you need a USB-A-to-micro-USB cable.

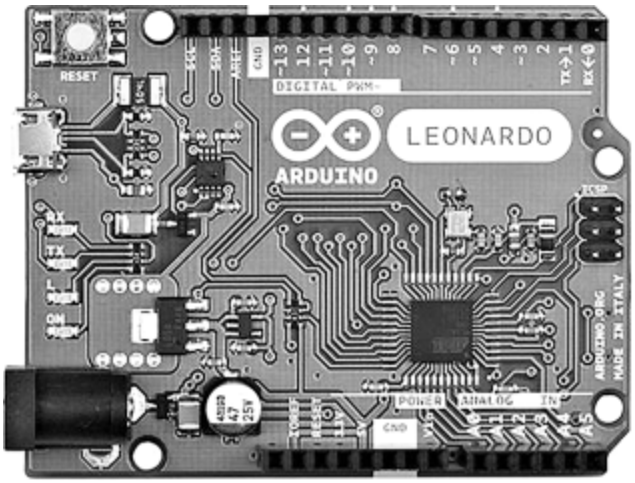


Figure 11-1: An Arduino Leonardo development board

You can activate keyboard emulation by including the Arduino Keyboard library, using the following two lines:

```
#include "Keyboard.h" // Entered at start of ske
Keyboard.begin();     // Entered in void setup()
```

To have the Leonardo send keypresses to the computer to mimic a user typing on a USB keyboard, use the following functions:

```
Keyboard.print()
Keyboard.println()
```

These functions operate in a similar manner to `Serial.print()` and `Serial.println()`: the former sends the text to be “typed

out" as keypresses, and the latter sends the same out and adds a return (or pressing RETURN or ENTER) to a new line. Using these two functions, you can send out any text that can generally be sent via your usual USB keyboard.

To stop emulating a USB keyboard, use the following function:

```
Keyboard.end() // Ends emulation
```

This function is necessary only when you expect a user to continue using the PC via the normal keyboard while the Leonardo is still connected.

Let's try this. First, connect your Arduino Leonardo to your computer. If this is the first time you've used the Leonardo, your PC may take a few moments to automatically install the required USB drivers. Next, open the Arduino IDE and change the board type to Arduino Leonardo using the drop-down menu shown in Figure 11-2 (note that your COM: port may be different from the one in the figure).

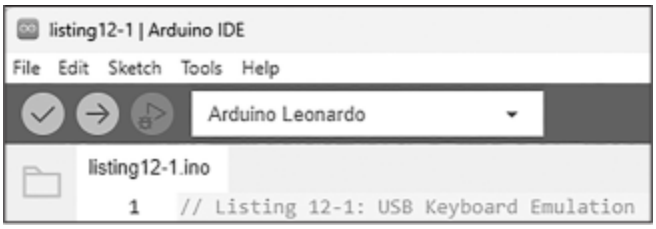


Figure 11-2: Selecting the Leonardo in the IDE

Always ensure that the board type is set to Leonardo when working with sketches that use the Keyboard library; otherwise, they will not compile if you're testing for errors.

If you have other software open on your PC, close it or ensure its windows are not active while you're experimenting with keyboard and mouse emulation. Otherwise, the "keypresses" being sent by the Leonardo may interfere with the other software. For example, before uploading the following sketch on my own machine, the only software I had open was the Arduino IDE and Notepad.

Now enter and upload the sketch for Listing 11-1; then immediately switch over to a simple text editor such as Notepad.

```
#include "Keyboard.h"
void setup()
{
    delay(5000);
```

```
    Keyboard.begin();
}

void loop()
{
  ❶ Keyboard.print("Hello, ");
  ❷ Keyboard.println("world. ");
    delay(1000);
}
```

Listing 11-1: USB keyboard emulation

Once uploaded, the Leonardo should type “Hello, world.” every second or so, as shown in Figure 11-3.

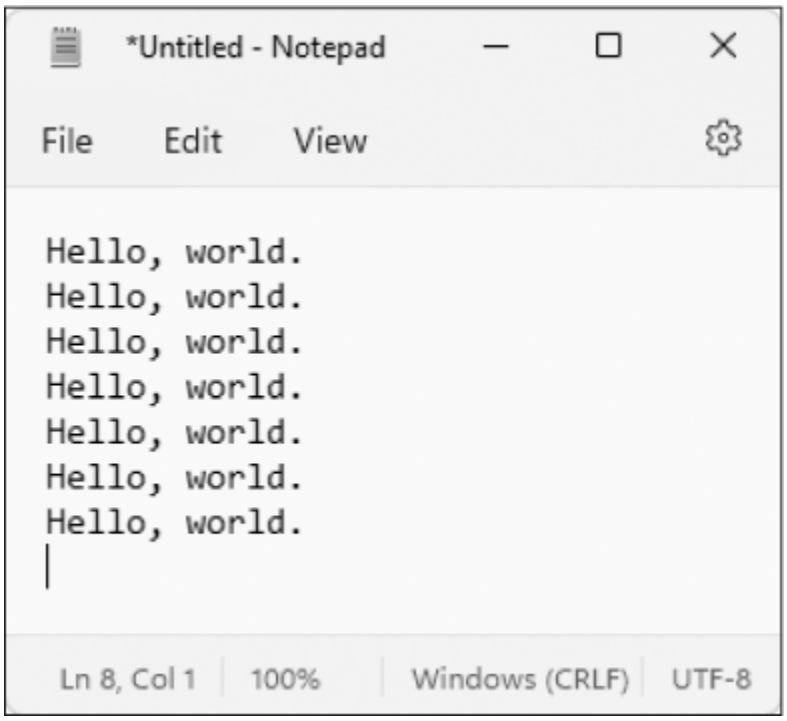


Figure 11-3: Example output from Listing 11-1

First, the sketch includes the Keyboard library. The five-second delay gives the user some time to get the PC ready for input from the Arduino. The Keyboard library is then initialized. The sketch then sends some text without a new line ❶ and then some more text along with a new line ❷.

When you've finished using the Leonardo for any USB keyboard or mouse emulation project, upload a simple sketch such as the IDE example *blink* so that your Leonardo won't try to take over the controls the next time you connect it to a PC. If you're having trouble taking control of the computer when uploading

a new sketch, you can use the CTRL-U keyboard shortcut to upload a sketch.

Emulating Keyboard Modifiers and Special Keys

In addition to creating keypresses for letters, numbers, and symbols, you can emulate buttons such as cursor keys, TAB, function keys, and so on, using a few more functions.

To send a single button press (that is, to press the button and then release it), use

```
Keyboard.write(x)
```

where *x* is the ASCII code for the letter, number, or symbol key to press, or the code for special modifier keys, listed in Table 11-1.

Table 11-1: Modifier Keys and Their Codes

Modifier key	Code	Modifier key	Code
KEY_LEFT_CTRL	128	KEY_F1	194

Modifier key	Code	Modifier key	Code
KEY_LEFT_SHIFT	129	KEY_F2	195
KEY_LEFT_ALT	130	KEY_F3	196
KEY_LEFT_GUI	131	KEY_F4	197
KEY_RIGHT_CTRL	132	KEY_F5	198
KEY_RIGHT_SHIFT	133	KEY_F6	199
KEY_RIGHT_ALT	134	KEY_F7	200
KEY_RIGHT_GUI	135	KEY_F8	201
KEY_UP_ARROW	218	KEY_F9	202
KEY_DOWN_ARROW	217	KEY_F10	203
KEY_LEFT_ARROW	216	KEY_F11	204
KEY_RIGHT_ARROW	215	KEY_F12	205

Modifier key	Code	Modifier key	Code
KEY_BACKSPACE	178	KEY_F13	240
KEY_TAB	179	KEY_F14	241
KEY_RETURN	176	KEY_F15	242
KEY_ESC	177	KEY_F16	243
KEY_INSERT	209	KEY_F17	244
KEY_DELETE	212	KEY_F18	245
KEY_PAGE_UP	211	KEY_F19	246
KEY_PAGE_DOWN	214	KEY_F20	247
KEY_HOME	210	KEY_F21	248
KEY_END	213	KEY_F22	249
KEY_CAPS_LOCK	193	KEY_F23	250

Modifier key	Code	Modifier key	Code
		KEY_F24	251

In addition to the modifiers listed in the table, you can review the complete list of ASCII codes including those for letters, numbers, and symbols, at *https://www.asciitable.com/*.

To simplify things, you can define the required codes to matching descriptions for the keys you want to emulate. For example, CAPS LOCK is 193, so add the following at the start of the sketch:

```
#define CAPS_LOCK 193
```

To press CAPS LOCK using the Leonardo, you'd use the following function:

```
Keyboard.write(CAPS_LOCK);
```

To test this, you can use the sketch in Listing 11-2, which presses CAPS LOCK every second.

```
#define CAPS_LOCK 193
#include "Keyboard.h"
void setup()
{
    Keyboard.begin();
}

void loop()
{
 ❶ Keyboard.write(CAPS_LOCK);
    delay(1000);
}
```

Listing 11-2: Blinking CAPS LOCK every second

This sketch tells the host PC that CAPS LOCK has been pressed ❶, which will be reflected on a USB keyboard connected to the PC. (If, at this point, you are considering making some mischief on a friend's computer, do so at your own risk!)

Pressing and Releasing One or More Keys

You can also emulate pressing down a button on the keyboard and then releasing it after a set period. To do this, use

```
Keyboard.press(x)
```

to hold down the button whose modifier or ASCII code is *x*. Then, to release that button, use the following:

```
Keyboard.release(x)
```

You can test this using Listing 11-3.

```
#include "Keyboard.h"
void setup()
{
Keyboard.begin();
}

void loop()
{
 ❶ Keyboard.press(122); // Hold down the "z" key
   delay(1000);
 ❷ Keyboard.release(122);
   Keyboard.println(); // Press ENTER for new li
   Delay(5000);
}
```

Listing 11-3: Testing press and release

This sketch demonstrates pressing and holding down the Z key ❶, waiting a moment, and then letting go of the key ❷. Figure 11-4 shows the results of the sketch. As you can see, the first two lines of text printed to the screen are shorter than the last two lines, since there is a slight difference in keyboard response time depending on the operating speed of the host PC.

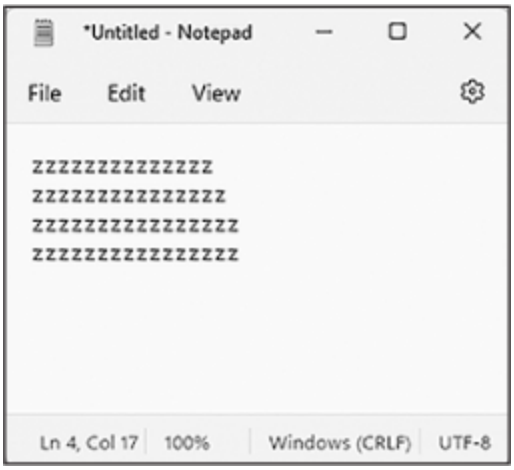


Figure 11-4: Example output from Listing 11-3

Finally, you can program your Leonardo to press several keys at once and then release them one at a time or all at once using multiple `Keyboard.press()` functions. This allows you to emulate useful multiple-keypress shortcuts used by your PC's operating system.

For example, you can sign out of a Windows PC by pressing CTRL-ALT-DELETE and then ALT-S. The following functions will press the left CTRL, left ALT, and DELETE buttons simultaneously:

```
Keyboard.press(128); // CTRL
Keyboard.press(130); // ALT
Keyboard.press(212); // DEL
```

You can then release them all by using this:

```
Keyboard.releaseAll(); // Let go of all buttons
```

You can also let go of individual buttons and keep others pressed by using `Keyboard.release(x)`, where *x* is the requisite modifier or ASCII code.

You'll put all this to work in the next project by logging data from the Leonardo directly to a PC.

Project #32: Logging Data with USB Keyboard Emulation

One exceptionally useful benefit of keyboard emulation is the ability to log data from a Leonardo directly to a PC. For this project, you'll log temperature and air pressure data from the BMP180 sensor used in Chapter 10 directly into a spreadsheet. If you can dedicate a PC for data logging—for example, in a

laboratory or your office—this is a rapid, inexpensive method of data capture.

You'll need the following parts for this project:

- An Arduino Leonardo or compatible board and USB cable
- A BMP180 temperature and air pressure sensor board
- A solderless breadboard
- Various jumper wires

Assemble the circuit as shown in Figure 11-5.

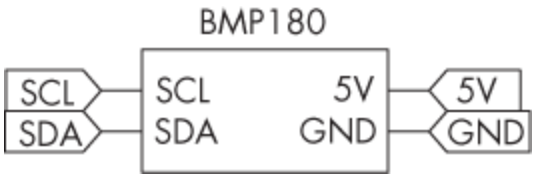


Figure 11-5: The schematic for Project #32

To get started, open your preferred spreadsheet, such as Microsoft Excel or Google Sheets. Switch back to the Arduino IDE and upload the Project #32 sketch. Then quickly switch back to the spreadsheet software, click the top-left cell, and wait. Ten seconds after you've reset the Leonardo, the date, time, temperature, and air pressure in pascals should be "typed" into the cells, as shown in Figure 11-6.

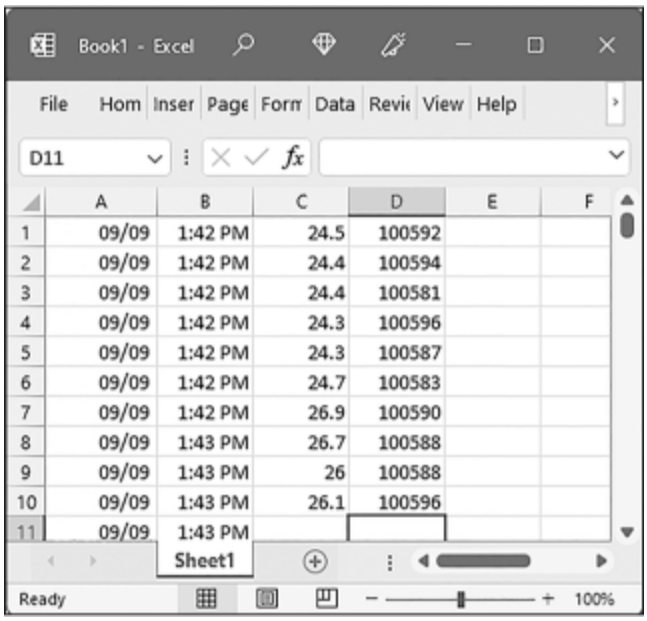


Figure 11-6: Example output data for Project #32 in an Excel spreadsheet

Once logging has started, you might disconnect the mouse and keyboard from the PC so others can't interrupt the proceedings. Furthermore, by using a cloud-based tool such as Google Sheets, you can monitor and review the results in real time from any web-enabled device in the world, as shown in Figure 11-7.

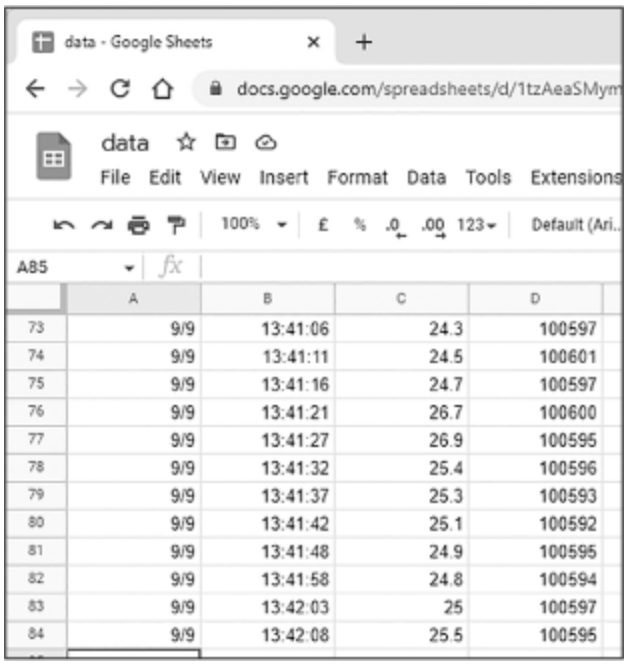


Figure 11-7: Example results in Google Sheets

Let's see how this works:

```
// Project #32 - Data logging using USB keyboard

#define Ctrl 128
#define Semicolon 59
#define Shift 129
#define Tab 179
#define Enter 176

❶ #include "Keyboard.h"
#include "Wire.h"
#include "Adafruit_BMP085.h"
Adafruit_BMP085 bmp;
```

```
❷ void spreadsheetDate()
  {
      Keyboard.press(Ctrl);
      Keyboard.press(Semicolon);
      delay(100);
      Keyboard.releaseAll(); // Let go of all butt
  }

❸ void spreadsheetTime()
  {
      Keyboard.press(Ctrl);
      Keyboard.press(Shift);
      Keyboard.press(Semicolon);
      delay(100);
      Keyboard.releaseAll(); // Let go of all butt
  }

  void pressTab()
  {
      Keyboard.write(Tab);
  }

  void pressEnter()
  {
      Keyboard.write(Enter);
  }
```

```
void setup()
{
    Keyboard.begin();
    bmp.begin();
  ❹ delay(10000);
}

void loop()
{
    spreadsheetDate();
    pressTab();
    spreadsheetTime();
    pressTab();
    Keyboard.print(bmp.readTemperature());
    pressTab();
    Keyboard.print(bmp.readPressure());
    pressEnter();
    delay(5000);
}
```

The sketch first initializes the sensor and the Keyboard library ❶. The custom function ❷ emulates pressing CTRL-+, a shortcut used in many spreadsheets to insert the date into the current cell. There is a small delay before releasing the keys. The next custom function ❸ emulates pressing CTRL-SHIFT-+, the shortcut to insert the time into the current cell. This is followed by

functions that press TAB and ENTER, for the sake of convenience, to save you using the raw commands.

The keyboard controls are based on those for a Windows-based computer. If you're using macOS or another operating system, you may need to change the required keys for moving the cursor around the screen or the shortcuts in the spreadsheet software.

The usual initialization takes place in `void setup()`, plus a long delay ❹, which gives the user enough time to navigate the cursor to the starting cell in the spreadsheet before the Leonardo starts typing. Finally, in `void loop()`, the date, time, temperature, and air pressure are "typed" into the spreadsheet using the required combination of keys, followed by a delay before the next sample is logged.

Now that you have a solid understanding of keyboard emulation, let's build another tool you can use on a regular basis: the shortcut keyboard.

Project #33: Building a USB Shortcut Keyboard

Those who work or play with a PC daily will, over time, need to type various things repeatedly, such as passwords, keyboard shortcuts, or commonly used lines of text. With an Arduino

Leonardo and the 12-button keypad described in Project #3 in Chapter 1, you can build a handy shortcut keyboard with 12 different options.

For the purposes of demonstration, the project will have a variety of examples assigned to the keypad. However, apart from useful text, you can research shortcuts for your PC's operating system or favorite software and then create your own. Table 11-2 lists a variety of keyboard and typing shortcuts.

Table 11-2: Example Shortcuts

Button	Shortcut	Required emulation presses
1	Enter PIN 456700.	456700
2	Lock the screen in Windows 11.	CTRL-ALT-DEL, then ENTER
3	Display Windows Task Manager.	CTRL-ESC-SHIFT

Button	Shortcut	Required emulation presses
4	Print to default printer.	CTRL-P, then ENTER
5	Switch between apps.	ALT-TAB
6	Save open pages as bookmarks in the Edge browser.	CTRL-SHIFT-D
7	Switch to Page Layout view in Excel.	ALT-W, then P
8	Switch to Normal Layout view in Excel.	ALT-W, then L
9	You can contact me via email at … or telephone … after 3 PM	Typed text

Button	Shortcut	Required emulation presses
10	We cannot solve your problem via social media. Please email … for technical support.	Typed text
11	Thank you for your email. I will reply in more detail once I can locate the required information.	Typed text
12	Without examining the product in person, we can’t determine the fault. Please post it to …	Typed text

You’ll need the following parts for this project:

- An Arduino Leonardo or compatible board and USB cable
- Keypad hardware as described in Project #3 in Chapter 1

Assembly is the same as in Project #3, substituting an Arduino Leonardo for an Uno, as shown in Figure 11-8. You may want to

check the ADC values reported by each keypad button by using the sketch from Project #1 and then updating the ranges in our analogkeypad library.

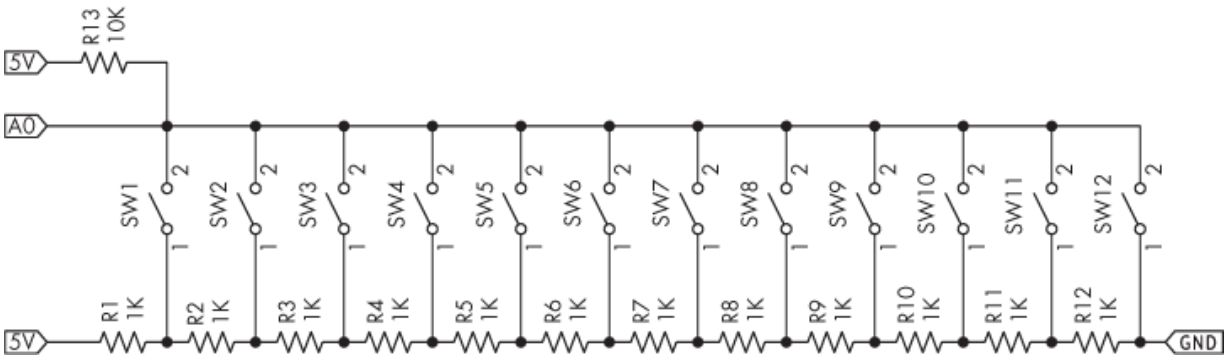


Figure 11-8: The schematic for Project #33

Now enter and upload the following sketch:

```
// Project #33 - USB shortcut keyboard

#define Ctrl 128
#define Alt 130
#define Escape 177
#define Shift 129
#define Tab 179
#define Enter 176
#define Delete 212

❶ #include "Keyboard.h"
#include "analogkeypad.h"
analogkeypad keypad;
```

```
void button1()
{
    Keyboard.print("456700");
}

void button2()
{
    Keyboard.press(Ctrl);
    Keyboard.press(Alt);
    Keyboard.press(212);
    delay(100);
    Keyboard.releaseAll(); // Let go of all butt
    delay(100);
    Keyboard.write(Enter);
    delay(100);
}

void button3()
{
    Keyboard.press(Ctrl);
    Keyboard.press(Shift);
    Keyboard.press(Escape);
    delay(100);
    Keyboard.releaseAll(); // Let go of all butt
}

void button4()
```

```
{
    Keyboard.press(Ctrl);
    Keyboard.press(112); // Press P
    delay(100);
    Keyboard.releaseAll(); // Let go of all butt
    delay(100);
    Keyboard.write(Enter);
    delay(100);
}

void button5()
{
    Keyboard.press(Alt);
    Keyboard.press(Tab);
    delay(100);
    Keyboard.releaseAll(); // Let go of all butt
}

void button6()
{
    Keyboard.press(Ctrl);
    Keyboard.press(Shift);
    Keyboard.press(100); // Press d
    delay(100);
    Keyboard.releaseAll(); // Let go of all butt
}

void button7()
```

```
{
    Keyboard.press(130);
    Keyboard.press(119); // Press w
    delay(100);
    Keyboard.releaseAll(); // Let go of all butt
    Keyboard.write(112); // Press p
}

void button8()
{
    Keyboard.press(Alt);
    Keyboard.press(119); // Press w
    delay(100);
    Keyboard.releaseAll(); // Let go of all butt
    Keyboard.write(108); // Press l
}

void button9()
{
    Keyboard.println("You can contact me via ema
or telephone 212-555-1213 after 3 p.m. ");
}

void button10()
{
    Keyboard.println("We cannot solve your probl
Please email support@company.com for technical s
}
```

```
void button11()
{
    Keyboard.println("Thank you for your email, I
once I can locate the required information. ");
}

void button12()
{
    Keyboard.println("Without examining the produ
determine the fault. Please post it to: ");
    Keyboard.println("129 West 81st Street");
    Keyboard.println("Apt 5B");
    Keyboard.println("New York NY 10024");
}

void setup()
{
  ❷ Keyboard.begin();
}

void loop()
{
  ❸ switch (keypad.readKeypad()) // Read button
    {
        case 1: button1(); break;
        case 2: button2(); break;
        case 3: button3(); break;
```

```
            case 4: button4(); break;
            case 5: button5(); break;
            case 6: button6(); break;
            case 7: button7(); break;
            case 8: button8(); break;
            case 9: button9(); break;
            case 10: button10(); break;
            case 11: button11(); break;
            case 12: button12(); break;
        }
        delay(250);
    }
```

The sketch takes advantage of the analogkeypad library for the hardware and the Keyboard library to emulate the USB keyboard. These libraries are included ❶ and then initialized ❷. The sketch then loops while it waits for a button press, which is actioned at ❸. After that, it's a simple matter of monitoring the keypad and calling one of the custom functions from `button1()` to `button12()` matched to each button. Each custom button function then performs the required emulation task, which you change for your own requirements.

Once the sketch has been uploaded, your keypad should operate as a shortcut keyboard. For example, upon receiving an

email about a faulty product, our example user can hit Reply on the email, press the 12th button, and send it. Figure 11-9 shows an example email exchange.



Figure 11-9: Example output email for Project #33

For someone answering customer support emails, this would save a lot of time. Based on your own daily computer usage habits, you can replace the shortcuts in the sketch with your own.

USB Mice

You can also activate mouse emulation with your Leonardo by including the Arduino Mouse library with the following two lines of code:

```
#include "Mouse.h" // Entered at start of sketch
Mouse.begin();    // Entered in void setup()
```

To have the Leonardo move the mouse pointer around as a USB mouse, use this function:

```
Mouse.move(x,y,z);
```

The parameter are as follows:

x The amount to move on the x-axis. Use a positive number to move to the right, a negative number to move to the left, and 0 to stay on the current x-axis.

y The amount to move on the y-axis. Use a positive number to move down, a negative number to move up, and 0 to stay on the current y-axis.

z Emulates movement of a mouse scroll wheel. Use a positive number to roll “toward you,” and a negative number to roll “away from you.” If you have changed the default mouse operation in your computer’s operating system, you will need to swap negative for positive numbers.

To end mouse emulation in a sketch, use:

```
Mouse.end();
```

When using the mouse emulation, you need a small delay of 2 ms after each mouse function to give the computer time to catch up. You can test controlling pointer movement with Listing 11-4.

```
#include <Mouse.h>
void setup()
{
    Mouse.begin();
}

void loop()
{
 ❶ Mouse.move(-10, -5, 0);
    delay(500);
}
```

Listing 11-4: Controlling pointer movement

The sketch first initializes the library. The mouse pointer then moves 10 pixels to the left and 5 up ❶. There's a short delay afterward, just to slow things down a little.

After you upload the sketch, your mouse pointer should meander toward the top left of the screen. You can always change its position with the PC mouse, too. When finished, upload a non-mouse sketch to stop the movement.

You'll now put these mouse-emulation functions to use with a simple project.

Project #34: Keeping Your PC Awake Automatically

Some PCs or other devices with USB mouse input may go into a "sleep mode" or require mouse movement on a regular basis to prove a user is present. In this project, you'll program your Leonardo to "wiggle" the mouse every 30 seconds or so to keep the PC awake or the boss happy.

The only required hardware is your Arduino Leonardo and a matching USB cable. Enter and upload the following sketch:

```
// Project #34 - PC "Keep Awake"

#include <Mouse.h>
void wiggleMouse()
{
 ❶ Mouse.move(-50, 0, 0);
   delay(500);
```

```
  ❷ Mouse.move(50, 0, 0);
    delay(500);
}

void setup()
{
        Mouse.begin();
}

void loop()
{
    wiggleMouse();
    delay(30000);
}
```

The mouse pointer should move to the left ❶, wait a moment, move back to the right ❷, and then wait another moment, as instructed by the functions in the `wiggleMouse()` custom function. The delays in the function and in the main loop are purely arbitrary; alter them to suit your requirements.

USB Mouse Buttons

To program your Leonardo to “click” (press and then release) the emulated mouse buttons, you can use

```
Mouse.click(); // Clicks the left mouse button b
```

or

```
Mouse.click(x);
```

where *x* can be `MOUSE_LEFT`, `MOUSE_RIGHT`, or `MOUSE_MIDDLE` for the respective buttons.

To press and hold down a mouse button, use the following function:

```
Mouse.press(); // Presses and holds down the lef
```

To release a mouse button being pressed, use the following function:

```
Mouse.release(); // Defaults to releasing the le
```

You can also use the same parameters as `Mouse.click()` for the middle and right buttons. Note that the definition of what is left and right with regard to the mouse buttons is defined by the

settings in your operating system. For example, if you've swapped the buttons (so "right" is the physical left button), you will need to take that into account.

Let's demonstrate mouse button control by creating a fun drawing project.

Project #35: Creating a PC Random Painter

This project will randomly move the mouse pointer and randomly CTRL-click the left mouse button. By running this sketch in conjunction with a PC painting program, you can create some avant-garde Arduino-generated works of art.

Again, the only hardware required is your Arduino Leonardo and matching USB cable. Enter, but don't upload, the Project #35 sketch. Now open a PC painting program, such as Microsoft Paint, then switch over to the Arduino IDE and upload the sketch. Switch back to the paint software and place the cursor in the painting area after selecting a paintbrush and color. The sketch should start drawing based on random mouse movements, creating a painting like the one in Figure 11-10.

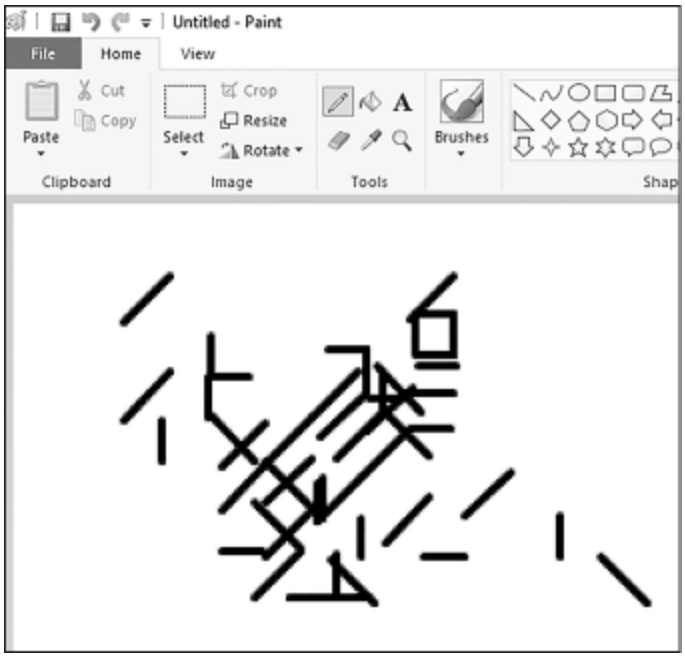


Figure 11-10: Example results from the random PC painter

Let's see how this works:

```
// Project #35 - PC Random Painter

#include <Mouse.h>
void setup()
{
    Mouse.begin();
    randomSeed(analogRead(0));
  ❶ delay(10000);
}

void loop()
{
```

```
❷ Mouse.press();
  delay(2);
  switch (random(8))
❸ {
      case 0: Mouse.move(-10, 0, 0); break;
      case 1: Mouse.move(10, 0, 0); break;
      case 2: Mouse.move(0, -10, 0); break;
      case 3: Mouse.move(0, 10, 0); break;
      case 4: Mouse.move(10, 10, 0); break;
      case 5: Mouse.move(10, -10, 0); break;
      case 6: Mouse.move(-10, 10, 0); break;
      case 7: Mouse.move(-10, 10, 0); break;
  }
  delay(1000);
❹ Mouse.release();
  delay(2);
  switch (random(8))
❺ {
      case 0: Mouse.move(-10, 0, 0); break;
      case 1: Mouse.move(10, 0, 0); break;
      case 2: Mouse.move(0, -10, 0); break;
      case 3: Mouse.move(0, 10, 0); break;
      case 4: Mouse.move(10, 10, 0); break;
      case 5: Mouse.move(10, -10, 0); break;
      case 6: Mouse.move(-10, 10, 0); break;
      case 7: Mouse.move(-10, 10, 0); break;
  }
```

```
    delay(1000);
}
```

To create this masterpiece, the sketch first presses the mouse button ❷, makes one of eight random mouse movements ❸, and then releases the mouse button ❹. It then makes another random mouse movement ❺ before repeating the process. The large delay ❶ gives you time to move from the Arduino IDE to the paint software after uploading the sketch.

For a final challenge, you might add a start button to your project so the Arduino doesn't take over the keyboard or the mouse until you're ready for it to do so. You might also try making an Etch A Sketch simulator, using two potentiometers for x- and y-axis control of the mouse.

Moving On

At this point, you have the skills and practice you need to implement USB keyboard and mouse emulation for your own projects. You learned how to instruct the Arduino Leonardo or compatible board to behave as a keyboard and also behave as a mouse. Whenever you do so, make sure that there is a sufficient delay at the start of your project to enable any required

preparation on the PC. This also gives you time to easily upload a new sketch to the board if you're finished with the emulation experiments. If you completely lose control of your board, you can upload a new sketch using USBasp hardware via the ICSP pins.

In the next chapter, you'll learn to read and write data to and from USB flash drives with your Arduino.

12

TRANSFERRING DATA TO AND FROM USB FLASH DRIVES



USB flash drives are a convenient tool to move data to and from your Arduino projects. This chapter shows you how to transfer data to and from these forms of portable storage, using your Arduino and an inexpensive interface module.

You can use these methods of recording and retrieving data to create more user-friendly ways of interfacing information with your projects. For example, you could change the movements of an Arduino-controlled robot by having it read control values from the USB flash drive, or you could

change the times for an automated sprinkler controller without needing to have a physical user interface.

In this chapter, you'll learn to:

- Install and test USB flash drive modules
- Write and append data from your Arduino to a USB flash drive
- Log sensor data to a USB flash drive
- Read numbers and text from a USB flash drive into your Arduino
- Build a project USB security key
- Configure Arduino projects with external data from a USB flash drive

Selecting and Preparing a USB Flash Drive

There are many brands and types of USB flash drives on the market. Because of the idiosyncrasies of the interface module, some units may or may not work at all. After some testing, I've found that Lexar- and Toshiba-branded flash drives have been successful. Ensure that they are advertised as USB 2.0, not USB 3.0.

USB flash drives should ship preformatted. However, if you find that they don’t work reliably, you may need to reformat them yourself. In this case, use the FAT32 filesystem with a 16KB allocation unit size.

Whenever it’s time to remove the USB flash drive from your PC, always use the safe removal option provided by your computer’s operating system. Don’t just pull the flash drive out when you think the PC is finished with the drive, as this will often make the drive unreadable to the USB interface module.

The USB Interface Module

To work with USB flash drives, you’ll need an interface module based around the CH376S USB file manager and control IC—for example, PMD Way part 566592, shown in Figure 12-1.

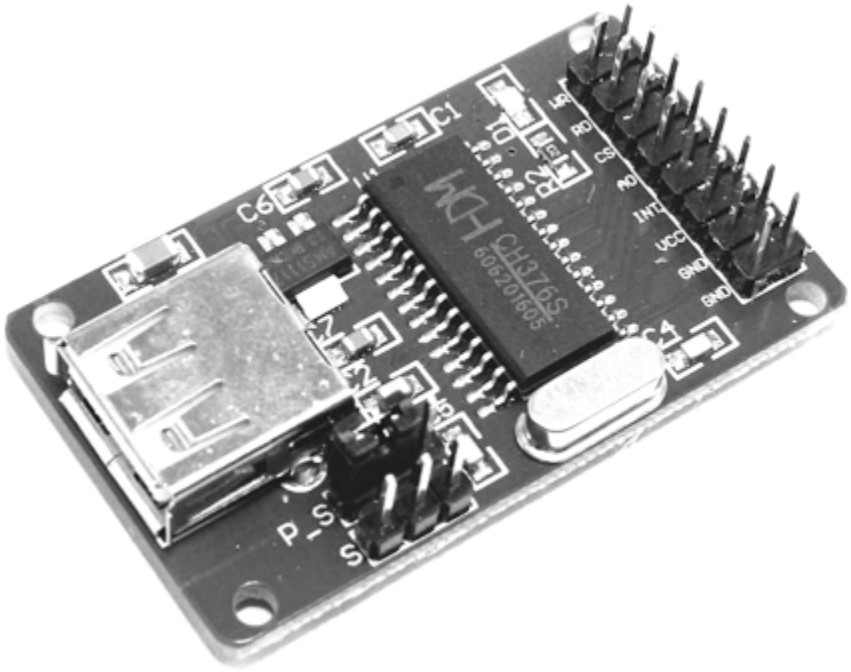


Figure 12-1: A USB flash drive interface module

You can use many possible interfaces to make connections between the module and a microcontroller. For the purposes of this chapter, we'll use a 9,600 bps serial UART connection between the module and a software serial port on the Arduino.

Since the module setup for all the projects in this chapter is the same, let's begin by testing your module with the required configuration. Locate the header pin jumper in the 2 × 3 group of header pins below the USB socket and then place the jumper across the second and third pins on the top row, as shown in Figure 12-2. This sets the unit into serial interface mode. If your module has a 1 × 3 set of jumper pins, you may need to try both

options (jumpering 1 and 2 or 2 and 3, if the pins aren't labeled).

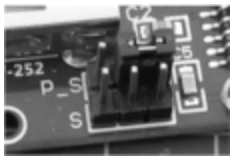


Figure 12-2: Setting the jumper on the USB flash drive interface module

Next, make the connections shown in Figure 12-3 using some male-to-female jumper wires between the module and your Arduino Uno or compatible board. The bottom row of three pins below the jumper you just placed are referred to as S1, S2, and S3.

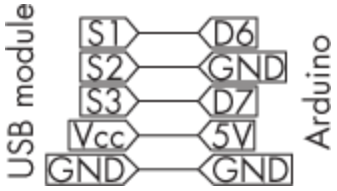


Figure 12-3: The schematic for the connections between USB module and Arduino

The module requires you to install an Arduino library. Search for **CH376** in the Library Manager and then click **Install** at the bottom of the library description.

Once you've installed the library, connect the Arduino with the interface module to your PC and upload the sketch

basicUsageSoftSerial, which is included with the CH376 library examples, as shown in Figure 12-4.

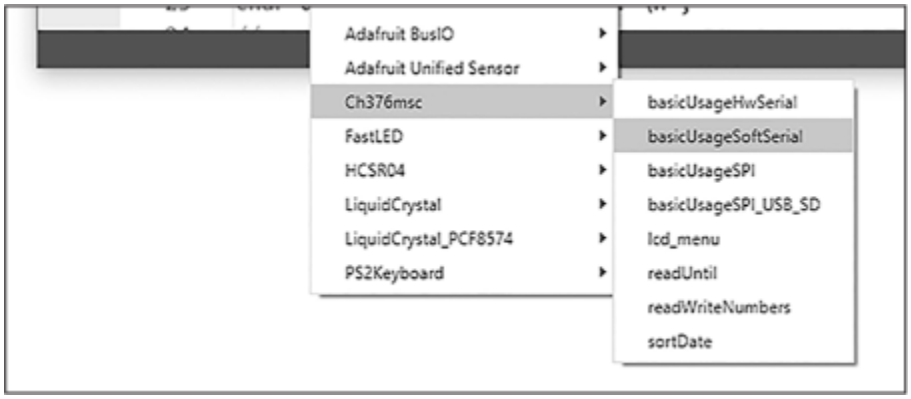


Figure 12-4: Locating the CH376 example sketch in the Arduino IDE

The purpose of running this sketch is to test the USB usage before building it into your own projects. Now open the Serial Monitor, change the speed to 115,200 bps, and reset your Arduino. The sketch should display a series of options and show the presence of a flash drive, as shown in Figure 12-5.



Figure 12-5: The output from the CH376 demonstration sketch

If you don't see the message `Flash drive attached!` in the monitor, gently remove and reinsert the USB flash drive into the module. A single LED on the module should also light up when a USB flash drive is inserted. If this is still not successful, try reformatting the flash drive as described in the previous section. Failing that, try a different flash drive.

If the Arduino reports the flash drive is attached, run the various operations described in the menu displayed on the Serial Monitor. First, send a 1 in the Serial Monitor to create a text file in the flash drive, then a 2 to append data to that file (which opens the original file and adds more text to it), and then a 7 to show the contents of the flash drive, as shown in Figure 12-6, for example.

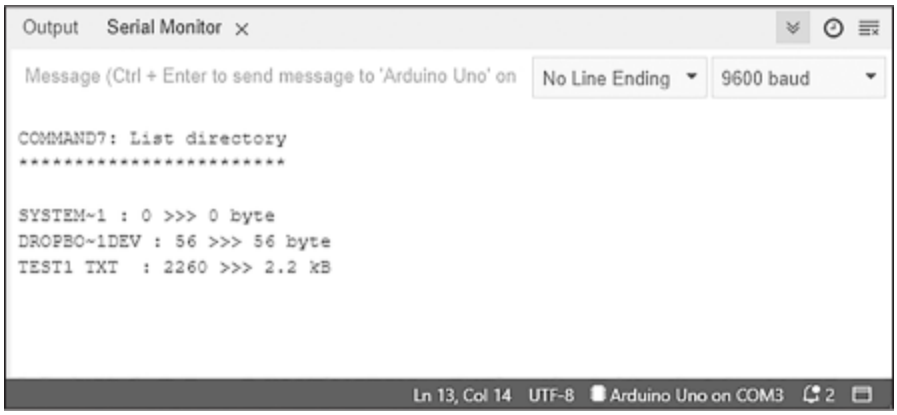


Figure 12-6: The results of flash drive operations

The sketch creates the file and calls it *TEST1.TXT*. If you have files with filenames that are longer than the classic 8 + 3 file format, they will be truncated and have a tilde inserted into the filename, as has been done with the first two files shown in Figure 12-6.

Once you've created and appended the file *TEST1.TXT*, you can be satisfied that your hardware and USB flash drive combination are operating correctly. It's time to move on to writing data to text files on the flash drive.

Writing Data

In this section, I'll show you how to initialize and write various kinds of numerical and text data to a text file on the flash drive and then demonstrate the process with an example sketch. This technique is perfect for data capture and logging applications.

First, I'll show you the required functions and how they work, which are then demonstrated in the following projects.

No matter what type of text data you want to write, you'll first need the following four lines at the start of the sketch to initialize the interface module and the required software serial port:

```
#include <Ch376msc.h>
#include <SoftwareSerial.h>
SoftwareSerial USBserial (7, 6);
Ch376msc USBdrive(Serial1);
```

This includes the libraries for the USB module (CH376) and the software serial port, then initializes the software serial port, and finally creates an instance of the USB module over software serial.

Next, you'll add some lines to `void setup()`:

```
Serial.begin(115200); // For Serial Monitor
USBserial.begin(9600); // For USB module
USBdrive.init();
```

This starts serial communications between the Serial Monitor and the USB module and starts the USB module communication. You must place the three functions in the order shown.

In the `void loop()` section of the sketch, use the following code to check the status of the USB module. In addition to checking for the presence of the USB flash drive, it's also handy for troubleshooting; if the flash drive isn't connected or working, this code will stop the sketch and notify you via the Serial Monitor:

```
if (USBdrive.checkIntMessage())
{
    if (USBdrive.getDeviceStatus())
    {
        Serial.println(F("Flash drive attached!"
    } else {
        Serial.println(F("Flash drive detached!"
    }
}
```

The text in `Serial.println()` is enclosed with `F()` to save the Arduino's SRAM (think of this as the memory used for operations, like RAM in your computer) and instead use the

flash memory (where sketches are stored in the microcontroller).

Next, set the filename and open the file:

```
USBdrive.setFileName("TEST1.TXT");
USBdrive.openFile();
```

Filenames must be in the classic 8 + 3 format. Use the *.txt* file extension if you intend to write plaintext, or use *.csv* if you intend to open the file in a spreadsheet. If you use the same filename and write a second or multiple times to that file, the data in the file will be overwritten from the start.

Next, write some data. To write a byte, integer, or double or long variable, use

```
USBdrive.writeNum(x)
USBdrive.writeNumln(x)
```

where *x* is the name of the variable you want to write. The second function writes a new line after the number.

To write a character of text or send a command in the form of a character (such as a new line `\n`), use this:

```
USBdrive.writeChar('x');
```

This will write the character *x* to the file.

To write a line of text, you must first store it in a character array no longer than 254 characters:

```
char text1[] = "This is a line of text. \n";
USBdrive.writeFile(text1, strlen(text1));
```

This writes the text in the array `text1` to the opened file, and a newline character. The `writeFile()` function accepts the character array and then the length of the character array as parameters.

To write floating-point variables, you must convert the number to a character array and then write the array. First create an array of characters in which to place the number:

```
char floatChar[8] = " ";
```

Ensure the array is long enough to contain all digits, the decimal point, and the negative sign, if necessary. Then assign the floating-point variable some value:

```
float f = 123.456;
```

Now use the following to convert the float to the character array:

```
dtostrf(a,b,c,d);
```

Here, *a* is the float to convert, *b* is the number of digits in the whole number, *c* is the number of digits in the fraction, and *d* is the array to place the resulting number. For example, to convert `f` into `floatChar`, use this:

```
dtostrf(f,3,3,floatChar);
```

You can now write the character array with the following (as you did with the line of text):

```
USBdrive.writeFile(floatChar, strlen(floatChar))
```

Finally, when you've finished writing the data to the text file, close the file:

```
USBdrive.closeFile();
```

At this point, you can remove the flash drive from the module and insert it into a PC to retrieve the text file.

You can test these functions with the following sketch. After you've connected the interface module to your Arduino and inserted the USB flash drive, enter and upload Listing 12-1.

```
char text1[] = "This is a line of text in the fo
array, followed by a new line function. \n"; ❶

char text2[] = "This is another line of text. I

char floatChar[8] = " ";
float f = 12.34567;

#include <Ch376msc.h> ❷
#include <SoftwareSerial.h>
SoftwareSerial USBserial(7, 6);
Ch376msc USBdrive(USBserial);

void fileWrite()
{
    Serial.println("fileWrite()");
    USBdrive.setFileName("TEST1.TXT"); // Set th
    USBdrive.openFile();               // Open t
```

```
    // Write some integers:
    for (int a = 0; a < 20; a++)
    {
        USBdrive.writeNum(a); ❹
        USBdrive.writeChar(',');
    }

    // Write new line:
    USBdrive.writeChar('\n'); ❺
    // Write lines of text:
    USBdrive.writeFile(text1, strlen(text1)); ❻
    USBdrive.writeFile(text2, strlen(text2));
    USBdrive.writeChar('\n');

    // Write floating-point number - first conve
    dtostrf(f,2,5,floatChar); // 2 whole digits,
    USBdrive.writeFile(floatChar, strlen(floatCha
    USBdrive.closeFile(); // Close the file ❽
    Serial.println("Finished.");
}

void setup()
{
    Serial.begin(115200); // For Serial Monitor
    USBserial .begin(9600); // For USB module
    USBdrive.init();
    delay(5000);
}
```

```
void loop()
{
    if (USBdrive.checkIntMessage()) ❿
    {
        if (USBdrive.getDeviceStatus())
        {
            Serial.println("Flash drive attached
        } else {
            Serial.println("Flash drive detached
        }
    }
    fileWrite();
    delay(5000);
}
```

Listing 12-1: Test writing to the USB flash drive

The sketch declares the required variables for text use and the float for demonstration ❶. It initializes the required libraries ❷ and then names and opens the file ❸. It then writes integers to the file ❹, along with characters (commas to separate the numbers being written). The program writes a newline command to the file ❺, followed by writing text ❻, converting a float to a character array, and writing it to the file ❼.

Once the file is closed ❽, the flash drive can be removed. The serial communication and USB drive start at ❾. The sketch then inserts the code for USB flash drive detection ❿ and finally calls the function to write the data by calling `fileWrite()`.

You can monitor the progress of the writing and remove the flash drive once `Finished` is displayed in the Serial Monitor. Check the results by viewing the text file on a PC, as shown in Figure 12-7.

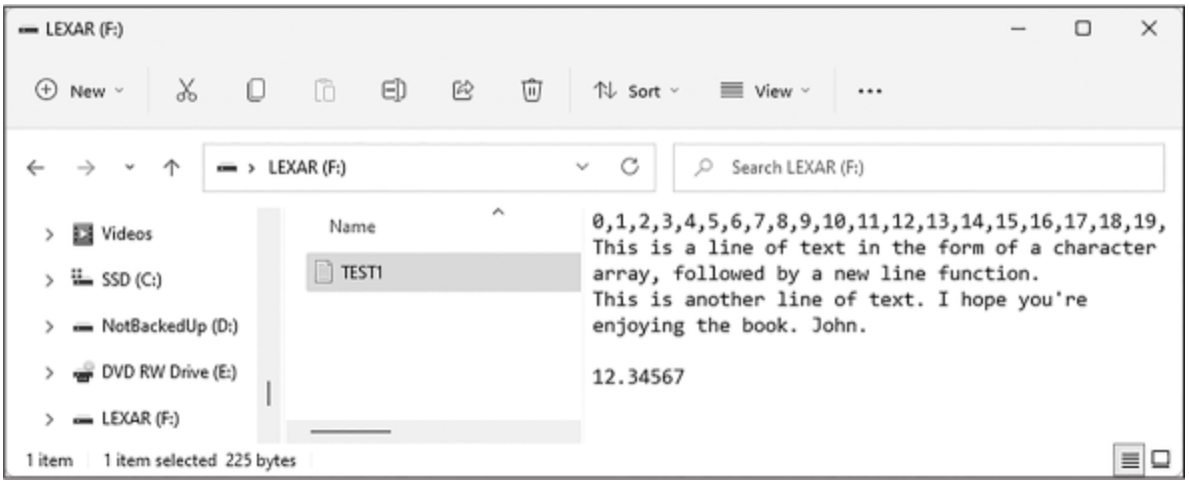


Figure 12-7: The results of Listing 12-1

All this allows you to write data in a once-off burst. However, if you want to add data to a file over time, you need to append the data instead.

Appending Data to a File

Writing numerical and text data to a text file stored on the flash drive is perfect for once-only types of situations. If you need to add data to a file, however, you can't just write to it as described in the previous section. Instead, after writing to a file the first time, you must append data to that file to keep all the data stored safely. In this section, I'll explain the required functions and then demonstrate them in the following projects.

The code for appending data to a file is like the code for writing, with some minor differences after opening the file. I'll explain what needs to be done and then demonstrate operation with a project. First, you check that the file exists, and if it does, you move the cursor (the position where the USB interface will start writing) to the end of the file with the following commands:

```
if (USBdrive.openFile() == ANSW_USB_INT_SUCCESS)
{
    USBdrive.moveCursor(CURSOREND);
}
```

You can then append data to the file with the previously described writing functions, if there's free space on the flash drive:

```
if (USBdrive.getFreeSectors())
{
  ❶ // Write data to flash drive here
} else
{
    Serial.println("Disk full");
}
```

Insert the required code to write your data to the file ❶; then close the file with `USBdrive.closeFile()` as normal.

Let's test appending data now with Listing 12-2, which will append a random number to the *TEST1.TXT* file generated by Listing 12-1.

```
#include <Ch376msc.h>
#include <SoftwareSerial.h>

SoftwareSerial USBserial (7, 6);
Ch376msc USBdrive(USBserial);

void fileAppend()
{
    Serial.println("fileAppend()");
    USBdrive.setFileName("TEST1.TXT"); // Set th
  ❶ if (USBdrive.openFile() == ANSW_USB_INT_SUCC
```

```
    {
        USBdrive.moveCursor(CURSOREND);
    }
  ❷ if (USBdrive.getFreeSectors())
    {
      ❸ USBdrive.writeNumln(random(1000));
    } else
    {
        Serial.println("Disk full");
    }
    USBdrive.closeFile();
    Serial.println("Finished");
    delay(1000);
}

void setup()
{
    Serial.begin(115200); // For Serial Monitor
    USBserial .begin(9600); // For USB module
    USBdrive.init();
    delay(5000);
  ❹ randomSeed(analogRead(0));
}

void loop()
{
    if (USBdrive.checkIntMessage())
    {
```

```
        if (USBdrive.getDeviceStatus())
        {
            Serial.println(F("Flash drive attach
        } else {
            Serial.println(F("Flash drive detach
        }
    }
    fileAppend();
  ❺ do {} while (1);
}
```

Listing 12-2: Appending a random number to the .txt *file*

After initializing the required libraries and proceeding through `void setup()`, the sketch calls the custom function `fileAppend()`, at which point the sketch continues only if the file to append exists ❶. If it does, the cursor (or point where new text is appended to the file) is moved to the end of the file.

A check for free space occurs ❷. If there's free space, the sketch writes a random number to a new line in the text file ❸. It then closes the file as usual. As the project uses random numbers, the sketch seeds the random number generator ❹. A "do nothing forever" line ❺ prevents data from being endlessly appended.

As with Listing 12-1, you can monitor the progress of the writing and remove the flash drive once the Serial Monitor displays `Finished`. Check the results by viewing the text file on a PC. For example, Figure 12-8 shows the results after the sketch has run six times.

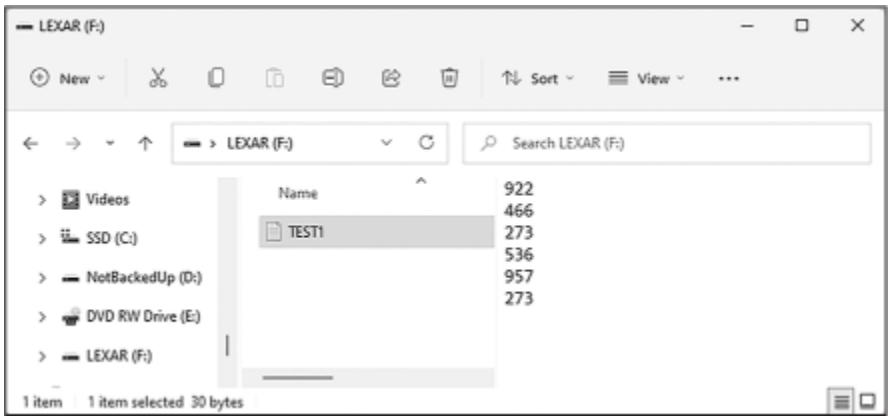


Figure 12-8: The results of Listing 12-2

In the next project, you'll use the framework you've just learned to append data from a temperature sensor to a text file.

Project #36: Logging Sensor Data

This simple data-logging demonstration records temperature, air pressure, and altitude as detected by the BMP180 sensor demonstrated in Chapter 10. The Project #36 sketch captures the data from the sensor and writes it to the USB flash drive over fixed intervals, which you can alter for your own purposes.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A USB flash drive interface module
- A BMP180 sensor module with inline pins fitted
- A solderless breadboard
- Various jumper wires

For this project, you'll use the breadboard as a bridge to help create more 5V and GND connections for the two modules to the Arduino. If you haven't previously used the BMP180 sensor, please refer to the section "BMP180 Sensors" in Chapter 10. Assemble the circuit as shown in Figure 12-9.

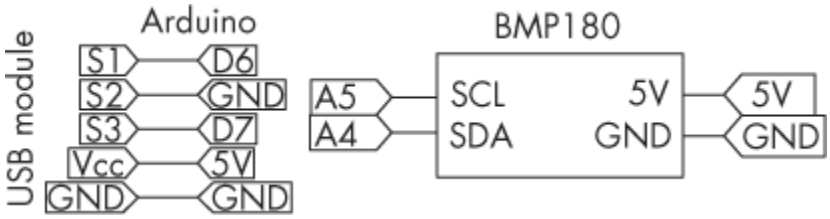


Figure 12-9: The schematic for Project #36

Enter and upload the Project #36 sketch. Once the sketch has run for a few minutes, remove the USB flash drive and review the file using the PC, as shown in Figure 12-10.

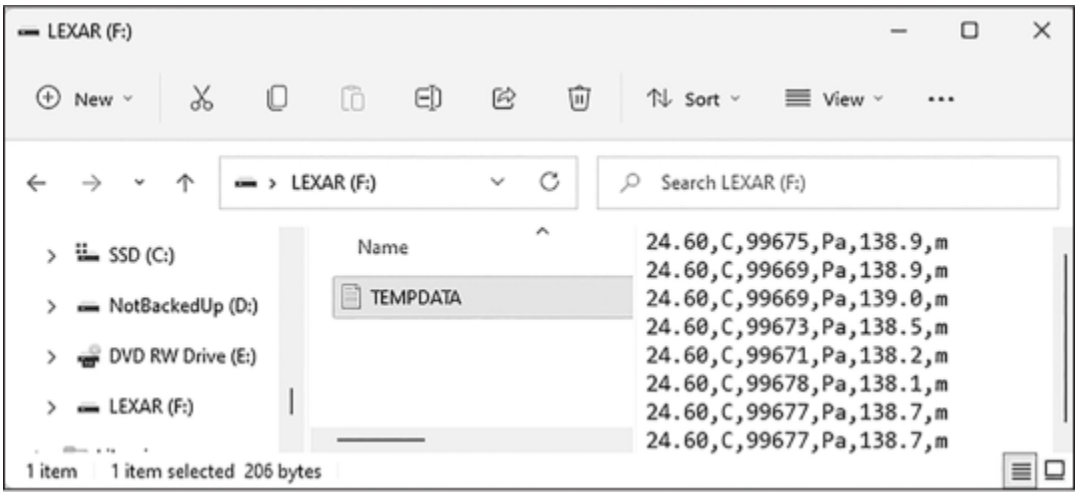


Figure 12-10: Example data captured by Project #36

Thanks to using the comma separator as a delimiter between data and descriptions, you can import the file into a spreadsheet such as Microsoft Excel, which allows for easy analysis or distribution of the captured data in a popular format, as shown in Figure 12-11.

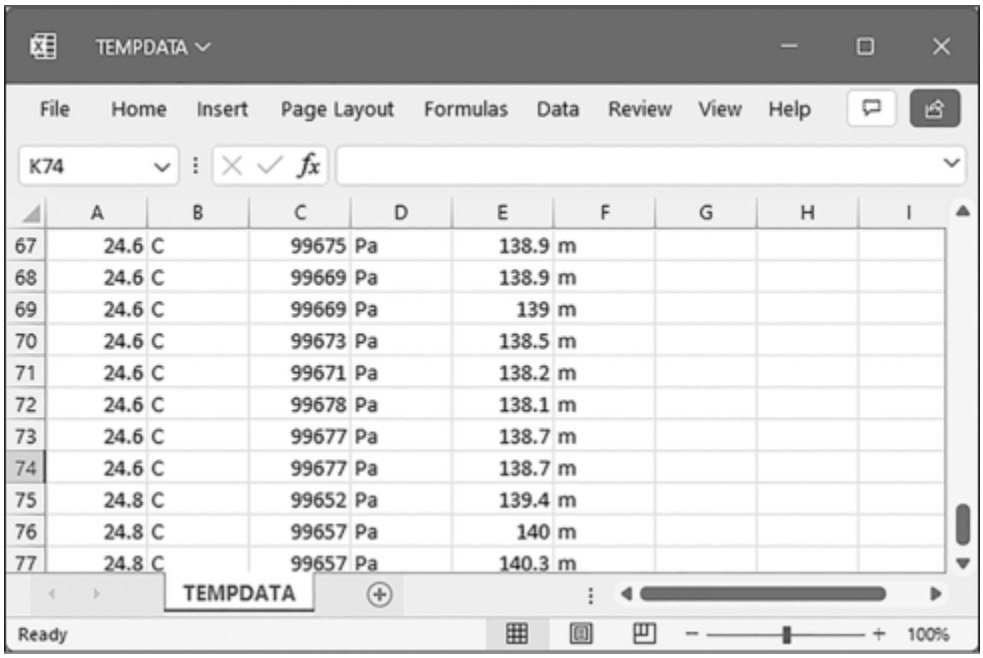


Figure 12-11: Example data captured by Project #36 in Excel

Let's see how this works:

```
  // Project #36 - Log sensor data to USB flash dr

❶ #include <Wire.h>
  #include <Adafruit_BMP085.h>
  #include <Ch376msc.h>
  #include <SoftwareSerial.h>

  Adafruit_BMP085 bmp;
  SoftwareSerial USBserial (7, 6);
  Ch376msc USBdrive(USBserial);

❷ char floatChar[8] = " ";
```

```
void fileAppend()
{
    Serial.println("fileAppend()");
    USBdrive.setFileName("TEMPDATA.TXT"); // Set
    if (USBdrive.openFile() == ANSW_USB_INT_SUCC
    {
        USBdrive.moveCursor(CURSOREND);
    }
    if (USBdrive.getFreeSectors())
    {
     ❸ dtostrf(bmp.readTemperature(),2,2,floatCh
       USBdrive.writeFile(floatChar, strlen(flo
       USBdrive.writeChar(',');
       USBdrive.writeChar('C');
       USBdrive.writeChar(',');

     ❹ USBdrive.writeNum(bmp.readPressure());
       USBdrive.writeChar(',');
       USBdrive.writeChar('P');
       USBdrive.writeChar('a');
       USBdrive.writeChar(',');

     ❺ dtostrf(bmp.readAltitude(),5,1,floatChar)
       USBdrive.writeFile(floatChar, strlen(flo
       USBdrive.writeChar(',');
       USBdrive.writeChar('m');
       USBdrive.writeChar('\n');
```

```
    } else
    {
        Serial.println("Disk full");
    }
    USBdrive.closeFile();
    Serial.println("Finished");
}

void setup()
{
    bmp.begin();
    Serial.begin(115200);  // For Serial Monitor
    USBserial.begin(9600); // For USB module
    USBdrive.init();
    delay(5000);
}

void loop()
{
    if (USBdrive.checkIntMessage())
    {
        if (USBdrive.getDeviceStatus()) {
        Serial.println(F("Flash drive attached!"
    }
    else
    {
        Serial.println(F("Flash drive detached!"
        return;
```

```
      }
  }

  fileAppend();
❻ delay(5000); // Arbitrary delay
      9
```

This practical extension of the Listing 12-2 sketch first includes and sets up the required libraries for the BMP180, USB module, and software serial, starting at ❶. The sketch then declares a character array ❷ used to convert floating-point numbers to characters for writing to the flash drive. The temperature in degrees Celsius, air pressure in pascals, and altitude in meters are taken from the BMP180 and written to the flash drive with some simple separators at ❸, ❹, and ❺, respectively. You can alter the arbitrary delay ❻ as needed.

You can use this project to log any data that can be captured or generated by your Arduino to a USB flash drive. For a challenge, you could add a real-time clock to log the time and date along with the weather data.

Reading Numerical Data

Reading data from a USB flash drive is a useful way to import settings, parameters, and other information into your Arduino project. In this section, I'll show you how to read numerical data from your USB; you'll learn how to read text data a little later.

Your Arduino can interrogate a file stored on the flash drive that contains purely integers and read them one at a time from start to finish. Each integer must be stored on its own line in the file and be followed by a new line character. You can create the text file in a Notepad-style text editor, being sure to use an 8 + 3 filename like *FILENAME.TXT*.

The sketch initialization is the same as when you are writing to or appending data to the USB flash drive. You'll first need the following four lines at the start of the sketch to initialize the interface module and the required software serial port:

```
#include <Ch376msc.h>
#include <SoftwareSerial.h>
SoftwareSerial Serial1(7, 6);
Ch376msc USBdrive(Serial1);
```

Next, set the name of the file you want to open and open the file, using the following two functions:

```
USBdrive.setFileName("FILENAME.TXT");
USBdrive.openFile();
```

You can then read each number in turn, from the start to the end of the file, using the following code:

```
while(!USBdrive.getEOF())
{
    number = USBdrive.readLong();
}
```

The integer in this example will be stored in the variable `number`.

Finally, close the file:

```
USBdrive.closeFile();
```

Let's demonstrate this with a sketch that writes 10 integers to a file on the flash drive and then retrieves and displays those numbers. You'll need the USB flash drive module connected to the Arduino, along with the flash drive, as usual.

Upload the Listing 12-3 sketch.

```
❶ #include <Ch376msc.h>
  #include <SoftwareSerial.h>
  SoftwareSerial Serial1(7, 6);
  Ch376msc USBdrive(Serial1);

  void writeIntegers()
  {
      int number;
      if (USBdrive.driveReady())
      {
          USBdrive.setFileName("NUMBERS.TXT");
          USBdrive.openFile();
          Serial.println(F("Writing integers:"));
          for (int a = 0; a < 10; a++)
          {
              number = random(-1000, 1000);
              Serial.print(number); Serial.print("
              USBdrive.writeNumln(number);
          }
          USBdrive.closeFile();
          Serial.println(F("Finished writing intege
          Serial.println();
      }
  }

  void readIntegers()
  {
    ❷ USBdrive.setFileName("NUMBERS.TXT");
```

```
    USBdrive.openFile();
    Serial.println(F("Reading integer numbers…")

 ❸ while(!USBdrive.getEOF())
    {
     ❹ Serial.print(USBdrive.readLong());
       Serial.print(" ");
    }
 ❺ USBdrive.closeFile();
    Serial.println(F("Finished reading integers.
    Serial.println();
}

void setup()
{
 ❻ randomSeed(analogRead(0));
    Serial.begin(115200);
    Serial1.begin(9600);
    USBdrive.init();
    delay(5000);
}

void loop()
{
 ❼ if (USBdrive.checkIntMessage())
    {
        if (USBdrive.getDeviceStatus()) {
            Serial.println(F("Flash drive attach
```

```
      } else {
              Serial.println(F("Flash drive detach
              return;
          }
      }
    ❽ writeIntegers();
      delay(2000);
    ❾ readIntegers();
      delay(30000);
  }
```

Listing 12-3: Writing and displaying integers on the USB flash drive

After a short delay, the sketch should display in the Serial Monitor the numbers written to the flash drive, then read them back from the flash drive and display the same numbers in the Serial Monitor once again, as shown in Figure 12-12.

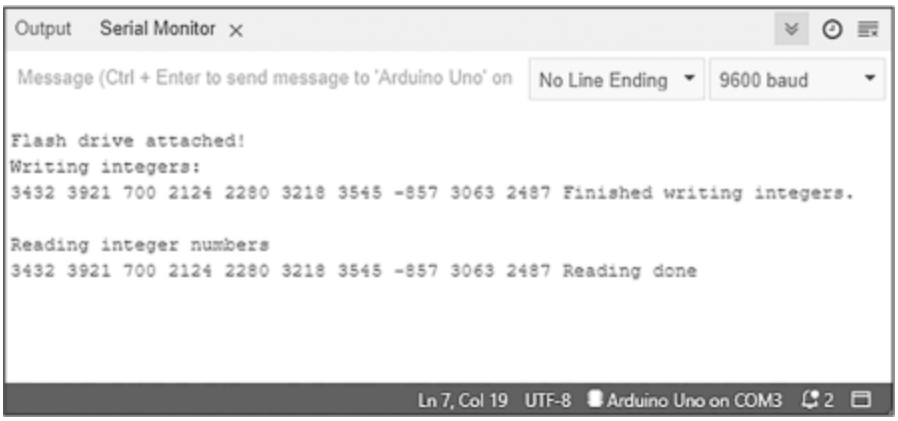


Figure 12-12: Example output from Listing 12-3

The sketch first includes and sets up the required libraries for the USB module and software serial ❶. The custom `writeIntegers()` function writes some integers to the USB flash drive for your demonstration, and the `USBdrive.setFileName()` function ❷ sets the name of the file to be stored. After setting the name of the file to open and opening the file, the sketch loops from the start of the file ❸ until the end of the file, reads each integer ❹, and displays them on the Serial Monitor.

The sketch then closes the file ❺, seeds the random number generator ❻, and starts the serial ports and USB module. It checks for the presence of a USB flash drive in the module ❼ and then runs the writing and reading demonstration functions at ❽ and ❾, respectively.

You're ready to put your knowledge of writing and reading integers to work in the following two projects by creating a USB security key and then loading parameters into your Arduino projects.

Project #37: Using a USB Security Key

This project shows you how to lock or unlock the ability for your Arduino to operate by forcing it to read a secret number text file stored on the USB flash drive before operating the main code. If the number in the text file matches what's loaded in the sketch, the code can operate; if not, the Arduino comes to a halt and will not operate. Nor will it operate without a USB flash drive.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A USB flash drive interface module
- Various jumper wires

Connect the USB flash drive module to the Arduino as described in Project #36 and then create a text file using your PC called *SECURITY.TXT* whose contents contain just the number 12345 on one line. Copy this file to your USB flash drive.

Next, insert the flash drive into the USB module, then enter and upload the following sketch.

```
// Project #37 - USB security key

#include <Ch376msc.h>
#include <SoftwareSerial.h>
SoftwareSerial Serial1(7, 6);
Ch376msc USBdrive(Serial1);

boolean readKey()
{
    boolean keyStatus=false;
 ❶ int securityKey=12345;
    int testKey=0;
 ❷ USBdrive.setFileName("SECURITY.TXT");
    USBdrive.openFile();
    while(!USBdrive.getEOF())
    {
     ❸ testKey=USBdrive.readLong();
    }
    USBdrive.closeFile();
    if (securityKey==testKey)
    {
     ❹ keyStatus=true;
    }
 ❺ return keyStatus;
```

```
}

void setup()
{
    pinMode(13, OUTPUT);
    Serial.begin(115200);
    Serial1.begin(9600);
    USBdrive.init();
    delay(5000);

  ❻ if (USBdrive.checkIntMessage())
    {
        if (USBdrive.getDeviceStatus())
        {
            Serial.println(F("Flash drive attach
        } else {
            Serial.println(F("Flash drive detach
            return;
        }
    }
  ❼ if (readKey()==false)
    {
        Serial.println("Incorrect key!");
     ❽ do {} while (1); // Halt
  }
}

void loop()
```

```
{
  ❾ digitalWrite(13, HIGH);
    delay(500);
    digitalWrite(13, LOW);
    delay(500);
}
```

In essence, this sketch opens a text file, reads the five-digit number, and compares it with a number stored in the sketch. If they match, it executes the code in `void loop()`, but if not, the sketch stops dead in `void setup()`.

The custom `boolean readKey()` function checks the security file, which has the security key number stored ❶. The sketch opens the file on the USB flash drive ❷ and then reads the key number from the file ❸. If the Arduino key number and the file key number match when the sketch compares the two, the function's value is set to `true` ❹. Otherwise, the function will return `false` by default. Either result is returned at ❺.

Checking the key begins in `void setup()`. The sketch first checks the existence of the USB flash drive ❻ and then the file key number ❼. If there isn't a match or a USB flash drive hasn't been inserted, the sketch stops ❽ and the Arduino must be reset

to try again. These results will be displayed in the Serial Monitor. In Figure 12-13, for example, the monitor tells me the flash drive is attached, but I have the incorrect key.

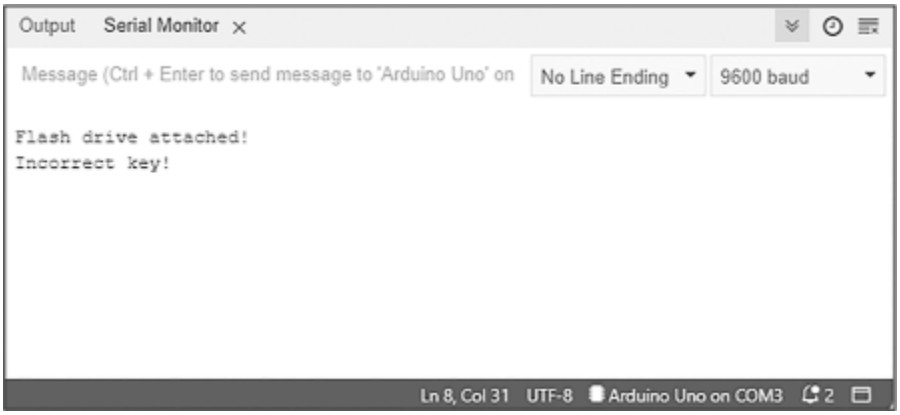


Figure 12-13: Example output from Project #37 when security keys don't match

If the key does match, however, the sketch can continue to `void loop()` and operate any code as usual ❾. In this case, it simply blinks the Arduino's onboard LED.

You can use this sketch to add a security key to your own Arduino-based projects. In the following project, you'll use similar hardware to import larger amounts of data into an Arduino, from which decisions can be made.

Project #38: Configuring a Project via USB Flash Drive

You can also use USB flash drive functionality to import data or parameters from the USB flash drive into the Arduino. For example, you might import preset instructions for a robot, set start and finish times for a timer-controlled project, or add on/off patterns for lights controlled by your project. You can do so by reading files on your USB, saving those values to your Arduino's internal EEPROM, and retrieving and acting on those values. This is useful for configuration purposes for more permanent projects.

To demonstrate this process, this project shows you how to read integers between the value of 0 and 255 from a file on the USB flash drive, store the values to the EEPROM, and then retrieve the values and display them in binary on eight LEDs. Once you've finished, you can use the functions as a framework for your own projects.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A USB flash drive interface module
- Various jumper wires
- Eight 1 kΩ, 0.25 W, 1 percent resistors
- Eight LEDs
- A solderless breadboard

Assemble the circuit as shown in Figure 12-14.

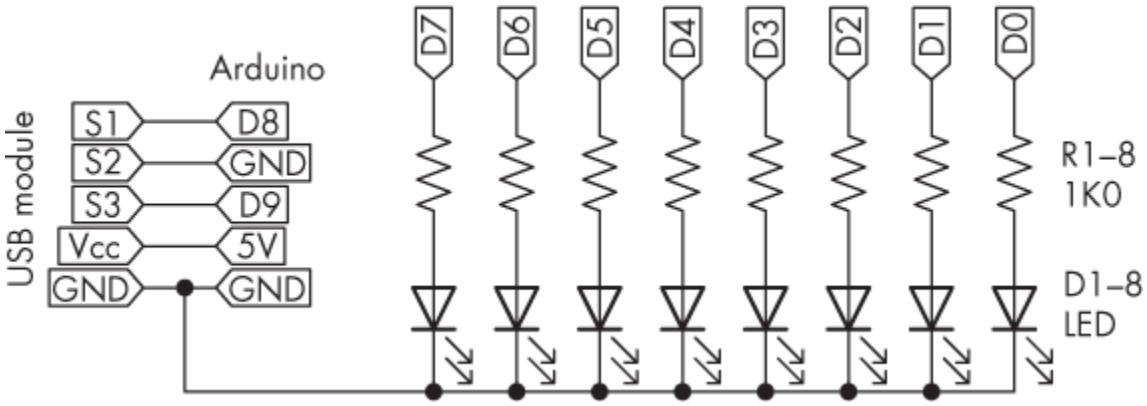


Figure 12-14: The schematic for Project #38

Next, create a text file on your USB flash drive called *SETTINGS.TXT* and place a number between 0 and 255 inclusive on each of the 100 lines. These are the values you'll copy from the flash drive to the EEPROM. (You can review writing and installing your own Arduino library in Chapter 18 of *Arduino Workshop*, 2nd edition.)

Next, insert the flash drive into the USB module, enter and upload the Project #38 sketch, and open the Serial Monitor. The LEDs should display the binary equivalent of each number stored in the EEPROM. (Values 0 through 3 may not display properly, as the LEDs are sharing the digital pins used for the USB serial connection to the PC.)

Let's see how this works:

```
// Project #38 - Project configuration via USB f

#include <EEPROM.h>
#include <Ch376msc.h>
#include <SoftwareSerial.h>
SoftwareSerial Serial1(9, 8);
Ch376msc USBdrive(Serial1);

void loadData()
{
    int _data;
    int pointer = 0;

  ❶ USBdrive.setFileName("SETTINGS.TXT");
    USBdrive.openFile();
    Serial.println(F("Reading data from USB:"));

    while(!USBdrive.getEOF())
    {
      ❷ data = USBdrive.readLong();
        EEPROM.write(_data,pointer);
        Serial.print(pointer); Serial.print(" ")
        pointer++;
    }
  ❸ USBdrive.closeFile();
    Serial.println(F("Finished."));
    Serial.println();
}
```

```
void useData()
{
    int a;
    int _data;
    Serial.println("Reading from EEPROM…");
    for (a = 0; a < 100; a++)
    {
      ❹ _data = EEPROM.read(a);
        PORTD = _data;
        Serial.print(a); Serial.print(" "); Seri
        delay(100);
    }
}

void setup()
{
    DDRD = B11111111;
    Serial.begin(115200);
    Serial1.begin(9600);
    USBdrive.init();
    delay(5000);
}

void loop()
{
    if (USBdrive.checkIntMessage())
    {
```

```
        if (USBdrive.getDeviceStatus()) {
            Serial.println(F("Flash drive attach
        } else
        {
            Serial.println(F("Flash drive detach
            return;
        }
    }
    loadData();
    delay(1000);

    useData();
    delay(1000);

    do {} while (1); // Halt

}
```

The sketch first includes and sets up the required libraries for the USB module, EEPROM, and software serial lines. The custom function `loadData()` retrieves data from the *SETTINGS.TXT* file in the USB flash drive and stores it in the EEPROM. The sketch opens the file on the flash drive ❶ and then reads each integer from the file ❷ and stores it in the EEPROM. The `pointer` variable tracks the EEPROM position, which is increased by 1 after every read. After the sketch has read the file, it closes it ❸.

The custom `useData()` function reads each value in turn from the EEPROM and stores it in the `PORTD` register to control the LEDs. (For a review of port manipulation, please revisit Chapter 2.) The `for` loop cycles, reading the value from each EEPROM position in turn ❹ and then sending that value to the `PORTD` register.

The sketch sets the usual configurations in `void setup()` and checks for a flash drive in `void loop()`. It then calls the custom functions `loadData()` and `useData()`. The sketch runs only once, thanks to the "do nothing" line.

When this project is operating, you'll see the data copied from the flash drive to the EEPROM, and then you'll see the same data displayed from the EEPROM. I've inserted a few `Serial.print()` lines into the `loadData()` and `useData()` functions to enable monitoring of the process; Figure 12-15 shows an example of this.



Figure 12-15: Example output for Project #38

However, you can remove the `Serial.print()` lines if you don't need them.

This simple example demonstrates how to read numerical data from the USB flash drive and store it away in the Arduino. It's up to you to get your own projects the data they need from the EEPROM.

Now that you've learned to read numerical data, I'll show you how to read text.

Reading Text

Your Arduino can also read text from simple files (the same type of text files as used for the integers) that contain only text and newline characters. The library will detect the newline character, but any other formatting codes created by word processor software will simply be displayed as characters from the ASCII table.

Again, it's best to create such files using a Notepad-style text editor, using an 8 + 3 filename. Reading text is quite similar to reading integers; first set the file you want to open, then open it:

```
USBdrive.setFileName("TESTTEXT.TXT");
USBdrive.openFile();
```

Next, read each character of text in turn, from the start to the end of the file, using

```
moreText = USBdrive.readFile(_buffer, sizeof(_bu
```

where `moreText` is a Boolean variable (true or false) and `_buffer` is a character array between 2 and 255 characters in length where the text will be placed. You need to repeat the previous statement until the variable `moreText` is returned as false (which signifies the end of the file being read).

Finally, close the file:

```
USBdrive.closeFile();
```

Let's demonstrate this with a sketch that displays the contents of a text file called *TESTTEXT.TXT* stored on the USB flash drive. Connect the USB module as shown in Figure 12-3, insert your USB flash drive into the module, and enter and upload the Listing 12-4 sketch:

```
#include <Ch376msc.h>
#include <SoftwareSerial.h>
SoftwareSerial Serial1(7, 6);
Ch376msc USBdrive(Serial1);

void readText()
{
    char _buffer[254];
    boolean moreText;

 ❶ USBdrive.setFileName("TESTTEXT.TXT");
    USBdrive.openFile();
 ❷ moreText = true;
    while(moreText)
    {
     ❸ moreText = USBdrive.readFile(_buffer, siz
     ❹ Serial.print(_buffer);
    }
 ❺ USBdrive.closeFile();
}

void setup()
{
    Serial.begin(115200);
    Serial1.begin(9600);
    USBdrive.init();
    delay(5000);
```

```
}

void loop()
{
    if (USBdrive.checkIntMessage())
    {
        if (USBdrive.getDeviceStatus())
        {
            Serial.println(F("Flash drive attach
        } else
        {
            Serial.println(F("Flash drive detach
        }
    }
  ❻ readText();
    delay(5000);
}
```

Listing 12-4: Reading text from the flash drive

Open the Serial Monitor, which should display the contents of the text file, as shown in Figure 12-16.

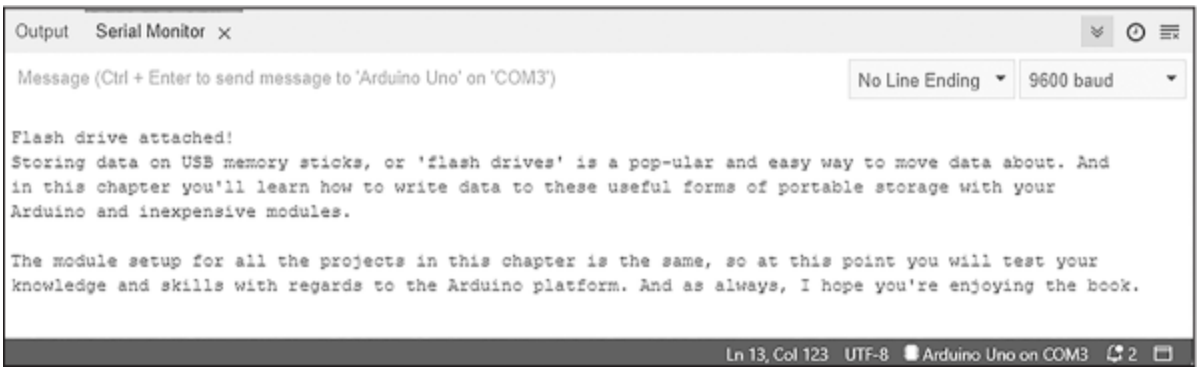


Figure 12-16: Example output from Listing 12-4

The custom `readtext()` function takes care of opening the text file, reading the contents, and using the text. The required variables are the character array `_buffer[254]`, which stores the data from the USB flash drive 254 characters at a time, and the Boolean variable `moreText`, which monitors the reading status. After the function sets and opens the file to use ❶, it sets the Boolean variable `moreText` to `true` ❷ to enable the following `while` loop to start. It reads 254 characters from the file and places them into the array `_buffer[254]` ❸. If there's more text to read, the function returns `true` into `moreText`, enabling the `while` loop to repeat again. The function then sends the text to the Serial Monitor ❹. When the read reaches the end of the file, `moreText` is set to `false` and the file is closed ❺. Finally, the sketch calls this entire `readtext()` function ❻.

You now have the basic framework to read text from a USB flash drive and work with it in your own projects. As

demonstrated with reading integers in the previous section, you could use load parameters as characters or words to configure your projects or use the array of characters collected to be displayed using other methods. If you use a smaller buffer array, perhaps 10 characters long, this will enable you to read command words for the Arduino to recognize and act upon.

Moving On

In this chapter, you learned how to write data to a USB flash drive and retrieve and use data from a USB flash drive with your Arduino. This is an easy-to-use method of transferring data to and from your Arduino.

In the next chapter, you'll expand your knowledge of input methods by learning how to interface a PS/2 PC keyboard with your Arduino.

13

INTERFACING WITH PS/2 KEYBOARDS



Some Arduino projects, such as motor controllers, lighting systems, or even games, need to accept text or numerical data via a familiar, user-friendly form of input. This chapter shows you how to accept such data with a standard PS/2-style PC keyboard and then display it with an I^2C-interface LCD.

You'll learn to:

- Receive data from a Personal System/2 (PS/2) keyboard through your Arduino
- Set up I^2C-interface LCDs for convenient data display

- Capture text from a PS/2 keyboard and display on the LCD

You'll also build an RGB LED tester with keyboard input, as well as a distraction-free text file entry device.

The PS/2 Keyboard

This chapter uses a keyboard with a PS/2-style interface, first introduced by IBM in 1987 along with its new PS/2 series of personal computers. This type of interface was prevalent until around 1997, when the USB standard increased in popularity. However, PS/2 keyboards are still available and used with various PCs and industrial devices due to the simplicity of the interface. The actual layout of the keys on a PS/2 keyboard is virtually identical to that of the keyboards you use today.

The PS/2 keyboard connector is in the six-pin mini-DIN format, as shown in Figure 13-1.



Figure 13-1: A PS/2 keyboard connector

We'll use a simple PS/2 socket breakout board (such as PMD Way part 694804) to interface the keyboard with our Arduino, as shown in Figure 13-2. The socket breakout board uses four connections: 5V, GND, Clock, and Data.



Figure 13-2: A PS/2 socket breakout board

Each key on the keyboard is represented by a numerical code. When a key is pressed on the keyboard, the keycode data is sent to the PC via a bidirectional half-duplex data line. This means data can travel in either direction between the PC and the keyboard but in only one direction at a time.

NOTE

PS/2 keyboards are 5 V devices. If you're using an Arduino or compatible with a 3.3 V operating voltage, you'll need to use a bidirectional level converter board between the Arduino and PS/2 breakout.

For each keypress, the keyboard sends out a LOW start bit, followed by 8 bits that represent the key pressed, then a parity bit along the data line, with a matching Clock line to maintain timing and data accuracy. For example, Figure 13-3 shows the Clock (1) and Data (2) lines when P is pressed on the keyboard.

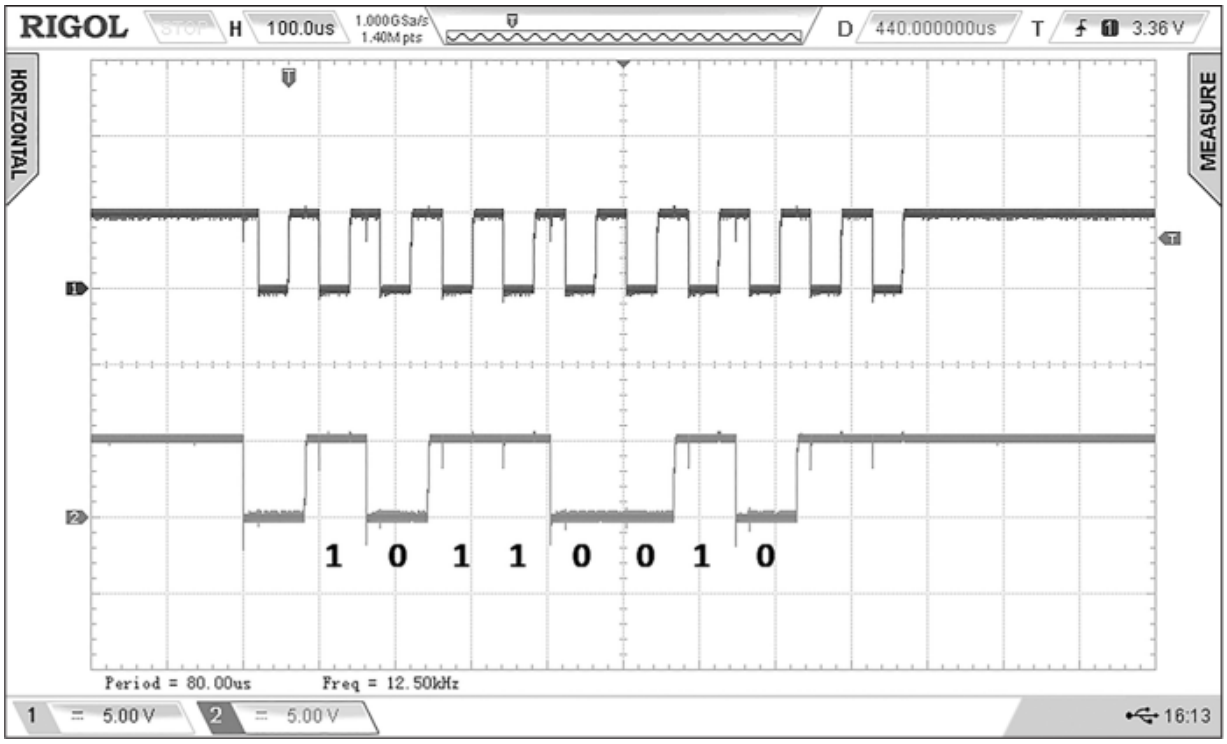


Figure 13-3: The timing diagram for PS/2 keypress shown on a digital storage oscilloscope

The keycode for the letter P is 0x4D in hexadecimal, or 01001101 in binary—you can see the bits marked on the data line in Figure 13-3. A complete list of the keycodes is available from *https://wiki.osdev.org/PS/2_Keyboard*.

To use the keyboard with your Arduino, you'll need to install a library. Open the IDE's library manager and search for **ps/2**; then install the resulting library.

To test the keyboard, connect the PS/2 breakout board to the Arduino, as shown in the schematic in Figure 13-4.

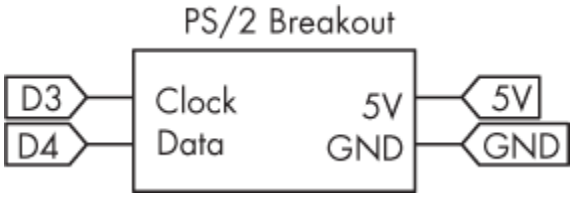


Figure 13-4: The schematic for keyboard connection

Next, enter and upload the Listing 13-1 sketch.

```
❶ #include <PS2Keyboard.h>
  PS2Keyboard keyboard;

  void setup()
  {
      delay(1000);
    ❷ keyboard.begin(4, 3);
      Serial.begin(9600);
      Serial.println("Keyboard Test:");
  }

  void loop()
```

```
{
    if (keyboard.available())
  ❸ {
        // Read the next key:
      ❹ char c = keyboard.read();
        // Check for some of the special keys:
        switch (c)
        {
            case PS2_ENTER: Serial.println(); br
            case PS2_TAB: Serial.print("[Tab]");
            case PS2_ESC: Serial.print("[ESC]");
            case PS2_PAGEDOWN: Serial.print("[Pg
            case PS2_PAGEUP: Serial.print("[PgUp
            case PS2_LEFTARROW: Serial.print("[L
            case PS2_RIGHTARROW: Serial.print("[
            case PS2_UPARROW: Serial.print("[Up]
            case PS2_DOWNARROW: Serial.print("[D
            case PS2_DELETE: Serial.print("[Del]
            // Otherwise, just print all normal
          ❺ default: Serial.print(c); break;
      }
    }
}
```

Listing 13-1: Testing the PS/2 keyboard

A few moments after uploading, open the Serial Monitor and type anything you like on the PS/2 keyboard. Figure 13-5 shows an example of the project in operation with what I typed on the keyboard displayed in the Serial Monitor.



Figure 13-5: Example output of the keyboard test sketch

The sketch initializes the library ❶ and then activates the connection to the keyboard on digital pins 4 and 3 ❷; the parameters for `keyboard.begin()` are the digital pin numbers for the Clock and Data lines, respectively. The sketch checks if a button has been pressed ❸ and assigns the result to the character variable `c` ❹. After checking for function keys using a `switch…case` statement, it sends the resulting key to the Serial Monitor ❺.

If you find the keyboard returns gibberish, check that the power to the keyboard is exactly or close to 5 V DC; the keyboard needs a straight 5 V to function correctly. Some

Arduino boards can offer a voltage very close to 5 V, but some offer less, around 4.7 V or worse. If your voltage is too low, use an external 5 V supply for the circuit and future projects that use a PS/2 keyboard.

Now that you've seen how to integrate keyboard input into your projects, let's set up an output display, using an I^2C-bus-compatible LCD.

PCF8574 LCD Modules

Using an LCD for display output is popular in the Arduino world. This generally involves using an LCD shield or wiring up an HD44780-compatible LCD with eight or so wires back to the Arduino. However, in recent years, a new method of using these LCDs has gained popularity: soldering a PCF8574 I^2C backpack unit to the display, which allows you to connect the LCD to your project with only four wires.

The PCF8574 is an I^2C-based port expander that can interface the LCD's control pins to the popular serial data bus. This obviates the need to use digital I/O pins and helps simplify the overall wiring. You can either buy the modules separately or purchase LCDs with the pre-attached modules, as shown in Figure 13-6.

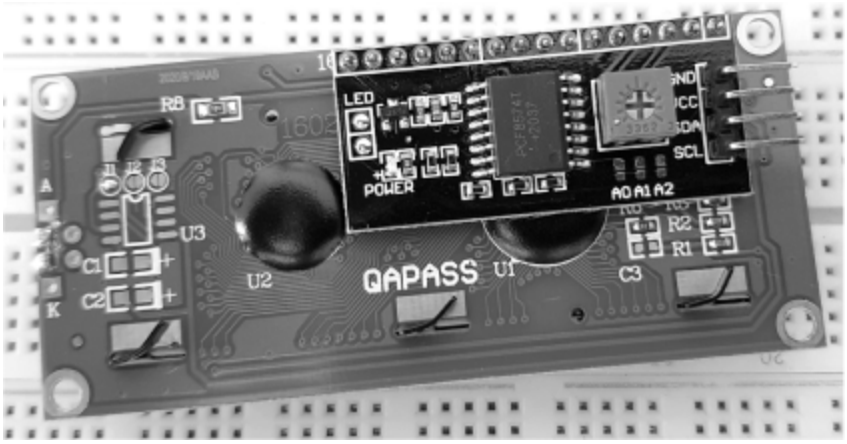


Figure 13-6: The rear of a PCF8574 I²C LCD module

To connect the module to the Arduino, use the 5V/V_{CC}, GND, Clock, and Data lines, as with any other I^2C device. You can adjust the display contrast using the trimpot on the module. You can also bridge the three pairs of solder bridge pads under the trimpot (A0, A1, or A2) to change the I^2C bus address from the default `0 x 27` to one of seven other options. To use the LCD with your Arduino, you must install a library. Open the IDE's library manager and search for **PCF8574**; then install the latest version of the library.

To test the display, connect to the Arduino, as shown in the schematic in Figure 13-7.

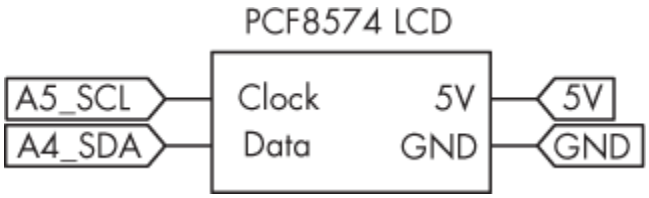


Figure 13-7: The schematic for LCD connection

Next, enter and upload the Listing 13-2 sketch.

```
❶ #include <Wire.h>
  #include <LiquidCrystal_PCF8574.h>
  LiquidCrystal_PCF8574 lcd(0x27);

  void setup()
  {
      Wire.begin();
   ❷ Wire.beginTransmission(0x27);
      lcd.begin(16, 2);
   ❸ lcd.setBacklight(255);
  }

  void loop()
  {
      lcd.home();
      lcd.clear();
      lcd.setCursor(2, 0);
      lcd.print("Hello, world!");
      delay(1000);
      lcd.setCursor(2,1);
```

```
      lcd.print("* Arduino *");
      delay(1000);
  }
```

Listing 13-2: Testing the PCF8574 LCD

After the sketch uploads, you should see `Hello, world!` and `* Arduino *` blinking on the LCD, as shown in Figure 13-8.

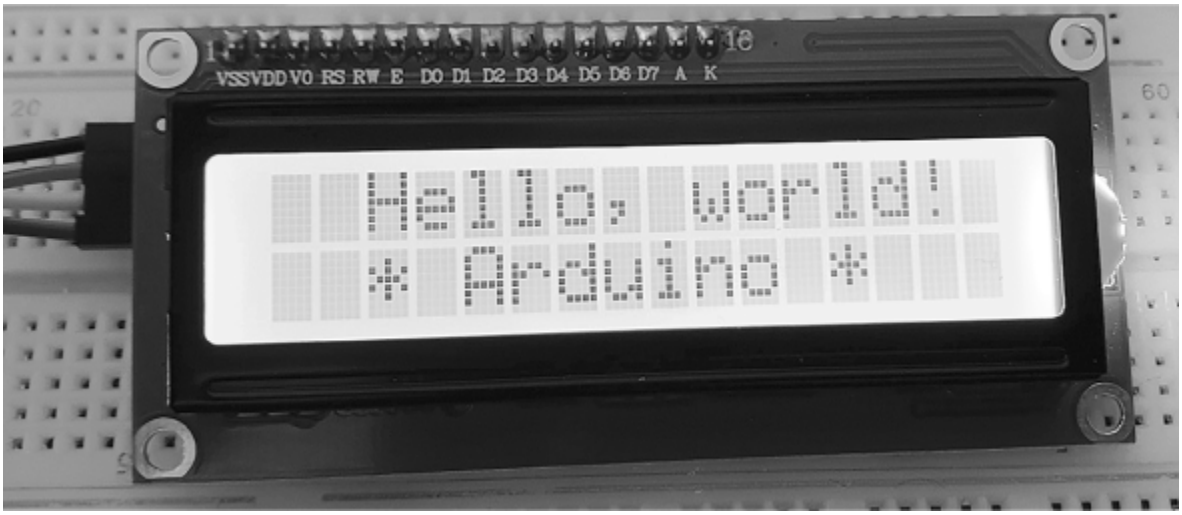


Figure 13-8: The I²C LCD in action

Once the LCD is initialized, basic operation is the same as the usual Arduino LCD library. The sketch includes the Wire library for I²C and the I²C LCD library ❶ and then creates an instance of the LCD using bus address `0x27`. It starts communication on the I²C bus to address `0x27` (the LCD) ❷, then configures the LCD size (16 characters, 2 rows) with `lcd.begin()`, and finally turns

on the backlight with `lcd.setBacklight()`. You can control the backlight by using 255 to turn the backlight on, and 0 to turn it off; in this sketch, it's set to on ❸.

The sketch uses the usual LCD functions: `lcd.home()` resets the cursor to the top-left character, `lcd.clear()` clears the display, and so on. If you are using other I²C bus devices in the same project, don't forget to restart the bus to the LCD with another `Wire.beginTransmission(0x27)` before sending display commands.

If your display doesn't work, the bus address for your display could be different from the one used in the sketch. You can use the Project #30 I²C scanner sketch described in Chapter 10 to check the device bus address and substitute it into the sketch in place of `0x27`.

Testing the I²C LCD and PS/2 Keyboard

Now that you've experimented with the I²C LCD and the keyboard separately, this section shows how they can work together as a test of the combined hardware. I hope this helps inspire your own independent project ideas, such as a text editor using an LCD or perhaps your own text-based game.

Connect the keyboard and LCD as described in the previous sections and then enter and upload the Listing 13-3 sketch.

```
❶ #include <LiquidCrystal_PCF8574.h>
  #include <Wire.h>
  LiquidCrystal_PCF8574 lcd(0x27);
  #include <PS2Keyboard.h>
  PS2Keyboard keyboard;

  int xPosition = 0;
  int yPosition = 0;

  void setup()
  {
      Wire.begin();
      Wire.beginTransmission(0x27);
      lcd.begin(16, 2);
      lcd.setBacklight(255);
      lcd.blink();
      keyboard.begin(4, 3);
  }

  void loop()
  {
    ❷ if (keyboard.available())
      {
          // Read the next key:
          char c = keyboard.read();
```

```
    // Check for some of the special keys:
❸ if (c == PS2_ENTER)
    {
        xPosition = 0;
        if (yPosition == 0)
        {
            yPosition = 1;
        } else if (yPosition == 1)
        {
            yPosition = 0;
        }
        lcd.setCursor(xPosition, yPosition);
❹ }
    else if (c == PS2_TAB)
    {
        lcd.print("[Tab]");
❺ }
    else if (c == PS2_ESC)
    {
        lcd.home();
        lcd.clear();
        xPosition = 0;
        yPosition = 0;
❻ }
    else if (c == PS2_PAGEDOWN)
    {
        lcd.print("[PgDn]");
    }
```

```
        else if (c == PS2_PAGEUP)
        {
            lcd.print("[PgUp]");
        }
        else if (c == PS2_LEFTARROW)
        {
            lcd.print("[Left]");
        }
        else if (c == PS2_RIGHTARROW)
        {
            lcd.print("[Right]");
        }
        else if (c == PS2_UPARROW)
        {
            lcd.print("[Up]");
        }
        else if (c == PS2_DOWNARROW)
        {
            lcd.print("[Down]");
        }
        else if (c == PS2_DELETE)
        {
            lcd.print("[Del]");
        }
        else
        {
            // Otherwise, just print all normal
          ❼ lcd.print(c);
```

```
            xPosition++;
          ❽ if (xPosition > 15)
            {
                xPosition = 0;
             ❾ if (yPosition == 0)
                {
                    yPosition = 1;
                }
                else if (yPosition == 1)
                {
                    yPosition = 0;
                }
             ❿ lcd.setCursor(xPosition, yPositi
            }
        }
    }
}
```

Listing 13-3: The LCD and keyboard working together

In a few moments, the LCD backlight should illuminate, and a block cursor should start blinking at the top left of the LCD. Now start typing. Whatever you type on the keyboard—including text, symbols, and most function keys—should appear and then wrap around the screen as if the LCD were a small

text editor, as shown in Figure 13-9. (I've turned off the backlight for a clearer photograph.)



Figure 13-9: The LCD displaying text entered via the PS/2 keyboard

You can press ESC to clear the screen and move the cursor back to the top left. The sketch simply takes input from the keyboard (apart from ESC) and sends each press to the display.

The sketch first initializes the required libraries for the LCD and keyboard and creates two variables to store the position of the cursor ❶. It then starts the keyboard and LCD and turns on the LCD backlight and blinking cursor.

The main loop of the sketch starts ❷ by checking for a press from the keyboard and deciding what to do based on the press ❸. First the project checks to see whether the user has pressed

ENTER. If so, the cursor on the LCD moves to the start of the second line or is sent back to the start of the first line if it's already on the second line. The sketch similarly checks for a press of TAB ❹ and displays this on the LCD if so. Or, if the user presses ESC ❺, the screen is cleared and the cursor moves back to the top left of the LCD.

The sketch continues to check for other keypresses and displays the name of the appropriate keys on the LCD ❻. This includes presses of any other keys that don't belong in the previous checks, including letters, numbers, and symbols ❼. After every keypress, the cursor position on the LCD is increased so it moves along one character for columns ❽ and rows ❾ and moves to the next line or the top left of the LCD when necessary ❿.

Now that you know how to read the PS/2 keyboard and display the keys pressed, you'll put these skills to use with the following project that reads integers from the keyboard and works with them to control the brightness and colors available with an RGB LED.

Project #39: Creating an RGB LED Tester

In future projects, you may want to test the various colors that can be displayed using a red, green, and blue (RGB) LED to calibrate the settings to reach the desired LED brightness and color. In this project, you'll build a device that accepts pulse-width modulation (PWM) values for each color and then controls each LED for a defined length of time. If you don't have an RGB LED, you can still enjoy and understand this project using three separate LEDs.

NOTE

You can learn more about PWM in Chapter 19 of Arduino Workshop*, 2nd edition (No Starch Press, 2021).*

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A PS/2 breakout board
- A PS/2 keyboard
- A PCF8574 LCD module
- One common-cathode RGB LED or one each of RGB LEDs
- Three 1 kΩ, 0.25 W, 1 percent resistors
- A solderless breadboard
- Various jumper wires

Assemble the circuit as shown in Figure 13-10. In this project, the breadboard is used as a bridge to help create more 5V and GND connections for the two modules to the Arduino.

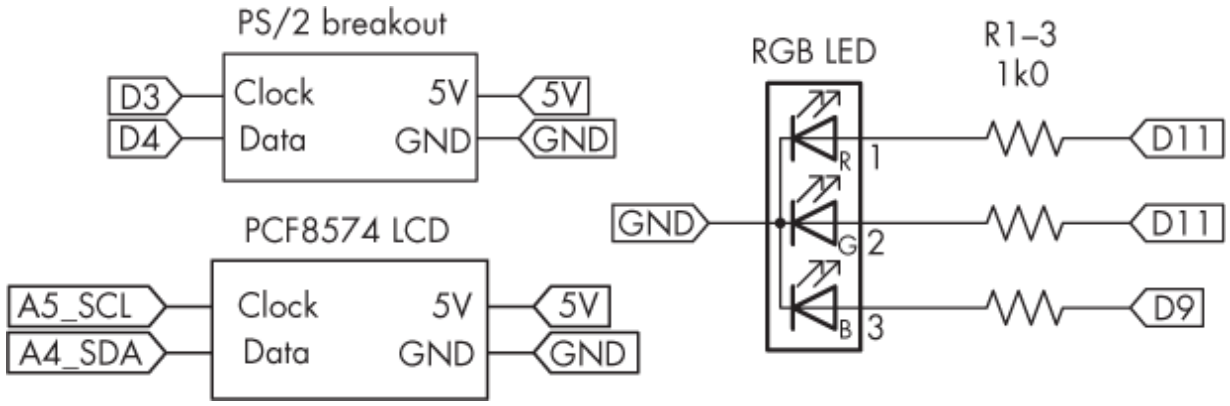


Figure 13-10: The schematic for Project #39

Now enter and upload the Project #39 sketch. The LCD should display `R:`. Using the PS/2 keyboard, enter a three-digit value between 000 and 255 to set the PWM value for the red LED. The greater the number, the stronger the color will be in the LED. Repeat this process for the green and blue LEDs in response to the `G:` and `B:` prompts. Finally, the LCD should display a `T:` prompt. Key in your required duration for the LEDs to be activated, between 000 and 255 seconds. The LEDs should then activate according to your instructions.

Figure 13-11 shows the results of entering in the brightness values for all three LEDs and the duration of activation in seconds.



Figure 13-11: The Project #39 user interface in operation

Let's see how this works:

```
  // Project #39 - RGB LED tester

❶ #include <Wire.h>
  #include <LiquidCrystal_PCF8574.h>
  LiquidCrystal_PCF8574 lcd(0x27);
  #include <PS2Keyboard.h>
  PS2Keyboard keyboard;

  #define red 11
  #define green 10
  #define blue 9

❷ int readDigit()
  {
      int _digit=-1;
    ❸ do
```

```
    {
        if (keyboard.available())
        {
            // Read the next key:
            char c = keyboard.read();
            switch (c)
            {
                case '1' : _digit=_digit+2; brea
                case '2' : _digit=_digit+3; brea
                case '3' : _digit=_digit+4; brea
                case '4' : _digit=_digit+5; brea
                case '5' : _digit=_digit+6; brea
                case '6' : _digit=_digit+7; brea
                case '7' : _digit=_digit+8; brea
                case '8' : _digit=_digit+9; brea
                case '9' : _digit=_digit+10; bre
                case '0' : _digit=0; break;
                default: _digit=0; break;
            }
          ❹ lcd.print(c);
        }
  ❺ }
    while(_digit==-1);
    return _digit;
}
❻ int get3digit()
{
    int z;
```

```
    z = readDigit()*100;
    z = z + (readDigit()*10);
    z = z + readDigit();
    return z;
}

void setup()
{
 ❼ Wire.begin();
    Wire.beginTransmission(0x27);
    lcd.begin(16, 2);
    lcd.setBacklight(255);
    keyboard.begin(4, 3);
    DDRB = B111111; // Set LED pins to output
}

void loop()
{
 ❽ int _red;
    int _green;
    int _blue;
    int _delay;
    lcd.home();
    lcd.clear();
    lcd.setCursor(1, 0);
    lcd.print("R:");
    lcd.setCursor(4, 0);
    _red = get3digit();
```

```
    lcd.setCursor(8, 0);
    lcd.print("G:");
    lcd.setCursor(11, 0);
    _green = get3digit();

    lcd.setCursor(1, 1);
    lcd.print("B:");
    lcd.setCursor(4, 1);
    _blue = get3digit();

    lcd.setCursor(8, 1);
    lcd.print("T:");
    lcd.setCursor(11, 1);
 ❾ _delay = get3digit();
    analogWrite(red, _red);
    analogWrite(green, _green);
    analogWrite(blue, _blue);
    delay(_delay*1000);

    analogWrite(red, 0);
    analogWrite(green, 0);
    analogWrite(blue, 0);
}
```

The sketch first includes and configures the I^2C LCD and keyboard libraries, as usual ❶, and then defines the LEDs'

Arduino digital pin numbers to color names for ease of reference. The custom function `readDigit()` ❷ returns an integer that is used to return the press of a number on the PS/2 keyboard. This function uses a variable `_digit`, which is set to `–1` upon initialization and stays at that value until a number key is pressed. It then returns the value of any button the user presses as the return value for the function.

Since the `readDigit()`function needs to wait until a number is pressed on the keyboard, a `do…while` function ❸ continues to loop while awaiting a keypress by testing the value of `_digit`. When the sketch detects a keypress, however, the function interrogates the key's value with a `switch…case` function. The value of `_digit` is then set to the value of the number key pressed, which is equal to the number plus 1. For example, if the user presses the 5 key, the sketch adds 6 to `_digit` to reflect this. If the user doesn't press one of the numbers, it will return a 0 for `_digit`.

Once the value of `_digit` has changed, the test in the `do…while` function ❺ fails. The code can thus continue to display the appropriate number on the LCD ❹ and return the value of the number pressed.

The project needs three-digit values from the user for PWM and time values. Therefore, the custom function `get3digit()` ❻ creates a three-digit number by calling for a digit from the keyboard three times and then combining the number into the variable `z`, which is returned as the result of this function.

The sketch starts the I^2C bus, LCD, and keyboard ❼ and then sets the digital pins for the LEDs to outputs using port manipulation (discussed in Chapter 2). I've set variables to hold the PWM values for each LED and time on delay ❽. The project then accepts the user's value for red PWM, green PWM, blue PWM, and the number of seconds for the LEDs to stay on. These are actioned at ❾, and the delay is converted to milliseconds. Once the delay has finished, the LEDs are turned off, and the system starts again.

Now that you have an example of capturing and working with numerical input from a PS/2 keyboard, you'll build a device to capture text as well, saving that input to an SD card for use with a PC.

Project #40: Building a Text Capture Device

The text capture device described in this project lets you record notes or other writing for later reference, without the

distractions of internet or cell phone notifications to lure you away from real work. Everything you type onto the project's PS/2 keyboard is recorded to a text file on an SD memory card, which you can then open on your PC for final editing.

NOTE

You can learn more about using SD memory cards in Chapter 7 of Arduino Workshop*, 2nd edition (No Starch Press, 2021).*

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A PS/2 breakout board
- A PS/2 keyboard
- A PCF8574 LCD module
- An SD card module or SD card shield for Arduino
- A blank SD memory card
- A solderless breadboard
- Various jumper wires

Assemble the circuit as shown in Figure 13-12. If you're using the SD card shield, insert it onto the Arduino and then connect the LCD and PS/2 modules. Again, this project uses the

breadboard as a bridge to help create more 5V and GND connections for the two modules to the Arduino.

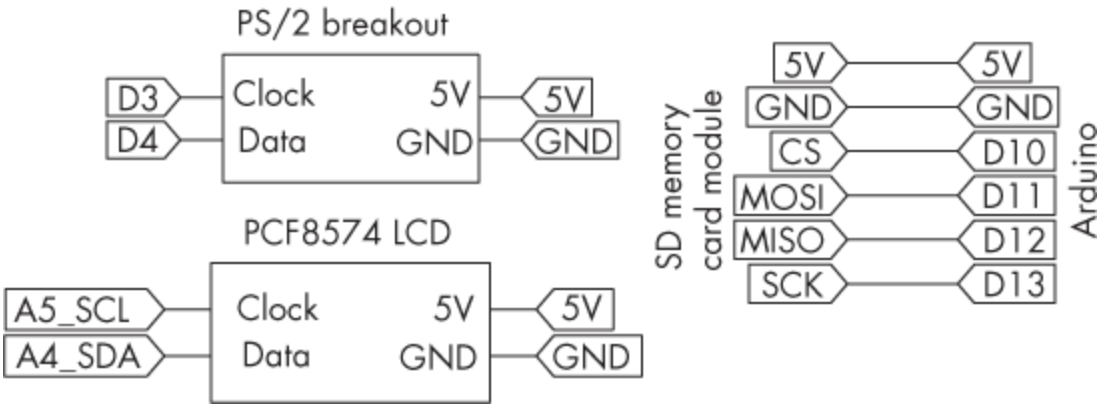


Figure 13-12: The schematic for Project #40

Now enter and upload the Project #40 sketch and then insert the SD card into the module. Start typing when prompted by the LCD, as shown in Figure 13-13.



Figure 13-13: The LCD from Project #40 displaying text captured from a keyboard

As you continue typing, the cursor should scroll around to the second row and then back to the first, in the same manner as shown in Figure 13-14.



Figure 13-14: Text scrolling across two lines in the LCD

As you type, the Arduino should automatically save every 100 characters to a file on the SD card. When you've finished and want to take the SD card away, press ESC on the keyboard. The Arduino should save any remaining characters and then prompt you to remove the SD card, as shown in Figure 13-15.



Figure 13-15: The LCD displaying the message that it's okay to remove the SD card

If there's an issue with the SD card anytime you attempt a save, the display should show the message `SD card fail`, as shown in Figure 13-16.



Figure 13-16: The LCD indicating a problem with the SD card

In this case, your SD card may not be properly formatted, your write-protect switch may be set to ON, or your card may be missing from the module. Once you've fixed any problems and

removed your SD card, you can view and edit the text file created with your PC by inserting the SD card into the PC.

Let's see how this works:

```
  // Project #40 - Text capture device

❶ char _text[100];
  int _counter = 0;
  int xPosition = 0;
  int yPosition = 1;

  #include <SD.h>
  #include <LiquidCrystal_PCF8574.h>
  #include <Wire.h>
  LiquidCrystal_PCF8574 lcd(0x27);
  #include <PS2Keyboard.h>
  PS2Keyboard keyboard;

❷ void halt()
  {
      do {} while (1);
  }

❸ void saveText(boolean _halt)
  {
      if (!SD.begin(10))
```

```
    {
        lcd.setCursor(1, 1);
        lcd.print("SD card fail");
        halt();
    }
❹ File dataFile = SD.open("WRITING.TXT", FILE_W
    if (dataFile)
    {
        // Save and clear array
        for (int i = 0; i < 100; i++)
        {
            dataFile.print(_text[i]);
            _text[i] = " ";
        }
        dataFile.close();
    }
❺ if (_halt == true)
    {
        File dataFile = SD.open("WRITING.TXT", FI
        if (dataFile)
        {
            dataFile.println();
            dataFile.println("-----------------
            dataFile.close();
        }
        lcd.setCursor(1, 1);
        lcd.print("OK remove card");
        halt();
```

```
    }
}

void setup()
{
 ❻ Wire.begin();
    Wire.beginTransmission(0x27);
    lcd.begin(16, 2);
    lcd.setBacklight(255);
    lcd.blink();
    lcd.clear();
    lcd.home();
    lcd.print("Start typing:");
    lcd.setCursor(xPosition, yPosition);
    keyboard.begin(4, 3);
    pinMode(10, OUTPUT);
}

void loop()
{
 ❼ if (keyboard.available())
    {
        char c = keyboard.read();
        if (c == PS2_ENTER)
        {
         ❽ _text[_counter] = '\n';
           _counter = _counter + 1;
           xPosition = 0;
```

```
            if (yPosition == 0)
            {
                yPosition = 1;
            }
            else if (yPosition == 1)
            {
                yPosition = 0;
            }
            lcd.setCursor(xPosition, yPosition);
        }
          else if (c == PS2_ESC)
          {
              lcd.home();
              lcd.clear();
              saveText(true);
          }
          else
          {
              _text[_counter] = c;
           ❾ lcd.print(_text[_counter]);
              _counter = _counter + 1;
              if (_counter >= 99)
              {
               ❿ saveText(false);
                  _counter = 0;
              }
              xPosition++;
              if (xPosition > 15)
```

```
                {
                    xPosition = 0;
                    if (yPosition == 0)
                    {
                        yPosition = 1;
                    }
                    else if (yPosition == 1)
                    {
                        yPosition = 0;
                    }
                    lcd.setCursor(xPosition, yPositi
                }
            }
        }
    }
```

This sketch builds on Listing 13-3 to include extra code for writing to the SD card. First, it declares the global variables ❶, including the `array_text[]` buffer to hold the user input before writing to the SD card, the `_counter` variable used to track the number of characters that are stored in the buffer, and the cursor position on the LCD.

Next, the sketch includes and initializes the required libraries for the SD card module, LCD, and keyboard. The custom function `halt()` ❷ is used to stop the sketch operation and is

called after the text is saved to the SD card so the user can safely remove the card. It simply loops around forever in the `do…while` function, as `1` is always true.

The custom function `saveText(boolean _halt)` ❸ saves data to the SD card, either saving text and returning to user input or saving text and ending operation, as indicated by the Boolean parameter `false` or `true`, respectively. The function checks whether the SD card is okay to use, stopping and returning an error if there's an issue, as shown in Figure 13-16. If there are no errors, the sketch opens the file whose name is set in this function, to which it will write the data; it creates that file if none exists ❹. The sketch then writes the text buffer to the SD card and clears the buffer to prepare it for more data.

If the parameter `true` was passed to the `saveText(boolean _halt)` function ❺, the code opens the file on the SD card, writes a line of dashes to end the text file, and then closes it and indicates this to the user, as shown in Figure 13-15. From ❻, the usual requirements for activating the I^2C bus, LCD, and keyboard, as well as setting D10 to an output for the SD card module's SPI interface, occur.

Next, the sketch sets up the project's main operation: capturing text from the keyboard, displaying it on the LCD, and writing it

to the SD card. At ❼ the sketch is ready to receive a keystroke from the PS/2 keyboard. If the user presses ENTER, the sketch inserts a new line character | in the text array ❽, increments the size counter of the array by 1, and moves the cursor along the LCD as required to the alternate line. If the user presses ESC, the sketch clears the LCD, saves the data to the SD card, and stops operation via a call to `saveText(true);`. All other keypresses are inserted into the text array as normal text ❾, which is then displayed on the LCD, while the sketch increments the size counter by 1.

The sketch then checks the text array size. If it is greater than or equal to 99 characters in size, the text array is written to the text file by calling `saveText(false);` ❿, which returns to normal text capture after the write, followed by resetting the counter to 0. Finally, the sketch updates the cursor position, checking if the text runs over the margin and repositioning the cursor if necessary.

For a final challenge, you might add a real-time clock to Project #40, as well as a data timestamp when the file is closed.

Moving On

Many projects suffer from a poor user interface; as you saw in this chapter, using a PS/2 keyboard simplifies the process of entering data into a project. In this chapter, you learned how to capture data from the keyboard into an Arduino and how to save data entered with the keyboard. I hope this chapter served as inspiration for improving the configuration or operation in your own work.

In the next chapter, I'll show you how to harness Bluetooth modules to allow for wireless communication between smartphones and other devices with your Arduino projects.

14

CONTROLLING THE ARDUINO WITH BLUETOOTH



Most contemporary smartphones, PCs, or tablets can communicate wirelessly with other devices using Bluetooth. In this chapter, you'll use inexpensive Bluetooth modules along with Windows, Android, or Apple machines to control your Arduino projects.

You'll learn to:

- Use level converter boards with serial Bluetooth interface modules for voltage translation

- Pair Bluetooth modules with Android-, Apple-, and Windows-based devices
- Edit the parameters of a serial Bluetooth module

You'll also send data to and from an Arduino project via Bluetooth and then build your own Android app with the MIT App Inventor to remotely control your project.

Bluetooth Modules

There are many types of Bluetooth modules. This chapter relies on the HC-05-type Bluetooth module, such as PMD Way part 590526, as shown in Figure 14-1.

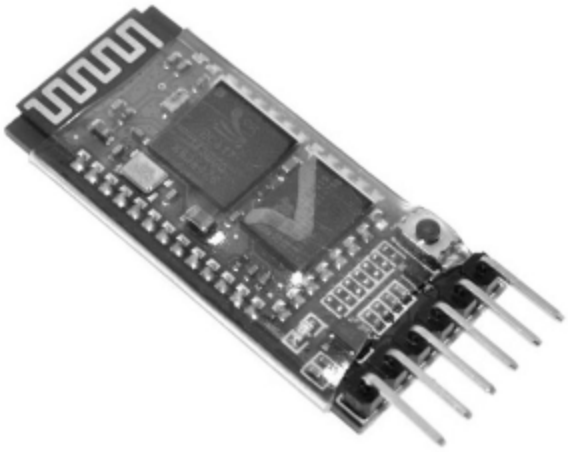


Figure 14-1: An HC-05 Bluetooth module

Other modules on the market like the HC-06 won't work with the examples and projects in this chapter, as they can only

transmit data, not receive it. These modules use a TX and RX line for serial communication between the module and your Arduino, which can be used with the SoftwareSerial library as usual. The data rate will default to 9,600 bps. I'll explain how to set this up later in the chapter.

Physically wiring the modules to your Arduino projects is easy, thanks to the standard pin spacing at the end of the module, and pins are always labeled on the rear of the module. They will both operate using a 5 V DC power supply. However, the communication pins use a lower logic voltage of 3.3 V DC. The best way to convert 5 V to 3.3 V is to use a level converter module, which I'll explain next.

Level Converter Modules

Many devices have data communication pins designed to operate at 3.3 V instead of the usual 5 V DC. This can be for serial commutations such as Bluetooth modules or when using other data buses such as SPI or I^2C. However, these pins do not work reliably with 5 V devices such as the Arduino Uno or compatible, so you'll need to convert signals between 3.3 V and 5 V DC with a voltage converter board.

To do so, you can use a level converter module such as the one shown in Figure 14-2, PMD Way part 441079.

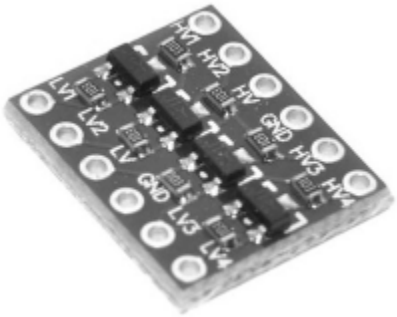


Figure 14-2: A level converter module

Most level converter modules include inline header pins, which you will need to solder to the board. Different level converters will have varying number of channels; you need at least two channels for our Bluetooth devices, one for serial TX and one for serial RX.

All level converters require connections to both a 5 V and 3.3 V supply for operation. Simply connect the lower voltage device to the LV channels side of the level converter and connect the Arduino to the matching HV channels side. I'll explain how to use the level converter in this chapter's projects.

Before using a Bluetooth module with your PC or other device, you'll need to pair the Bluetooth module with that device, as you would with Bluetooth headphones or speakers. For future

reference and when working through this chapter, the following sections give instructions for pairing Windows PCs, Android devices, and macOS computers with the module that you can follow as required for the projects. Note that the LED on the Bluetooth module will blink slowly when successfully paired with any of these devices.

Pairing with Windows Devices

The steps to pair the Bluetooth module with a Windows machine are similar to those for any other Bluetooth device, such as headphones or speakers. First ensure the module is connected to the 5V and GND pins for power. Once power is connected, the LED on the module will blink rapidly, meaning there's no Bluetooth connection between the module and a device.

When using a Windows-based PC, simply search for a new Bluetooth device from the Bluetooth & devices page in Settings. Select the HC-05 device in the list and you'll be prompted for a PIN. Enter **1234**, click **Connect**, and you're ready to go.

You can then send and receive data to the Bluetooth module via a COM: port address. These addresses can be found in the Windows Device Manager.

To continue, skip to the “Sending Data via Bluetooth” section on page 228.

Pairing with macOS Devices

Open the System Preferences window from the Apple menu and select **Bluetooth**. Make sure Bluetooth is turned on, so that the computer begins to search for nearby devices.

Find HC-05 among the connection options and click **Connect**. In a moment, you should be shown a passkey error; that’s okay. Click **Options**, enter the PIN **1234**, and then click **Connect**.

After another moment, your Mac should pair with the Bluetooth module.

To continue, skip forward to the “Sending Data via Bluetooth” section on page 228.

Pairing with Android Devices

The steps to pair the Bluetooth module with an Android are similar to those for any other Bluetooth device, such as headphones or speakers. Turn on Bluetooth on your Android device, search for a new device, and then tap the listing for **HC-**

05. You should be prompted for the PIN. Enter **1234** to pair your module.

Now that you've paired your module with an Android or computer, I'll show you how to send data from your Arduino via Bluetooth.

Sending Data via Bluetooth

The Bluetooth module is a serial device, and you can easily send data out using the SoftwareSerial library. Let's test this now.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A solderless breadboard
- A level converter module
- Various jumper wires
- An HC-05 Bluetooth module

Assemble the circuit as shown in Figure 14-3.

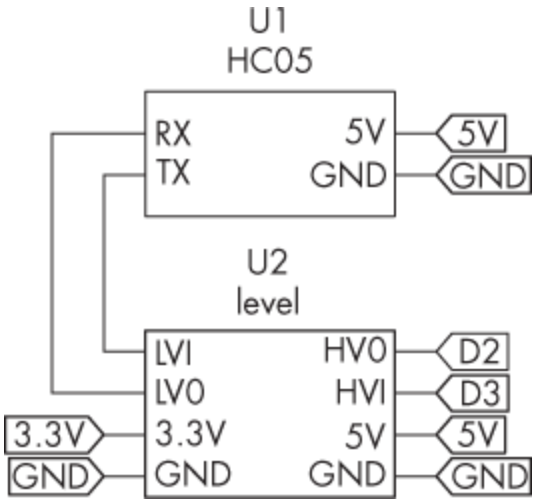


Figure 14-3: The schematic for connecting a Bluetooth module to Arduino via a level converter

Now enter and upload Listing 14-1.

```
❶ #define PIN_BT_TX 3
  #define PIN_BT_RX 2
  #include <SoftwareSerial.h>
  SoftwareSerial BT(PIN_BT_RX, PIN_BT_TX);
  void setup()
  {
      // Set the data rate for the SoftwareSerial
    ❷ BT.begin(9600);
      // Send test message to other device:
  }

  void loop()
  {
      BT.print("millis = ");
```

```
    BT.println(millis());
    delay(500);
}
```

Listing 14-1: Testing the Bluetooth output

Sketch operation is simple. Since the Bluetooth module is a serial data device, the sketch configures it to use a software serial port defined on pins D2 and D3 ❶. The sketch then starts the software serial port ❷ and uses the `print()` and `println()` functions to send the data out via the module. The LED will blink twice around every five seconds when there's a data connection between it and another device.

You now need to select a method to receive the data being sent from the Arduino via Bluetooth. I'll show you how to do this with a PC, a macOS computer, and an Android device. Follow the instructions for all the platforms you have available to you.

To Windows Devices

To receive Arduino data on your Bluetooth-enabled PC, you can use any terminal software that can communicate using a COM: port. For this chapter, I suggest using the CoolTerm software by Roger Meier, available from *http://freeware.the-meiers.org*.

Once you've installed CoolTerm, use the Options menu to select the appropriate COM: port and click **Connect**. After a few moments, the data from the Arduino should scroll down the window, as shown in Figure 14-4.



Figure 14-4: Example output of Listing 14-1 on a PC terminal

Now that you can receive Arduino data on your Windows device, skip to Project #41 to put this capability to use.

To macOS Devices

To receive the data on your Bluetooth-enabled Mac, you can use any terminal software that can communicate using a COM: port. For this chapter, I suggest using the CoolTerm software by Roger Meier available from *http://freeware.the-meiers.org*.

Once you've installed CoolTerm, you may be presented with a macOS security error when trying to run the software. To get

around this, locate the CoolTerm icon in the Applications folder, right-click it, and select **Open**; you may need to click **Open Anyway** in the Privacy and Security tab in the macOS System Preferences. Afterward, CoolTerm should run as normal when you use it in future.

Once you've resolved the error, the data from the Arduino should begin to scroll down the terminal window, as shown in Figure 14-5.

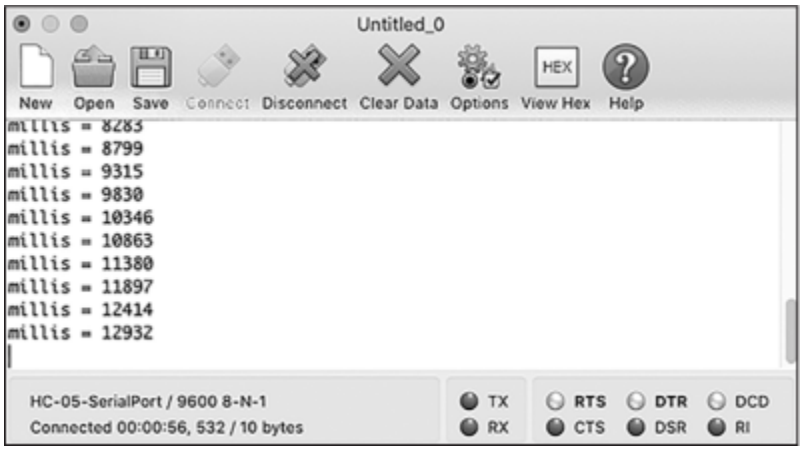


Figure 14-5: Example output of Listing 14-1 on a macOS computer

Next, move on to Project #41 to put this capability to use.

To Android Devices

On your Android device, find the Bluetooth Terminal HC-05 app in the Google Play store and download it. Once you install and open the terminal app, it should prompt you with a list of

paired Bluetooth devices. Select **HC-05**. Once the connection is complete, the terminal should display the `millis()` values from the Arduino in the terminal app window. Figure 14-6 demonstrates this sequence.

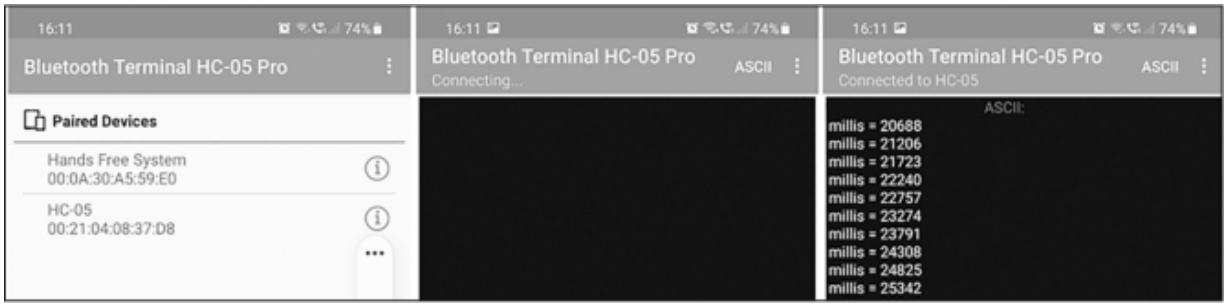


Figure 14-6: Example output of Listing 14-1 on an Android device

Now that you can receive Arduino data on your Android device, move on to Project #41 to put this capability to use.

You'll use your new ability to view data from the Arduino on your device in the following project.

Project #41: Capturing Data for Real-Time Monitoring

In this project, using the BMP180 temperature and air pressure sensor introduced in Chapter 10, you'll build a system to monitor data using the Arduino and the Bluetooth module you just set up. You can use this type of data capture with a smartphone for real-time monitoring if you need to stay mobile in your area or with a PC running CoolTerm for recording and

analysis. You can also try both if you have both devices available.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A solderless breadboard
- A level converter module
- Various jumper wires
- An HC-05 Bluetooth module
- A BMP180 temperature and air pressure sensor board

Next, assemble the circuit as shown in Figure 14-7.

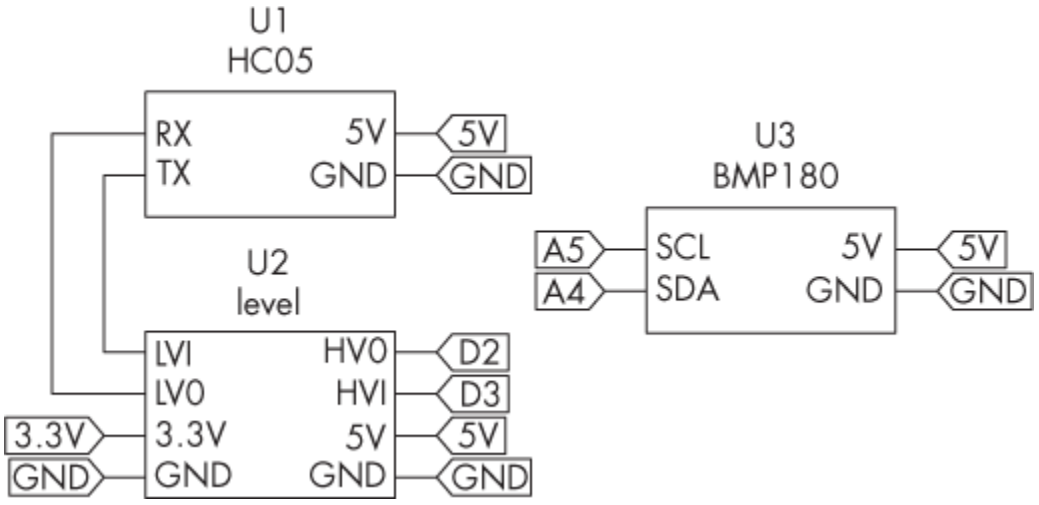


Figure 14-7: The schematic for Project #41

Now enter and upload the Project #41 sketch. Next, open the terminal app on your Android device or Windows or Mac

machine and connect to the Bluetooth module as described in the previous section. You should begin to see the temperature and air pressure displayed, as shown in Figure 14-8.

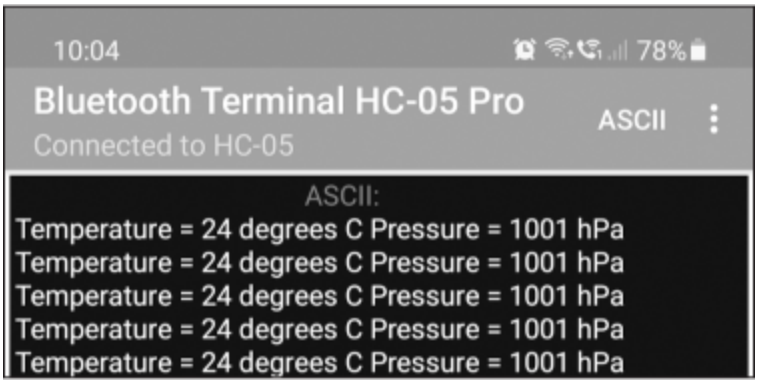


Figure 14-8: Example output from Project #41

Let's see how this works:

```
  // Project #41 - Capturing Data via Bluetooth

❶ #define PIN_BT_TX 3
  #define PIN_BT_RX 2
  #include <SoftwareSerial.h>
  #include "Wire.h"
  #include "Adafruit_BMP085.h"
  SoftwareSerial BT(PIN_BT_RX, PIN_BT_TX);
  Adafruit_BMP085 bmp;

❷ int temperature;
  int pressure;
```

```
void setup()
{
    // Set the data rate for the SoftwareSerial
  ❸ BT.begin(9600);
    bmp.begin();
}

void loop()
{
  ❹ BT.print("Temperature = ");
    temperature = bmp.readTemperature();
    BT.print(temperature);
    BT.print(" degrees C");
    BT.print(" Pressure = ");
  ❺ BT.print(int(bmp.readPressure() / 100));
    BT.println(" hPa");
    delay(1000);
}
```

This sketch combines reading data from the BMP180, neatly formatting the data, and sending it to the Bluetooth module. The sketch includes and configures the required libraries ❶ and then declares two variables ❷ to store the temperature and air pressure data. It starts the software serial connection and BMP180 sensor ❸ and sends out the temperature data ❹. The

pressure readings from the BMP180 are converted from pascals to hectopascals ❺ and then sent out as well.

Now that the data is being sent out via Bluetooth, you can capture it to a text file using the CoolTerm software. If you have a device connected to the project, disconnect it, and then connect your macOS or Windows PC as described in the previous section. Open CoolTerm and connect. The data should be shown in the terminal window, as in Figure 14-9.

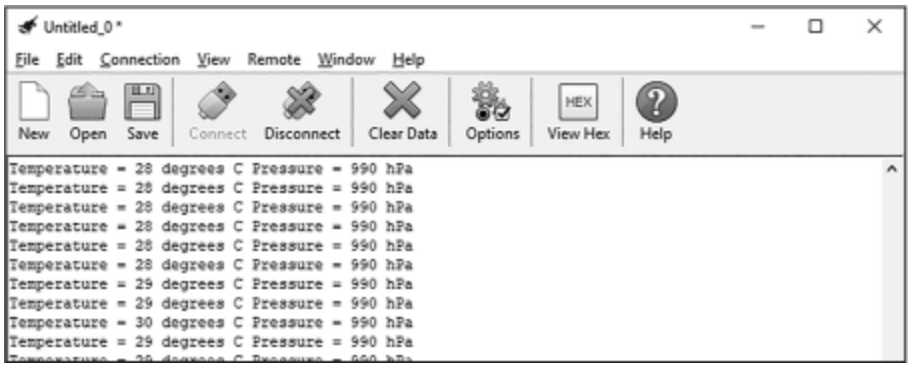


Figure 14-9: Example output displayed on a PC

Press CTRL-R to open the Capture to Text File dialog, enter a filename, select a location, and click **Save**. All the data from the Arduino should be written to the text file. When you're finished, press CTRL-SHIFT-R to stop and finalize the text file.

You can now open the text file in a spreadsheet such as Microsoft Excel, as shown in Figure 14-10, using space as the

delimiter during the import process. Your data is now ready for analysis.

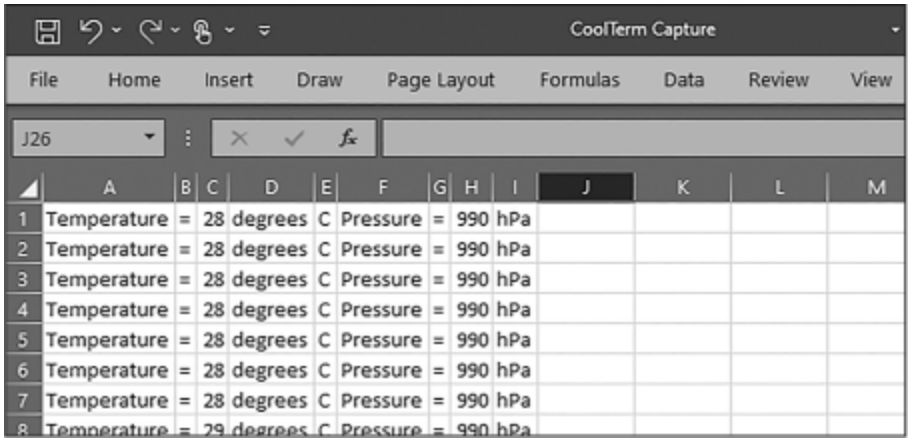


Figure 14-10: The captured data in Excel

The ease of working with any data sent from your Arduino will depend on how it's formatted for transmission. However, you now have a wireless option for capturing any data your Arduino projects can generate and transforming the data into a user-friendly format.

Furthermore, by using either a smartphone and a terminal app or a laptop computer with Bluetooth, you can create a walk-up monitoring system for various devices rather than going to the expense of building a display system for the local data. Just bring your computer or Android phone within range of your Bluetooth Arduino project and connect to the project to monitor the data readings. With this technique, you can create secure

data transmission known only to yourself, monitoring it with your devices rather than displaying it for anyone to see.

In the next project, you'll learn to send data to and from the Arduino and another device by creating a Bluetooth remote-controlled project.

Project #42: Using Bluetooth for Digital I/O Control

Like any external serial device, the Bluetooth module can both send and receive data, allowing you to send data to your Arduino for various purposes. With this project, you'll make a simple remote control for LEDs connected to four digital pins, which will serve as a framework for your own wireless remote-control projects.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable
- A solderless breadboard
- A level converter module
- Various jumper wires
- An HC-05 Bluetooth module
- Four LEDs
- Four 1 kΩ, 0.25 W, 1 percent resistors

Assemble the circuit as shown in Figure 14-11. Once you're finished with this project, keep the circuit assembled, as you'll use it in the following project.

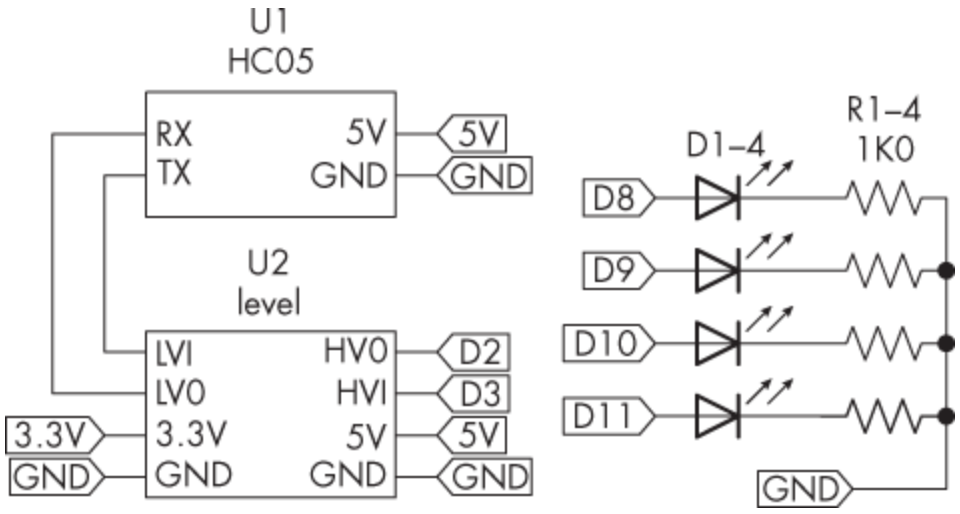


Figure 14-11: The schematic for Project #42

Enter and upload the Project #42 sketch and then open the terminal app on your Android device or computer and connect to the Bluetooth module. Send a question mark (**?**) with the terminal app, and the Arduino should respond with a quick control summary.

You can also send a 0, 1, 2, or 3 to turn on digital pins D8 to D11, respectively. To turn them off, send 4, 5, 6, or 7, respectively. After each command, the Arduino will return it as confirmation. Figure 14-12 shows this output in the Android terminal app.

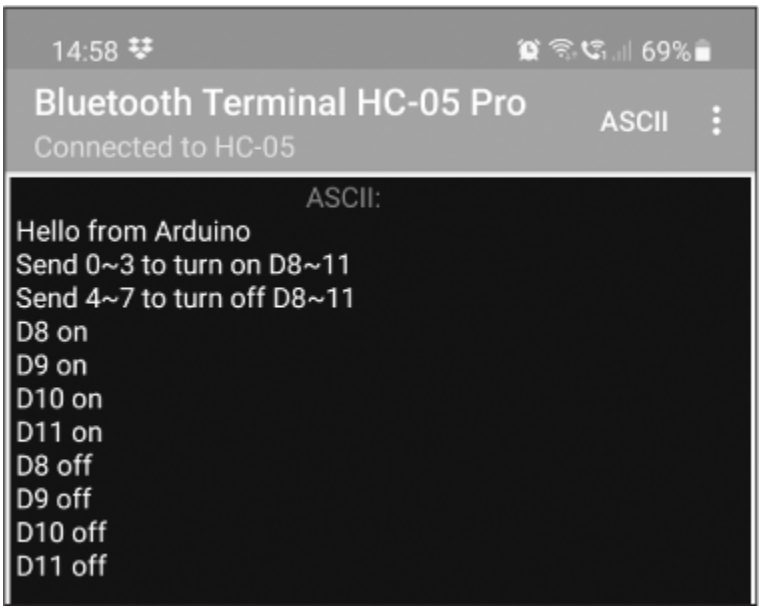


Figure 14-12: Controlling Project #42 from an Android phone

As with the previous project, you can use the Bluetooth-equipped computer for the same control methods, as shown in Figure 14-13.

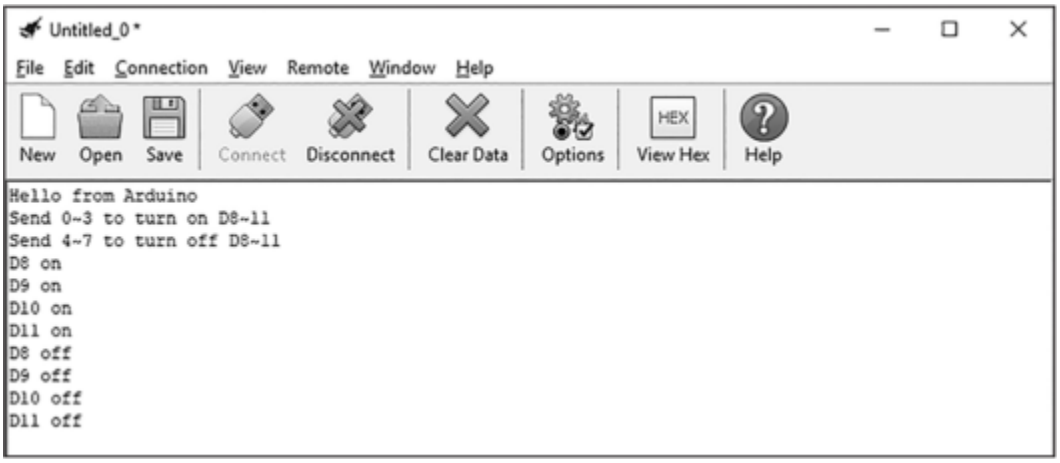


Figure 14-13: Controlling Project #42 from a PC

Let's see how this works:

```
// Project 42 - Digital I/O control with Android

❶ #define PIN_BT_TX 3
#define PIN_BT_RX 2
#include <SoftwareSerial.h>
SoftwareSerial BT(PIN_BT_RX, PIN_BT_TX);
void sendHelp()
{
    BT.println("Send 0~3 to turn on D8~11");
    BT.println("Send 4~7 to turn off D8~11");
}

void setup()
{
    // Set digital pin to control as an output:
 ❷ DDRB = B111111;
    // Set the data rate for the SoftwareSerial p
 ❸ BT.begin(9600);
    // Send test message to other device:
    BT.println("Hello from Arduino");
}

void loop()
{
 ❹ char a; // Stores incoming character from oth
    if (BT.available())
    {
```

```
      ❺ a = (BT.read());
        if (a == '0') {digitalWrite(8, HIGH); BT
        if (a == '1') {digitalWrite(9, HIGH); BT
        if (a == '2') {digitalWrite(10, HIGH); B
        if (a == '3') {digitalWrite(11, HIGH); B
        if (a == '4') {digitalWrite(8, LOW); BT.
        if (a == '5') {digitalWrite(9, LOW); BT.
        if (a == '6') {digitalWrite(10, LOW); BT
        if (a == '7') {digitalWrite(11, LOW); BT
      ❻ if (a == '?') {sendHelp();}
    }
}
```

The software serial port for the Bluetooth module is first set up ❶. The custom function `sendHelp()` sends out the instructions when the user sends a question mark to the Arduino. The sketch sets the digital output pins for D13 to D8 to outputs ❷ and starts the software serial connection ❸. It also declares the character variable used to store the incoming command from Bluetooth ❹.

When a character is received from the Bluetooth module, it's stored into the character variable `a` ❺; a series of `if` functions then convert the character into the required actions. Along with turning digital pins on or off, a message is sent out to confirm

that the command has taken place. If the Arduino receives a question mark, the sketch calls the `sendHelp()` function ❻, which tells the user how to operate the project.

You can use this project's remote-control framework in your own future work to keep controls out of the public view, by using Bluetooth on an Android phone or a computer rather than a control panel. You can also modify this project to control the relay board described in Project #26 in Chapter 8.

You'll now make your own smartphone app to control this project to make it more user-friendly.

Project #43: Using an Android App for Bluetooth Control

Controlling the Arduino digital outputs with the terminal app on an Android device is simple enough for you, the creator. When others need to control the output, however, a custom Android smartphone app is preferable, as it insulates the end user from any chances of incorrect usage. Instead of writing the code for such an app from scratch, in this project I'll show you how to create your own Android app to control the hardware in Project #42 using the free MIT App Inventor, an online service

provided by Google and the Massachusetts Institute of Technology.

NOTE

You can use the MIT App Inventor website on Windows, macOS, or Linux, but the resulting app will run only on an Android device.

Figure 14-14 shows what your complete app interface will look like, showing buttons to connect the Android phone or tablet to the project via Bluetooth and then turn the Arduino digital outputs D8 to D11 on or off.

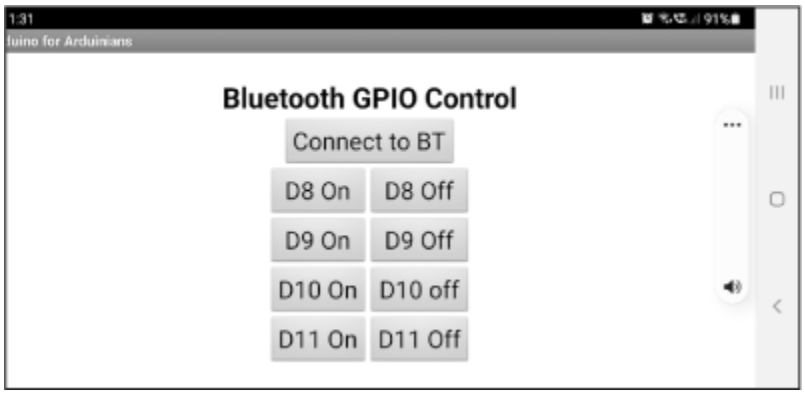


Figure 14-14: The completed Android App to control Project #42

The process of building your own app involves two main stages. The first revolves around the design of the app, such as location of buttons and text on the display. The second stage involves

determining what actions the app takes in response to user input or interactions with the hardware. With the MIT App Inventor, each stage involves a drag-and-drop process of moving items into place to build the app interface and operational instructions. I'll show you how to navigate the process to create your app.

The hardware for this project is identical to that in Project #42. Once you have set up the circuit for that project, visit the MIT App Inventor website at *https://appinventor.mit.edu*. Click the **Create Apps** button at the top of the home page. A new window should appear and ask you to log in with a Google account. You will need to create a Google account if you don't have one.

Once you've logged in and agreed to the terms of service, a welcome screen should appear. Click **Start a Blank Project** and enter a project name when prompted. I've named mine Project42, but you can choose whatever name you like. This will also be the name of the app when you later download it to your Android device. Note that you can save and load your progress as with any other editor using the Projects menu at the top of the screen.

Your screen should now look like the one in Figure 14-15, with a blank phone screen and various menus.



Figure 14-15: The Android app to control Project #42

You'll start building the interface from top to bottom, beginning with the header text. On the left of the display is the Palette. Click **User Interface**, which reveals various interface options. Place the mouse pointer over **Label**, and then click and drag it into the display of the phone, as shown in Figure 14-16.



Figure 14-16: Building a label

To edit the label, click the **Label1** entry in the Components windows to the right of the phone. This should allow you to edit

the properties of the label using the Properties window on the right side of the screen, shown in Figure 14-17.

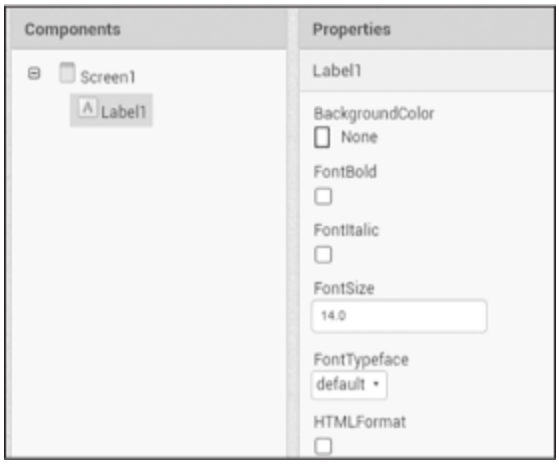


Figure 14-17: The label properties

Turn on **FontBold**, set the size to **28**, and enter **Bluetooth GPIO Control** into the text field. These changes should take place immediately and be reflected in the phone display in the center of the screen, as shown in Figure 14-18.



Figure 14-18: The completed label

Notice that when you add an item to the phone display, it appears in the Components list on the right. Clicking that item in the list allows you to edit its properties in the Properties box on the far right. You'll use this feature in a moment.

Next, you'll start adding the buttons to the app. The first button will activate the Bluetooth connection menu on the phone. To add this button, click and drag **ListPicker** from the User Interface list on the left of the web page into the phone display. Then click **ListPicker** in the Components list and edit the properties to the right. Change the text field to **Connect to BT**. Your button should now resemble that shown in Figure 14-19.



Figure 14-19: The completed button

Now it's time to ensure everything is centered in the display. Click **Screen1** in the Components list and change the AlignHorizontal setting to **Center**. The items on the display should now be aligned in the center, as shown in Figure 14-20.



Figure 14-20: The items are now centered.

Next, you'll create a table of eight buttons, one to turn on and off each of the four digital output pins. Click the **Layout** list in the Palette and then drag **TableArrangement** into the display. This should leave a box below the connect button, as shown in Figure 14-21. Also change the number of rows in the TableArrangement properties to four.



Figure 14-21: The table has been inserted.

Now for the buttons: click and drag eight buttons from the User Interface list on the left into the gray table in the phone display.

Thanks to TableArrangement, you can easily arrange them in four rows of two columns, as shown in Figure 14-22.

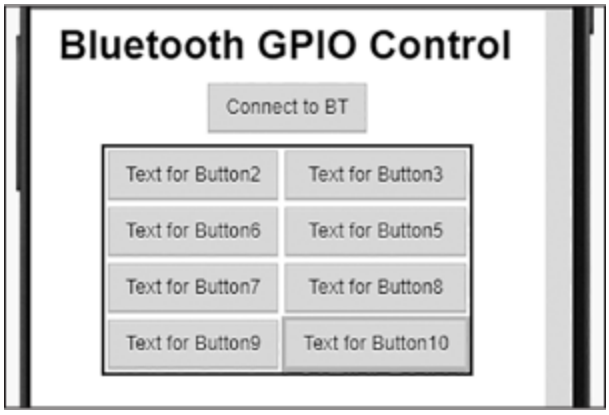


Figure 14-22: Working with the buttons

Click **Button2** in the Components list and change the text to **D8 On**. Work through each of the other seven buttons, changing their text properties, until they match the buttons in Figure 14-23.



Figure 14-23: The buttons are now complete.

To make the buttons easier to work with in the later steps of app development, rename the buttons in the Components window: click each button in turn, click **Rename**, and change the name of the button to its function. For example, change the name of Button2 to **D8 On**, as shown in Figure 14-24.

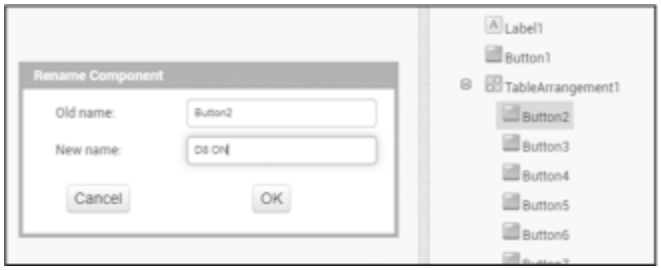


Figure 14-24: Editing the button references

Next, we need to add a reference to Bluetooth. This is a virtual link between the app and the phone's Bluetooth hardware. Click

the **Connectivity** tab in the Palette, drag **BluetoothClient** into the main window, and place it on top of the phone. The Bluetooth icon should appear below the phone, as shown in Figure 14-25.

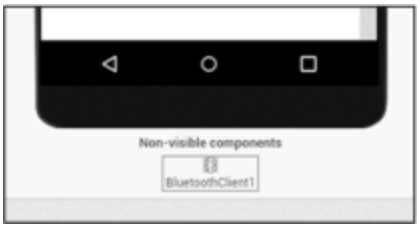


Figure 14-25: Bluetooth added to the app design

Finally, you can edit the text in the top line of the app or remove it by editing the Screen1 item in the Components menu. To change the name of the app to something more useful or interesting to your needs, edit the title field that appears at the bottom of the Properties window, or uncheck the **TitleVisible** box if you don't want to show the app's title.

At this point, you've created the app design and interface in the Designer window. You'll now begin the second stage of the app design: determining what actions the app will take upon user interaction. Start by clicking the **Blocks** button at the top right of the display, as shown in Figure 14-26.



Figure 14-26: Switching to the Blocks editor

In MIT App Inventor, the Designer screen is used to build the user interface of your app and the Blocks page is used to build the code that defines how the app works. Instead of writing code, you can use the Blocks editor page to drag and drop various “blocks” together to form sequences of instructions or activates based on user input. Each block represents a type of instruction, a function, or an activity. You can move between the two design screens by alternating between the **Designer** and **Blocks** buttons.

I’ll now show you how to create the required block structure to enable app operation. In the Blocks editor page, click **ListPicker1** in the Blocks menu. A list of blocks will appear, some of which are shown in Figure 14-27.



Figure 14-27: Various blocks

Drag and drop the **when ListPicker1 BeforePicking** block from the **Blocks** menu on the left into the Viewer window. Once again, click **ListPicker1** in the Blocks menu and then drag and drop a **set ListPicker1 elements to** block into the **when** block from the previous step. It should click into the **when** block.

To delete an unwanted block during this process, simply drag it over the garbage can at the bottom right of the Viewer window.

Next, click the **BluetoothClient1** item in the Blocks menu and drag the **BluetoothClient1 AddressesAndNames** item so it clicks into the last added item.

You now need to create the set of instructions to connect to the selected Bluetooth device as selected by the user. Assemble the blocks together. You can find the **if then** block in the Control section of the Blocks menu.

Now you need to configure the operations for each of the eight control buttons. To mirror the way Project #42 controlled the LEDs, you want the app to send a 0 through 3 to turn on D8 to D11, respectively, and send a 4 through 7 to turn them off. Click the **D8** button in the TableArrangement1 list in the Blocks menu and then drag and drop a **when D8_On.click** block into the viewer window.

Drag and drop **call BluetoothClient1. SendText text** block from the BluetoothClient1 list into the **when** block.

To complete the button action group, click the **Math** item in the Blocks menu and drag the first block (which just holds a 0) into the notch to the right of the word **text**.

This group of blocks will activate when the user presses the button D8 On; this will send the text character 0 out via Bluetooth.

You'll need to create another seven of these groups so that you end up with eight in total to match the buttons. To save time,

you can right-click over the first block in a group and select **Duplicate** to repeat the group you're editing. The other groups must refer to different buttons. Click the **D8_ON** and a list of all the available buttons should drop down, as shown in Figure 14-28.

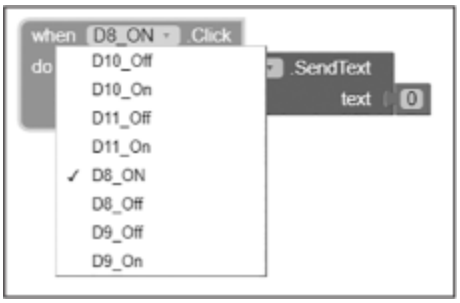


Figure 14-28: The button group selection

Ensure you have changed this setting in each of the eight button groups to match the eight buttons and then change the value of the text character being sent for the rest of the buttons. That is, pressing D8 off should send 4, D9 on should send 1, and so on. You can just click over the 0 in each group and enter a new value. When you've finished, your screen should resemble that shown in Figure 14-29.

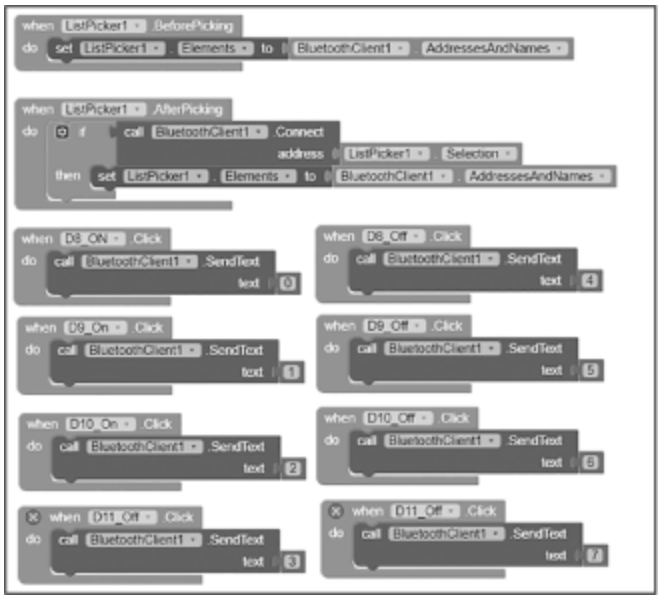


Figure 14-29: The completed control block groups

Your app is now complete and ready for installation on your Android phone or tablet. Save your project using the Projects menu (in case you want to alter it later) and then select **Android App (.apk)** in the Build menu. The App Inventor should compile the instructions and, after a minute or two, present you with a download button and QR code, as shown in Figure 14-30.



Figure 14-30: Download your app!

You can then download and save the app's *.apk* file to load it onto your device or just scan the QR code with your device's camera. When prompted, visit the URL decoded from the image and go through the download and installation prompts. Click **Yes** or **Agree** to any dialogs that Android may produce.

At long last, your app should pop up. Drag it onto the home screen, as shown in Figure 14-31.

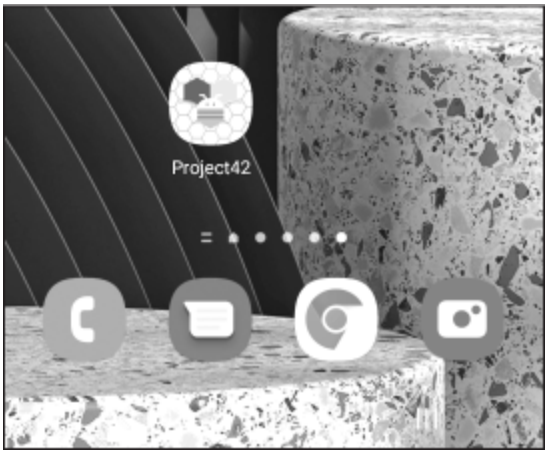


Figure 14-31: Your app installed on a phone

Congratulations! You've made an Android app. Now to give it a test: ensure you have the Project #42 hardware correctly wired up and then open your app. Turn on your device's Bluetooth function and then touch the app's **Connect to BT** button. You should be presented with a list of paired Bluetooth devices. Select the HC-05 listing, and the app should return to the main page.

You should now be able to press each button to turn on or off the digital output pins on the Arduino, with the LEDs reflecting the Arduino's status. If the app isn't working, first go back to Project #42 and ensure that the hardware can be controlled via the terminal app or a Bluetooth-enabled PC. Failing that, review the app setup in the App Inventor. You can also download the AI project file in the *Project #43* folder of the downloadable files on the book web page, *https://nostarch.com/arduino-arduinians*.

This simple example demonstrates just one way you can use the Android App Inventor to make your own remote-controlled projects with professional user interfaces. If you like, spend more time experimenting with App Inventor. I know you'll come up with some interesting use cases of your own.

Let's learn one more way to keep your projects user-friendly by making some modifications to the Bluetooth module.

Updating the Module Name and PIN

Now that you've built an Android app, the final step is to update the module's name and PIN/password for additional security, as every unit generally ships from the factory with a PIN of 1234. To do so, you must change the module's operation mode to AT command mode, allowing you to send AT commands to update and review various parameters (in the same way you would with a cellular shield for Arduino).

To switch to AT command mode, wire up the module to your Arduino as shown in Figure 14-3 and then add the connection from 3.3 V to the Bluetooth module's EN pin. This extra connection is required only when altering the parameters—remove or disconnect the connection for normal use.

Next, enter and upload the sketch in Listing 14-2, which enables communication between the Arduino Serial Monitor and the Bluetooth module.

```
  #define PIN_BT_TX 3
  #define PIN_BT_RX 2
  #include <SoftwareSerial.h>
❶ SoftwareSerial BT(PIN_BT_RX, PIN_BT_TX);

  void setup()
```

```
{
  ❷ BT.begin(38400);
  ❸ Serial.begin(9600);
}

void loop()
{
    if (BT.available())
        Serial.write(BT.read());
    if (Serial.available())
        BT.write(Serial.read());
}
```

Listing 14-2: Configuring the HC-05 Bluetooth module

This sketch first sets up a serial connection between the Arduino and your Bluetooth module ❶. The serial speed for the Bluetooth module in AT mode is set at 38,400 baud ❷. The serial connection between your PC and the Arduino stays at 9,600 baud ❸. The module's LED should blink slowly to indicate that AT command mode is operating.

Open the Serial Monitor and ensure that both NL (for "new line") and CR (for "carriage return") are selected in the menu next to the speed. Enter **AT** and press **Send**. The Bluetooth

module should reply `OK` in the Serial Monitor. If it doesn't, double-check your wiring and that the LED is blinking slowly.

Just like network devices, each Bluetooth module has a unique media access control (MAC) address. Send **`AT+ADDR?`** in the Serial Monitor, which should return your module's MAC address.

My unit returned a MAC address of 00:21:06:08:30:BF. To change the four-digit pairing PIN, send **`AT+PSWD =`*`xxxx`***, where ***`xxxx`*** is the new four-digit PIN, such as 8675, for example. You can then test the new PIN by forgetting the device in the Bluetooth menu and then searching for it again.

To change the Bluetooth's name from HC-05 to something more descriptive, send **`AT+NAME =`*`NewName`***, where ***`NewName`*** is the name you'd like the module to transmit when in operation. For example, if you were building a project that used a relay to operate a relay in your garage, you might change its name to GarageRelay by sending **`AT+NAME =GarageRelay`**. The new name should show up on your device's Bluetooth screen when pairing, as shown in Figure 14-32.

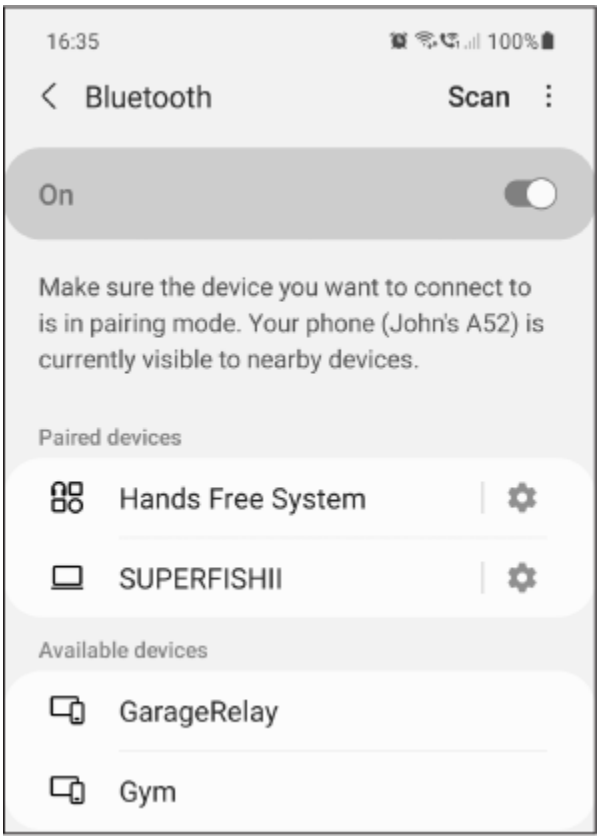


Figure 14-32: Results of changing the module name

Changing the module's name can be useful when working with several Bluetooth-equipped projects in the same area, allowing you to easily keep track of which devices you'd like to pair with. If you'd ever like to reset the module's name and pin back to HC-05 and 1234, send **`AT + ORGL`** to reset the unit to factory default. You might do this if your project was temporary or if you're passing the module on to another person.

Moving On

In this chapter, you learned several ways to use serial Bluetooth modules with your Arduino projects. You can now use Bluetooth for wireless remote control, capturing data from an Arduino to a computer via Bluetooth and building your own remote-control apps for Android devices. In the future, you can use these skills in plenty of applications, such as remote-control relay boards, logging data from sensors in hard-to-reach places, or building customized remote-control Christmas lights.

In the next chapter, you'll learn to reduce your Arduino's power consumption.

15

ENERGY EFFICIENCY FOR PORTABLE PROJECTS



Arduino-based projects are generally powered by AC mains adapters or USB power sources without much thought about power consumption. For portable projects, however, it's important to reduce energy consumption to increase the battery life for a longer runtime. This includes projects such as portable games or sensor and data-logging projects that don't have access to external power.

This chapter covers various methods of improving your Arduino project's power efficiency. You'll learn to:

- Measure the power consumption of your Arduino projects

- Estimate project battery life
- Reduce Arduino power consumption with software- and hardware-based methods
- Build low-power data-logging devices

You'll also learn some useful tools and methods to work with your Arduino such as the Arduino library for the DS3231-type clock IC module, sound sensor modules, and a USBasp device to upload sketches to Arduino boards and compatible circuits.

Basic Principles of Electricity

Portable projects are always, in some form, battery powered. The less power you use, the longer your project will last between battery charges or replacements. This chapter aims to teach you various ways to enable your projects to last longer when away from wired power. Before moving forward, let's review some basic principles of electricity to be sure you have the conceptual foundation to measure power consumption and determine battery life later in this chapter.

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: current, voltage, and power.

Current

The flow of electrical energy through a circuit is called the *current*. Electrical current flows through a *circuit* 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 (AC)*.

In some circuits, the negative side is called *ground (GND)*. Current is measured in *amperes (A)*, also known as *amps*. Smaller amounts of current are measured in *milliamps (mA)*, where 1,000 milliamps is equal to 1 amp.

Voltage

Voltage is a measure of the difference in potential energy between a circuit's positive and negative ends. It is measured in *volts (V)*. 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 incandescent

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:

Power (W) = Voltage (V) × Current (A)

You'll use this formula in calculations completed later in this chapter. Next, I'll introduce you to types of power consumption and how that power is measured.

Measuring Power Consumption

The appropriate method of determining how much power your Arduino project is using at a given moment depends on how the power is connected to your Arduino. To calculate power consumption in the examples and projects later in this chapter, you'll need to be able to pick an appropriate method of measuring power consumption for the project at hand. In this section, I'll show you various methods of measuring your Arduino project's power consumption.

USB-Sourced Power

If your project is powered via a USB-based power supply, the easiest way to measure power consumption is to use an inline device such as a USB Doctor, shown in Figure 15-1. The USB Doctor is a small device that measures voltage and current passing through a USB cable.



Figure 15-1: A USB Doctor, an inline power measurement device

USB Doctors are connected inline between the Arduino and the power source (such as a PC, wall wart, or battery bank) and will alternate between showing you the current and voltage being used on the digital display. The main value of interest is the current, as you use this in the calculation to determine battery life.

Wired Power

If a project has *wired power consumption*, that means it connects to a power source via the Arduino's DC socket or directly into the 5V and GND pins on the board or in your circuit. To measure power consumption for such projects, use a multimeter with a current setting.

Unlike measuring voltage, where you can just probe across two points, measuring current requires the power to enter the multimeter via the current socket and exit via the COM socket. To quickly determine if your multimeter has this capacity (most do), check whether it has a socket labeled mA or A.

For example, consider the multimeters in Figure 15-2. Starting with the leftmost multimeter and moving right, you'd use the A mA and COM sockets; the 10A and COM sockets; the 20A and COM sockets; and the 10A mA and COM sockets.



Figure 15-2: Multimeters with current-measuring capability

If you're presented with both an A-style and an mA-style socket, use the amps socket (A) first. If the value displayed falls within the range of the mA setting, you can then use that socket and range to get a more precise measurement.

Don't forget to set your multimeter to the current mode that matches the socket being used. If in doubt, review the instruction manual with your multimeter. Once you set your multimeter to the current range, the probes need to be connected into the power circuit.

If you're using the DC socket power supply, an easy way to measure is to use a pair of DC terminal connectors such as PMD Way parts 116101M and 116101F, examples of which are shown in Figure 15-3. You can then run wires between the two units and tap in with your meter probes. If your multimeter returns a negative value, swap the probes across to change the polarity.



Figure 15-3: DC terminal connectors

However, if you're using direct connection to the Arduino's 5V and GND pins, you can easily probe between the power supply positive end and the Arduino's 5V pin.

Arduino Power Consumption

Now that you have an understanding of the types of measurement tools available, I'll give you a baseline idea of consumption. Let's examine the current consumption values I measured with a multimeter from a standard Arduino Uno R3 board with the onboard D13 LED turned on. You can then follow along and determine consumption for your own project.

Table 15-1 lists the current in milliamps along with the power source used to power the Arduino. When you measure the current draw for your own projects using the methods described in the previous section, your measurements may differ from mine a little depending on the brand of Arduino or compatible board you're using.

Table 15-1: Arduino Current Draw at Various Supply Voltages

Power source	Current draw (mA)
12 V via the DC socket	47.9
9 V via the DC socket	47.9
5 V via the 5V pin or USB socket	44.5
4.8 V via the 5V pin	42.5

Bear in mind these values are for the Arduino alone. Whatever you add to the project will increase the current draw. I've measured the current at 4.8 V for this example because this is a common voltage when using four AA rechargeable cells (as they

are 1.2 V, not 1.5 V) or four AA disposable cells that are almost at the end of their life span.

The Arduino can happily operate on these lower voltages. However, if you're using the microcontroller's internal ADC to measure analog signals, you'll need to take the reduced operating voltage into account, as the ADC's reference voltage is assumed to be 5 V and not the actual operating voltage. Instead, use the external reference voltage pin and the `analogReference()` function. If you're not familiar with this process, you can learn more about this in Chapter 4 of *Arduino Workshop*, 2nd edition.

Now that you have an idea of an Arduino's usual power consumption when running without any external parts, I'll show you how to calculate approximate battery life based on your battery capacity and project current draw.

Estimating Battery Life

To determine the battery life for your project, you need to determine how much current over time (measured in milliamp hours) is used by the project. To do this, first measure the current your project draws for a moment from the connected power supply. Once you have a current figure for consumption, such as 65 mA, your project should draw that amount of

current for one hour. This becomes the consumption in milliamp hours, or mAh. You can then consider the consumption to be 65 mAh.

If the Arduino activates external devices such as a motor or servo, don't forget to take that extra current into account. If your project runs a motor for, say, 1 minute every hour, measure the current when the motor is both on and off and then determine the total current used for one hour. For example, if the project ran a motor at a constant speed for 10 minutes, if the current was measured at 100 mA, and if the project ran for 50 minutes at 60 mAh, then the consumption would be 100 mA × (10 / 60 minutes) + 60 mA × (50 / 60 minutes) = 66.6 mAh.

Next, consider the battery type your project uses. For example, if you have a 20 Ah (Amp hours, more often advertised as 20,000 mAh) USB battery bank, as shown in Figure 15-4, your project should last for about 300 hours, since 20,000 / 66.6 = 300.3003. If your battery is rated in Ah, multiply that value by 1,000 to convert to mAh.



Figure 15-4: A typical USB battery bank

Some battery banks will turn off if there isn't enough current draw, as part of a power-saving measure. Shop for units that have an always-on mode so they don't turn off if your Arduino's power consumption drops to a very low level.

Aside from battery banks, another popular and simple method of powering portable projects is to use nickel–metal hydride (NiMH) AA rechargeable cells in groups of four. This gives a supply voltage of 4.8 V DC, which, again, works fine to power an Arduino. You would use a four-AA cell holder to hold the cells, as shown in Figure 15-5, and then directly wire this to the Arduino's 5V and GND pins.

The individual NiMH cells should be marked with their capacity, such as 2,000 mAh. With a fully charged set of 4 cells in new condition, your 66.6 mA project would run for about 30 hours.

If you only have disposable AA cells available, place a diode such as a common 1N4001 in series between the positive on the battery pack and the 5V pin on the Arduino. The anode connects to the battery pack positive output, and the cathode connects to the 5V pin on the Arduino. This should drop the voltage of a new pack of cells from around 6 V DC down to around 5.3 V DC, which your Arduino can comfortably use.



Figure 15-5: A holder for four AA-cell batteries

Batteries such as NiMH rechargeable units and disposable cells don’t offer a constant voltage throughout their life—rather, the voltage will drop toward the end of their discharging cycle. Just

to make things more complex, disposable battery life can vary depending on the amount of current being drawn at any one time. For example, the graph shown in Figure 15-6 shows that the more current is discharged from an Energizer-brand alkaline-type AA cell, the less capacity there is for use.

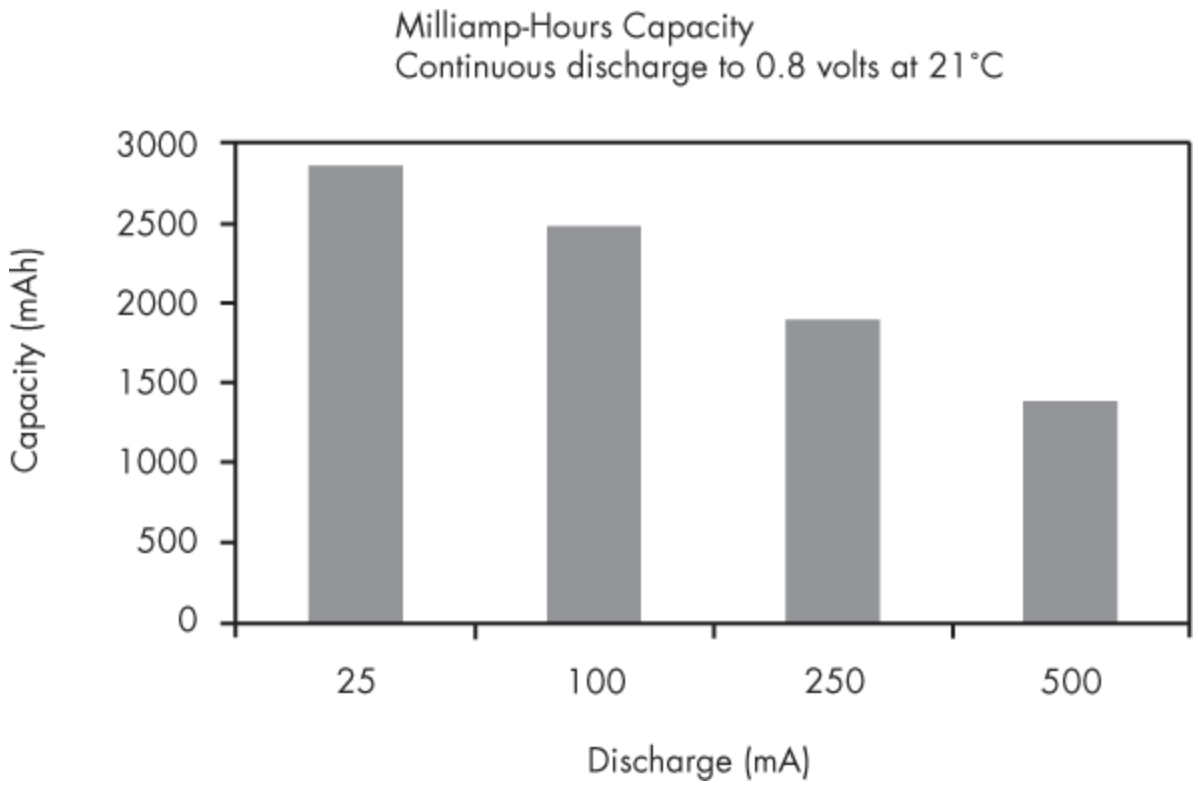


Figure 15-6: Capacity versus discharge rate for a disposable Energizer-brand alkaline AA-cell battery

To get the most accurate battery life estimate possible for your project, locate and review the data sheets of the battery type you're interested in. They will offer discharge-rate graphs, performance at various temperatures, and more interesting information that helps you make an informed decision.

AVOIDING 9 V BATTERIES

Never power your project using 9 V, 216-type batteries via the DC socket, as shown in the image at right.

The capacity of these batteries, generally between 250 and 670 mAh, is much smaller than that of AA cells. Moreover, a lot of precious, expensive energy is wasted during the 9 V to 5 V conversion by the regulators on the Arduino.



Now that you know how to measure your project's power consumption and can choose between various portable power options, I'll explain various methods of reducing power consumption. You'll put these methods into use in the following projects.

Software-Based Methods to Reduce Power Consumption

To use low-power functions with your Arduino, you'll need to install a library. Open the IDE's library manager and search for **low-power rocket**; then install the Low-Power library by Rocket Scream Electronics, as shown in Figure 15-7.



Figure 15-7: Low-Power library installation

In the following sections, I'll show you two ways to use this library to reduce power consumption. You'll first practice programming Power Down/Wake mode in the following homework and then move on to testing Wake on Interrupt mode. The examples and projects in this chapter use the Arduino Uno or compatible board with an ATmega328 microcontroller.

Power Down/Wake Periodic Mode

The Power Down/Wake Periodic mode is useful when you need your Arduino to perform a certain task on a regular basis and then stop drawing current once the task is complete—for example, to wake up and check the status of something or to log data and then go back to sleep. To use this mode, include the library in your sketch using the following statement:

```
#include "LowPower.h"
```

You can use this function to turn off the ADC and the microcontroller's brown-out detector with the `ADC_OFF` and `BOD_OFF` parameters as shown next. When you want the Arduino to go to sleep with all power-saving features turned on, use the following function in your sketch:

```
LowPower.powerDown (SLEEP_8S, ADC_OFF, BOD_OFF);
```

There are several parameters in that function, the first being the duration of idle. In this example, I've set the duration to eight seconds with `SLEEP_8S`. However, you can also use one of the alternative delays:

`SLEEP_15MS` 15 milliseconds

`SLEEP_30MS` 30 milliseconds

`SLEEP_60MS` 60 milliseconds

`SLEEP_120MS` 120 milliseconds

`SLEEP_250MS` 250 milliseconds

`SLEEP_500MS` 500 milliseconds

`SLEEP_1S` 1 second

`SLEEP_2S` 2 seconds

`SLEEP_4S` 4 seconds

`SLEEP_8S` 8 seconds

The following simple sketch uses this function to shut down the Arduino for 8 seconds, then repeats the shutdown again, and then runs at full power for 16 seconds. This allows you to quickly check power consumption at both full power and while operating in power-saving mode with a multimeter.

Connect your Arduino Uno to your PC and upload Listing 15-1. Once it's uploaded, connect your USB power monitor or

multimeter as described earlier in the chapter to watch the power consumption change.

```
❶ #include "LowPower.h"

  void setup() {}

  void loop()
  {
      // Idle for 16 seconds
    ❷ LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF
      LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OF
      // Operate at normal power for 16 seconds
    ❸ delay(16000);
  }
```

Listing 15-1: Testing the Power Down/Wake Periodic mode

The sketch includes the library ❶ and then runs the idle function twice with all parameters set to `OFF`, with an eight-second delay ❷. Next, the sketch introduces another delay, in which the Arduino runs at full power ❸.

If you measure the current of this circuit, the consumption should drop from around 42 mA on full power to around 30 mA

when in power-down mode, a consumption drop of around 12 mA. That's a 28 percent reduction in power use when your project is in power-down mode.

I quote the figures for this example as "around 42 mA" and so on because consumption will vary within a range of 1 or 2 mA lower or higher than the figures quoted, depending on the brand of Arduino or compatible board you are using. It may also vary depending on your own test equipment. To keep things simple, I've stuck to whole numbers and average consumption.

Before putting power saving to use in a practical example, I'll introduce a convenient library for the DS3231 real-time clock IC as you'll use it in a following project.

The DS3231 Real-Time Clock Library

The DS3231 real-time clock modules provide a convenient and accurate way to track time and date information, though using them does require a lot of code. In this section, I'll show you how to write and read time and date data to and from the DS3231 in an easier way using a DS3231 library.

First, open the Arduino IDE, enter the Library Manager, and search for **DS3231 by Andrew Wickert**.

Connect the real-time clock module to your Arduino Uno, as shown in Figure 15-8, and then connect your Uno to your PC with the USB cable.

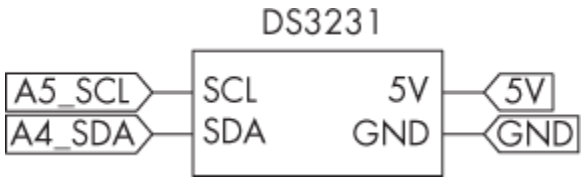


Figure 15-8: Connections between the DS3231 and Arduino Uno

Next, enter but don't upload Listing 15-2.

```
❶ #include <DS3231.h>
  #include <Wire.h>

  DS3231 RTCmodule;

❷ bool century = false;
  bool h12Flag = false;
  bool pmFlag = false;
  int hh, mm, ss, dd, dow, mo, yy;

  void setTime()
  {
    ❸ RTCmodule.setClockMode(false); // Set to 24-h
      RTCmodule.setYear(22);
      RTCmodule.setMonth(1);
      RTCmodule.setDate(28);
```

```
    RTCmodule.setDoW(6); // Sunday = 1
    RTCmodule.setHour(14);
    RTCmodule.setMinute(5);
    RTCmodule.setSecond(30);
}

void showTime()
{
    // Get data from RTC.
 ❹ dd = RTCmodule.getDate();
    dow = RTCmodule.getDoW();
    mo = RTCmodule.getMonth(century);
    yy = RTCmodule.getYear();
    hh = RTCmodule.getHour(h12Flag, pmFlag);
    mm = RTCmodule.getMinute();
    ss = RTCmodule.getSecond();

    // Send information to Serial Monitor
 ❺ switch(dow)
    {
        case 1: Serial.print("Sunday "); break;
        case 2: Serial.print("Monday "); break;
        case 3: Serial.print("Tuesday "); break;
        case 4: Serial.print("Wednesday "); brea
        case 5: Serial.print("Thursday "); break
        case 6: Serial.print("Friday "); break;
        case 7: Serial.print("Saturday "); break
    }
```

```
    Serial.print(dd, DEC);
    Serial.print("/");
    Serial.print(mo, DEC);
    Serial.print("/");
    Serial.print(yy, DEC);
    Serial.print(" | ");
    Serial.print(hh, DEC);
    Serial.print(":");
    if (mm < 10) // Check for leading 0 on minut
    {
        Serial.print("0");
    }
    Serial.print(mm, DEC);
    Serial.print(":");
    if (ss < 10) // Check for leading 0 on secon
    {
        Serial.print("0");
    }
    Serial.println(ss, DEC);
}

void setup()
{
    Serial.begin(9600);
    Wire.begin();
  ❻ // setTime();
}
```

```
void loop()
{
    showTime();
    delay(1000);
}
```

Listing 15-2: Testing the DS3231 library

The sketch includes and initializes the required libraries ❶ and then declares the required global variables ❷. These variables are used for storing RTC data and settings. The `setTime()`function is used to set the time and date to the RTC. The sketch uses `false` ❸ for 24-hour time, but you can also enter `true` for 12-hour time. Otherwise, enter all the time and date data in one- or two-digit format. The value for `setDoW` is for the numerical day of the week, with 1 being Sunday and 7 being Saturday.

To retrieve the data from the RTC and display it on the Serial Monitor, the `showTime()`function stores the data in turn from the RTC into variables ❹ and then starts displaying it on the Serial Monitor ❺, starting with a `switch…case` function to show the day of the week.

Set the time by uncommenting the function at ❻ and updating the data in the `setTime()` function to match the current time and date when you upload the sketch. After uploading the sketch, comment this line out again so the time isn't reset on every reset of the Arduino.

Finally, … normal clock display. The Serial Monitor will output your current time and date.

You'll use the DS3231 library in the upcoming project.

Project #44: Creating a Low-Power Periodic Data Logger

In this project, you'll create a data logger that records temperature, humidity, and the time and date to an SD card around once per minute. Between these times the Arduino will go into idle mode to conserve power. This type of project is useful for long-term data logging—for example, charting the weather data over a season or several months.

For this project, you'll need the following hardware:

- An Arduino Uno or compatible board and USB cable
- A solderless breadboard
- Various jumper wires

- A BMP180 temperature and air pressure sensor board
- A DS3231 real-time clock module
- An SD or microSD card module and matching memory card

Assemble the circuit as shown in Figure 15-9.

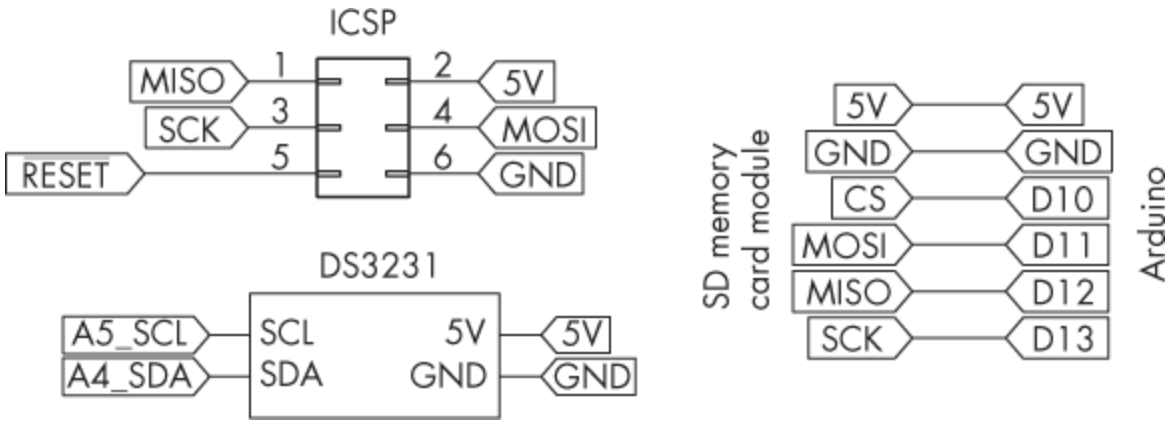


Figure 15-9: The circuit schematic for Project #44

Next, enter and upload the following sketch:

```
  // Project #44 - Periodic low-power data logging

❶ #include <SD.h>
  #include <LowPower.h>
  #include <DS3231.h>
  #include <Wire.h>
  #include <Adafruit_BMP085.h>

  Adafruit_BMP085 bmp;
  DS3231 RTCmodule;
```

```
❷ bool century = false;
  bool h12Flag = false;
  bool pmFlag = false;

  int hh, mm, ss, dd, mo, yy, temperature, pressur

  void setup()
  {
      delay(3000); // Allow time for SD card to be
      Serial.begin(9600);
    ❸ Serial.println("Initializing SD card…");
      pinMode(10, OUTPUT);
      // Check that the memory card exists and is
      if (!SD.begin(10))
      {
          Serial.println("Card failed, or not pres
          // Stop sketch
          return;
      }
      Serial.println("memory card is ready");
      Wire.begin();
      bmp.begin();
  }

  void setTime()
  {
      // Set to 24-hour time
```

```
    RTCmodule.setClockMode(false);
    RTCmodule.setYear(2022);
    RTCmodule.setMonth(1);
    RTCmodule.setDate(11);
    RTCmodule.setHour(17);
    RTCmodule.setMinute(11);
    RTCmodule.setSecond(8);
}

void logData()
{
    // Create the file for writing:
    File dataFile = SD.open("DATA.TXT", FILE_WRI
    // If the file is ready, write to it:
    if (dataFile)
    {
        // Get data from RTC:
        dd = RTCmodule.getDate();
        mo = RTCmodule.getMonth(century);
        yy = RTCmodule.getYear();
        hh = RTCmodule.getHour(h12Flag, pmFlag);
        mm = RTCmodule.getMinute();
        ss = RTCmodule.getSecond();
        // Write time and date to SD card:
        dataFile.print(dd, DEC);
        dataFile.print("/");
        dataFile.print(mo, DEC);
        dataFile.print("/");
```

```
        dataFile.print(yy, DEC);
        dataFile.print(" | ");
        dataFile.print(hh, DEC);
        dataFile.print(":");
        if (mm < 10) // Check for leading 0 on m
        {
            dataFile.print("0");
        }
        dataFile.print(mm, DEC);
        dataFile.print(":");
        if (ss < 10) // Check for leading 0 on s
        {
            dataFile.print("0");
        }
        dataFile.print(ss, DEC);
        dataFile.print(" | ");

        // Write temperature data:
        temperature = bmp.readTemperature();
        pressure = int(bmp.readPressure() / 100)
        dataFile.print(temperature, DEC);
        dataFile.print(" C - ");
        dataFile.print(pressure, DEC);
        dataFile.println(" hPa");
        dataFile.close();
    }
}
```

```
void loop()
{
 ❹ // setTime(); // Used to set time in RTC if r
   for (int snooze = 0; snooze<8; snooze++)
   // Idle for 64 seconds
   {
    ❺ LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD
   }
   logData();
}
```

Once again, the sketch includes and initializes the required libraries ❶ and declares the required global variables that are used to store RTC data and settings ❷. It initializes and tests the SD memory card reader ❸ and then uses the `setTime()`function to set the time and date to the RTC. As in Listing 15-2, the sketch uses `false` for 24-hour time, but you can change this to `true` for 12-hour time. Be sure to replace the values in the `RTCmodule` functions to match the current time in your location.

The `logData()`function writes the date, time, temperature, and pressure to the SD memory card when called. Set the time and date in the DS3231 by uncommenting the function at ❹ so that the changes you've made to the `setTime()` function take effect.

After uploading the sketch, comment this line again so the time isn't reset every time the Arduino resets.

Finally, a `for` loop runs the 8-second power-down 8 times for a total of 64 seconds; then the Arduino wakes up and logs the data ❺. At this point, the sketch loops again, putting the Arduino back into a powered-down state.

Figure 15-10 shows an example of the output text file. Your date and time will of course be different.

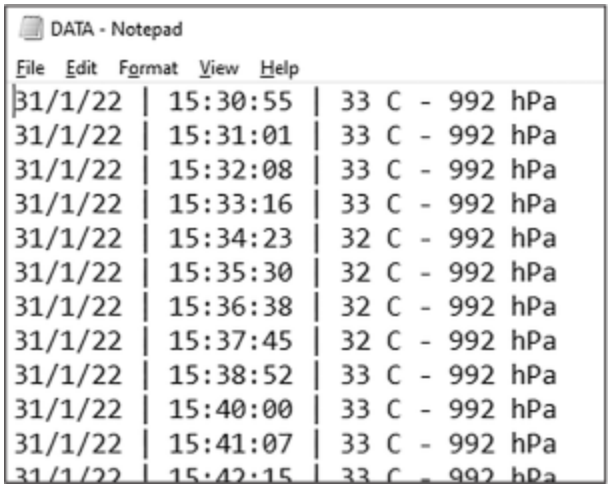


Figure 15-10: Example output from Project #44

If you measure the current of this circuit, the consumption should drop from around 51 mA on full power to around 38 mA when the project is powered down. There is a brief jump in consumption when the Arduino is writing data to the SD card. Setting this aside, the power used by the entire project drops by

about 10 mA. (The Arduino's own power consumption drops without affecting the devices to which it is connected.)

This may not seem like significant savings, but consider that this amounts to a 20 percent reduction in power consumption when in power-down mode, which takes between 97 to 98 percent of operation time. Moreover, when you're running from a battery, any saving is a good saving.

Wake on Interrupt Mode

There's a second software-based way to save power on your Arduino: Wake on Interrupt mode. Using this mode puts your Arduino to sleep until an interrupt is triggered, causing the Arduino to wake up and continue to run the sketch. After it completes the required tasks, it goes back to sleep, awaiting another interrupt.

NOTE

For more information about interrupts, review Chapter 7 of Arduino Workshop*, 2nd edition.*

To use this mode, include the library in your sketch with the following line of code:

```
#include "LowPower.h"
```

As each Arduino interrupt calls a function, you need to define a function to allow the sketch to compile. However, as you don't need to call any specific code when an interrupt is triggered, leave the function empty:

```
void wakeUp() {}
```

You must also declare the hardware interrupt pin as an input in `void setup()`, using pins D2 or D3.

In the main section of the sketch, attach the interrupt using the following:

```
attachInterrupt(pinNumber, interruptFunction, mode
```

Replace `pinNumber` with 0 to attach to digital pin D2 or with 1 to attach to digital pin D3.

In the `attachInterrupt()` function, substitute `mode` with one of the following four interrupt types:

`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.

The Arduino interrupt functions need a function to call once the interrupt has been triggered, but running code directly from a function attached to an interrupt will cause a recursive loop. To avoid this, we use a blank (also known as a *stub*) custom function—that is, a function without any code. For example, you could use this:

```
void interruptFunction() {}
```

Next, when you want to put the Arduino to sleep, use the following line:

```
LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_O
```

After the interrupt triggers the Arduino to wake up, detach the interrupt from the digital pin with the following function:

```
  detachInterrupt(0);
```

Next, run the code or call the required function to operate. After this operation is complete, the Arduino will go back to sleep.

I'll demonstrate this in the following simple example in which the signal at D2 changes from HIGH to LOW, the Arduino's onboard LED blinks once, and then the Arduino goes back to sleep. Connect your Arduino Uno to your PC, run a jumper wire from the 5V pin to the D2 pin, and upload Listing 15-3. Connect your USB power monitor or multimeter to watch the power consumption change.

```
❶ #include <LowPower.h>

❷ void interruptFunction(){}

  void setup()
  {
      ❸ pinMode(2, INPUT);
      ❹ pinMode(13, OUTPUT); // For demo LED
  }

  void loop()
```

```
{
    // start using interrupt pin
 ❺ attachInterrupt(0, interruptFunction, FALLING

    // Enter power down state with ADC and BOD m
    // Wake up when D2 is low.
 ❻ LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BO

    // Disable external pin interrupt on wake up
    // Not using interrupt anymore
 ❼ detachInterrupt(0);

    // Do something here when interrupt triggere
 ❽ digitalWrite(13, HIGH);
    delay(1000);
    digitalWrite(13, LOW);
}
```

Listing 15-3: Waking up on interrupt mode

The sketch includes the low-power library ❶, followed by a blank function, as required by the interrupt functions ❷. It then sets the digital pin used for interrupts as an input ❸ and configures the Arduino's built-in LED ❹.

The sketch also configures the interrupt type ❺ on to pin D2 with the name of the blank interrupt function and the type of event to call the interrupt. In this case, the event type is `FALLING`, which means a change of signal from 5 V to 0 V at D2.

Once the interrupt is configured, the sketch puts the Arduino to sleep ❻. Nothing else happens until the interrupt is triggered and the Arduino goes into full-power mode. At this point, the sketch disables the interrupt so that it won't be called again while the rest of the sketch is operating ❼.

At this point, the sketch turns the onboard LED on and off to demonstrate that the Arduino is operating normally ❽, but you could substitute another function here, if you like. Once the code has been executed, `void loop()` returns to the start, reactivating interrupts ❺ and putting the Arduino back to sleep ❻.

You can test this project by removing the jumper wire from the 5V pin to D2, which triggers the interrupt and brings the Arduino back to full-power mode. Then restore the jumper wire, and after the LED is activated and turned off, the Arduino should go back to sleep. In this example, power consumption drops from around 42 mA when the Arduino is awake to 28 mA when in sleep mode, a reduction of around 59 percent.

In the next project, you'll use this technique for a more serious application by logging noisy events detected by a sound sensor.

Inexpensive Sound Sensors

When building a project that must detect when a loud noise occurs, you can use adjustable sound level sensor, such as PMD Way part 596692, as shown in Figure 15-11.

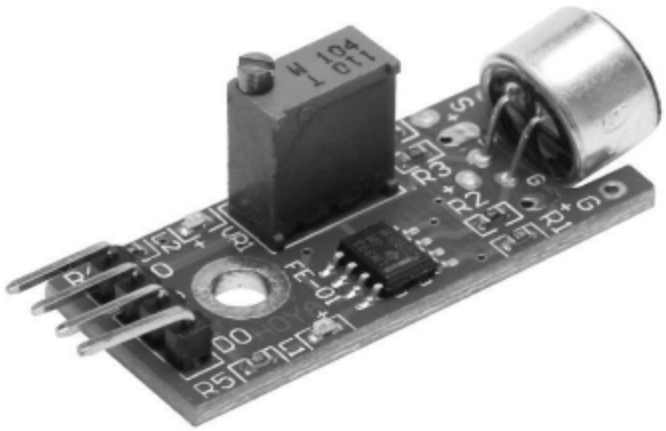


Figure 15-11: An adjustable sound level sensor

These sensors, which can be found at most retailers, have two types of output: analog and digital. In the following project, you'll use this sensor in digital mode. After you connect 5V and GND to the sensor's + and – (or G) pins, respectively, a 5 V signal will be sent to the digital output (DO) pin when the ambient sound reaches a certain level. An LED turns on when

the sound reaches the threshold level, which you can adjust using the trimpot on the sensor.

Project #45: Logging Interrupt Triggers

The goal of this project is to log the date and time of a sound event, such as a noisy neighbor or strong vibration in the surrounding environment. It uses a sound sensor module to trigger an interrupt, bringing the Arduino out of low-power mode, logging the date and time to an SD memory card, and then putting the Arduino to sleep until the next event. If you're not interested in logging sound events, you can use any type of sensor or switch that has a digital output.

For this project, you'll need the following hardware:

- An Arduino Uno or compatible board and USB cable
- A solderless breadboard
- Various jumper wires
- A sound sensor module
- A DS3231 real-time clock module
- An SD or microSD card module and matching memory card

Assemble the circuit as shown in Figure 15-12.

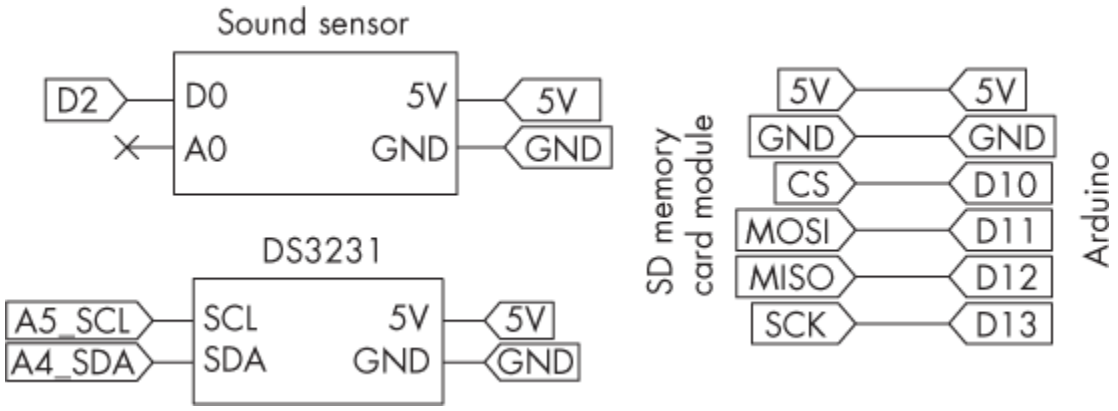


Figure 15-12: The circuit schematic for Project #45

Next, enter and upload the following sketch:

```
// Project #45 - Interrupt-triggered logging wit

#include <SD.h>
#include <LowPower.h>
#include <DS3231.h>
#include <Wire.h>
DS3231 RTCmodule;

bool century = false;
bool h12Flag = false;
bool pmFlag = false;

int hh, mm, ss, dd, mo, yy;

void setup()
{
    delay(3000); // Allow time for SD card to be
```

```
    Serial.begin(9600);
    Serial.println("Initializing SD card…");
    pinMode(10, OUTPUT);
    // Check that the memory card exists and is
    if (!SD.begin(10))
    {
        Serial.println("Card failed, or not pres
        // Stop sketch
        return;
    }
    Serial.println("Memory card is ready!");
    Wire.begin();
    pinMode(13, OUTPUT);
}

void interruptFunction() {}

void setTimeData()
// Use once to set the time and date in the RTC
{
    RTCmodule.setClockMode(false); // Set to 24-
    RTCmodule.setYear(2022);
    RTCmodule.setMonth(1);
    RTCmodule.setDate(11);
    RTCmodule.setDoW(2);
    RTCmodule.setHour(17);
    RTCmodule.setMinute(11);
    RTCmodule.setSecond(8);
```

```
}

void logData()
{
    // Create the file for writing:
    File dataFile = SD.open("DATA.TXT", FILE_WRI
    // If the file is ready, write to it:
    if (dataFile)
    {
        // Get data from RTC:
        dd = RTCmodule.getDate();
        mo = RTCmodule.getMonth(century);
        yy = RTCmodule.getYear();
        hh = RTCmodule.getHour(h12Flag, pmFlag);
        mm = RTCmodule.getMinute();
        ss = RTCmodule.getSecond();

        // Write time and date to SD card:
        dataFile.print(dd, DEC);
        dataFile.print("/");
        dataFile.print(mo, DEC);
        dataFile.print("/");
        dataFile.print(yy, DEC);
        dataFile.print(" | ");
        dataFile.print(hh, DEC);
        dataFile.print(":");
        if (mm < 10) // Check for leading 0 on m
        {
```

```
            dataFile.print("0");
        }
        dataFile.print(mm, DEC);
        dataFile.print(":");
        if (ss < 10) // Check for leading 0 on s
        {
            dataFile.print("0");
        }
        dataFile.print(ss, DEC);
        dataFile.println(".");
        // Close the file once the system has fi
        dataFile.close();
    }

}

void loop()
{
 ❶ attachInterrupt(0, interruptFunction, RISING
 ❷ LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BO
 ❸ detachInterrupt(0);
 ❹ logData();
}
```

By now you should understand the initial code in this sketch that sets up the low-power library, real-time clock, and SD memory card. The final lines of code configure the interrupt to

`RISING` ❶, so when the voltage at the Arduino's D2 pin rises from 0 V to 5 V (the digital output from the sound sensor), the sketch tells the Arduino to power down ❷. Once the sensor has been triggered, the sketch cancels low-power mode and disables the interrupt ❸ to prevent recursive interrupts. Finally, the sketch logs the time and date of the sound event to the SD card ❹, and the sketch repeats by activating interrupts ❶ and powering down the Arduino again in preparation for the next round ❷.

The power consumption for this project is around 28 mA when the Arduino is asleep and around 60 mA during the brief period in which the project is logging data to the SD card. That means this project is saving around 50 percent of power consumption when in sleep mode.

Hardware-Based Methods to Reduce Power

You can also reduce power consumption by making hardware modifications to your Arduino. The Arduino development platform was originally designed to be easy to use. While the standard Arduino Uno–type circuit serves that purpose well, more advanced users can improve power efficiency by

removing several components that are unnecessary for many projects.

You can remove the following pieces of an Arduino or compatible circuit if they won't be used:

- Four LEDs (power, D13, TX, and RX)
- Linear voltage regulator circuits for the onboard 5 V and 3.3 V power supplies (if you are powering your Arduino with 5 V DC)
- A comparator circuit that decides whether to use USB or DC socket power (if you are powering your Arduino with 5 V DC)
- A USB-to-serial interface microcontroller and circuit
- A reset button, if unnecessary

If you're prepared to make your own Arduino-compatible circuit and provide a suitable power supply, then it's easy to have your project use a lot less energy than a standard Arduino Uno. To demonstrate this, I'll show you how to make two versions of a bare minimalist Arduino-compatible circuit, one that operates on 5 V and the other on 3.3 V. You can build these circuits on a solderless or prototyping board and then build projects around them.

To build these projects, you'll first need to know how to upload sketches to your Arduino and minimalist circuits with the ICSP pins, rather than the USB interface.

Uploading Sketches with the ICSP Pins

Until now, you've most likely been uploading your sketches using the USB connection on your Arduino or compatible board. However, you can reduce the amount of circuitry you're using—and thereby make your projects more power efficient—by using the in-circuit serial programmer (ICSP) connection to the microcontroller instead.

The ICSP is the six-pin connector on the right side of your Arduino board, shown in Figure 15-13. When you're not uploading sketches, you can also use it as another connection to the SPI data bus.



Figure 15-13: ICSP header pins on the Arduino Uno

These six pins are arranged in a set configuration, as shown in Figure 15-14.

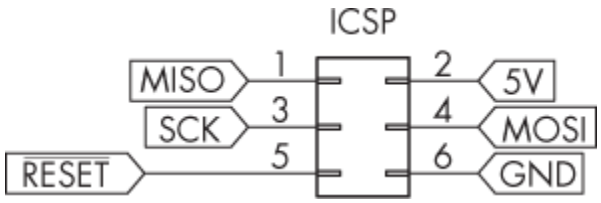


Figure 15-14: ICSP header pins layout

Pin 1 is usually marked on the PCB with a number 1 or a small dot close to the top-left pin, as you can see in Figure 15-13. To upload sketches to your circuit, you'll need an external programmer that connects to these pins, called a USBasp device. These open source devices were created by Thomas Fischl and require USB drivers only when used with a Windows-based PC. In addition to allowing you to reduce power usage, USBasps also allow you to upload sketches to your Arduino Uno or compatible board if its USB interface has been damaged or destroyed.

NOTE

USBasps also come in handy if you're interested in programming AVR microcontrollers without the Arduino environment, as described in my book AVR Workshop *(No Starch Press, 2022).*

USBasps vary cosmetically, but all have the same function. Figure 15-15 shows two examples. The programmer at the top of the figure is a generic unit from the supplier PMD Way listed in the book parts list, and the lower programmer is from Freetronics in Australia.

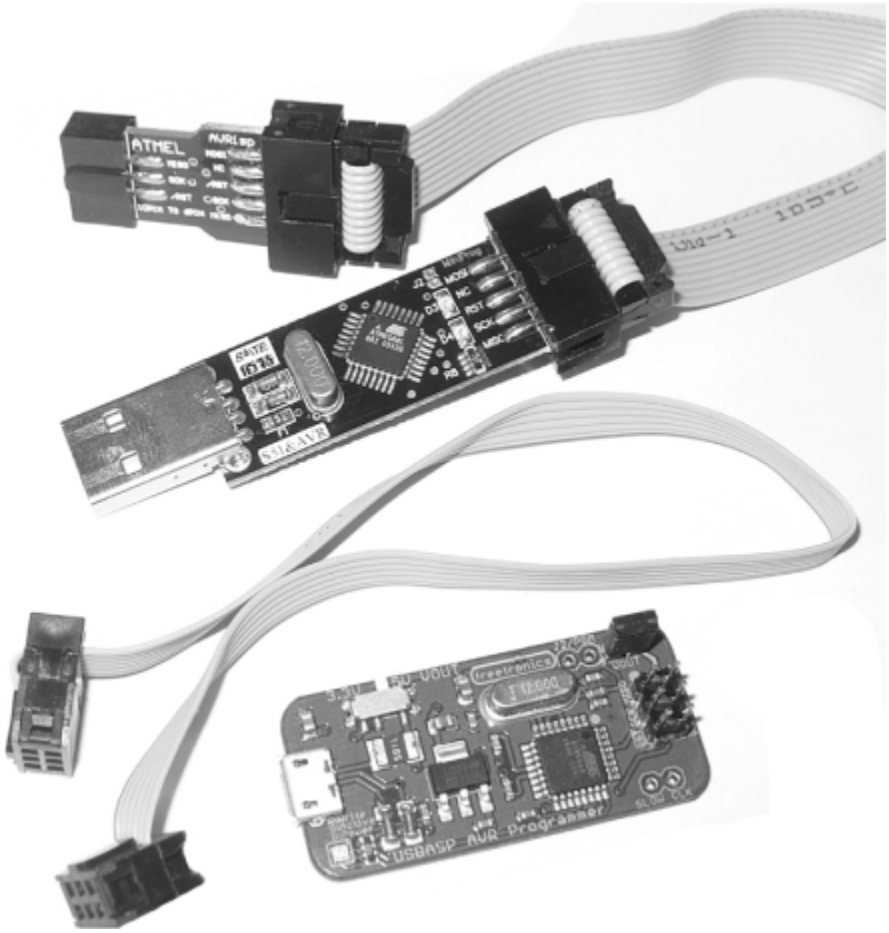


Figure 15-15: Example USBasp programmers

When shopping for your USBasp programmer, be sure you buy one that comes with a 6-pin (rather than 10-pin) ribbon cable. Windows users should consult the supplier for information on

how to install the programmer. Linux and Mac users can simply insert into a USB socket and the USBasp should be ready for use in a few moments.

Quickly test your USBasp to ensure it works before moving on to the minimalist circuits that you'll build in the next section. Start by connecting it to your Arduino Uno or compatible board, as shown in Figure 15-16.

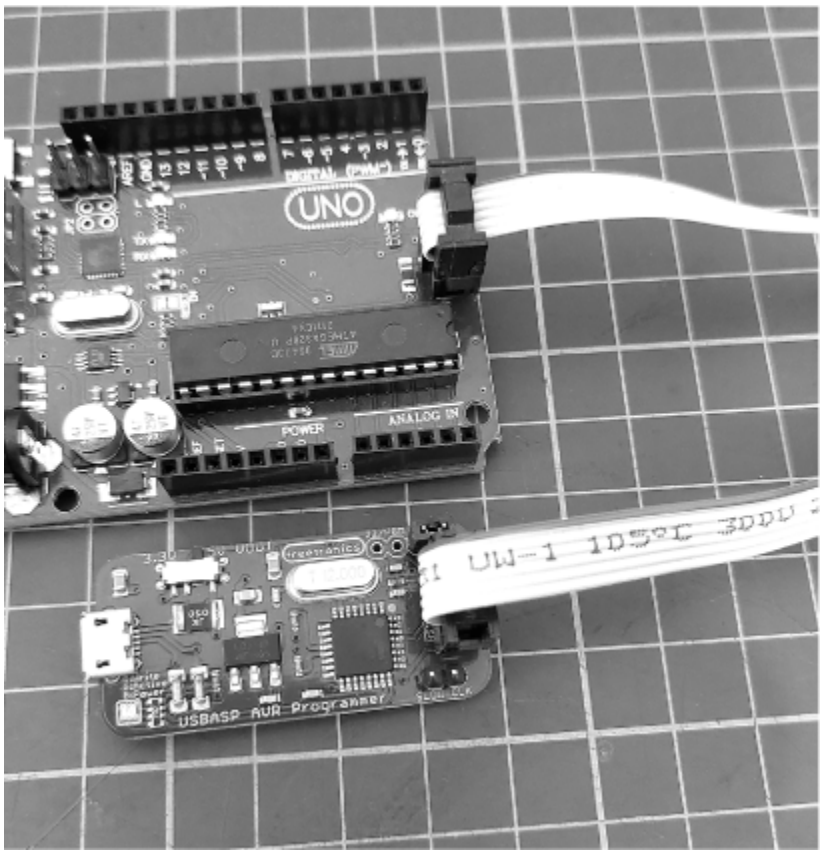


Figure 15-16: A USBasp programmer connected to an Arduino Uno

The USBasp provides power to the Arduino, so you don't need to connect any other power source. Next, connect the USBasp to your PCB via the included USB cable. Open the Blink sketch in the Arduino IDE and then select **Tools ▸ Programmer ▸ USBasp**.

Finally, to upload the sketch, select **Sketch ▸ Upload Using Programmer**.

The sketch should finish uploading to the Arduino, and the onboard LED should blink on and off, confirming that your USBasp is working correctly. If so, you're ready to build a minimalist low-power 5 V Arduino circuit.

Project #46: Building a Minimalist 5 V Arduino Circuit

In this project, you'll build the most minimalist Arduino-compatible circuit possible that can operate from around 5 V DC. You can expand on this template to build your own future low-power Arduino-compatible projects.

You'll need the following hardware:

- A USBasp programmer and matching USB cable
- A solderless breadboard
- Various jumper wires
- An ATmega328P-PU microcontroller

- One 0.1 µF polyester capacitor (C1)
- Two 22 pF ceramic capacitors (C2, C3)
- One 16 MHz HC49S crystal
- An ICSP breadboard adapter
- One 560 Ω, 0.25 W, 1 percent resistor
- One LED

Assemble the circuit as shown in Figure 15-17. Try to keep the crystal, C2, and C3 as close as possible to the microcontroller.

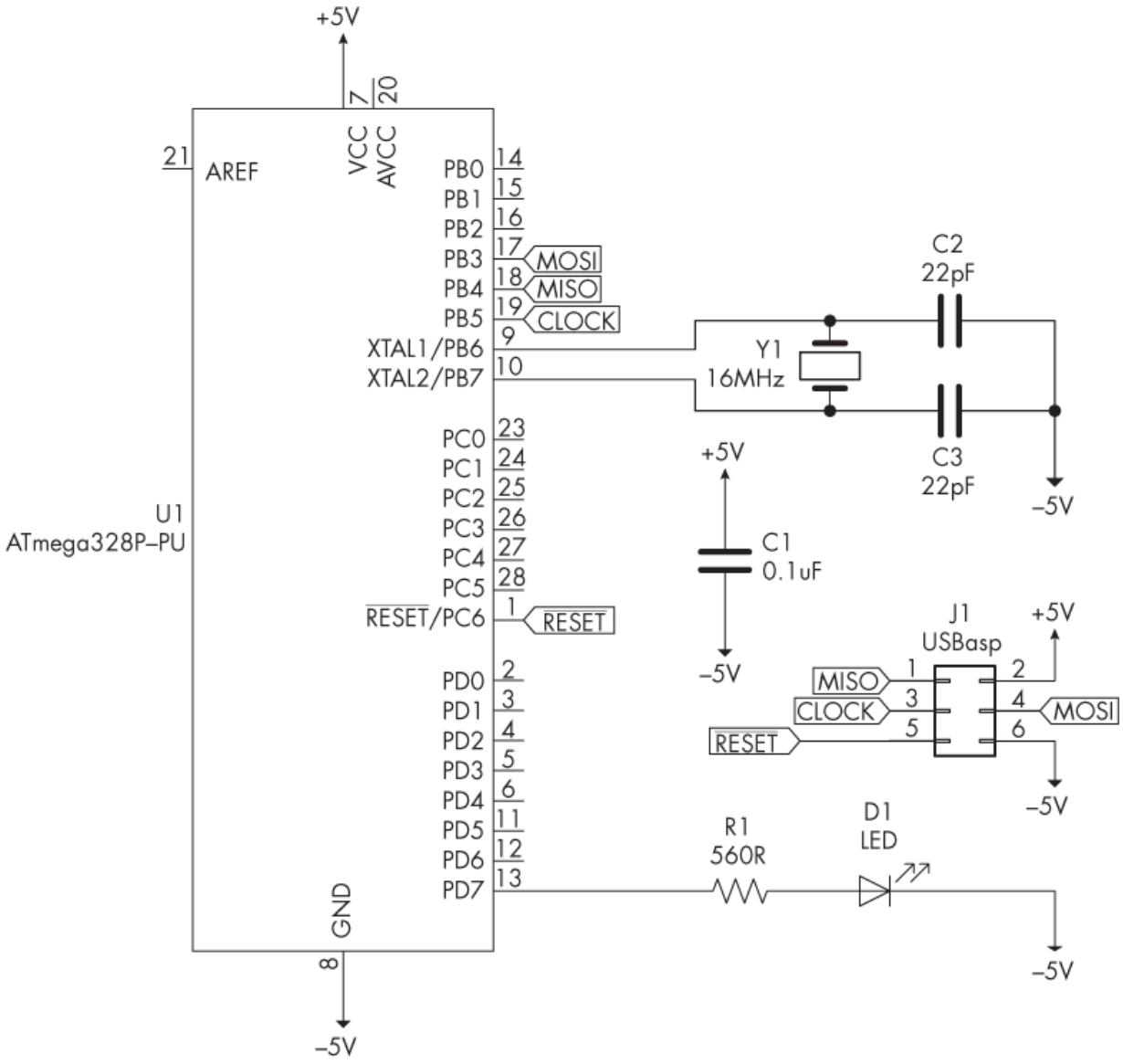


Figure 15-17: The schematic for Project #46

Now upload the following sketch:

```
// Project #46 - Blink LED on D7

void setup()
{
```

```
    pinMode(7, OUTPUT);
}
void loop()
{
    digitalWrite(7, HIGH);
    delay(1000);
    digitalWrite(7, LOW);
    delay(1000);
}
```

The LED connected to the microcontroller's pin 13 should blink as expected, on and off every second. You now have a working example of a minimalist Arduino circuit, which you can use for experimenting; if you want to permanently include it in your projects, you can solder it to the stripboard or on your own customized PCB. The resistor and LED are just for demonstration purposes, so of course you can remove them if not required for future projects.

The goal of this circuit is to reduce power consumption. After removing the LED and resistor, I measured a current draw of around 22 mA—much better than the standard Uno board, which usually draws around 44 mA.

This is a great start, but it's possible to lower power consumption even further using the software-based methods

you learned earlier in the chapter. You'll need a multimeter capable of measuring current down to microamps to measure the consumption levels when the project is running. When I pair this circuit with the Power Down/Wake Periodic mode sketch in Listing 15-1, the current measures around 3 µA (microamps, each equal to 1,000 milliamps) when the Arduino is powered down. Trying out Listing 15-3 with this circuit also returned around 3 µA. Battery-powered projects that combine hardware- and software-based methods of saving power can last for an incredibly long time.

For future projects that use this type of minimalist circuit, you'll need to know which pin on the microcontroller relates to which pin in the Arduino sketches, since the pin numbers on an Arduino board don't match the pins on the microcontroller. All the analog, digital, and other pins available on the normal Arduino board are also available in the breadboard version; you simply need to connect directly to the microcontroller.

The R2 and LED2 are on digital pin 13 in your breadboard Arduino. Table 15-2 lists the Arduino pins on the left and the matching ATmega328P-PU pins on the right.

Table 15-2: 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

Arduino pin name	ATmega328P-PU pin
D8	14
D9	15
D10	16
D11	17
D12	18
D13	19
(5 V only)	20
AREF	21
GND	22
A0	23
A1	24

Arduino pin name	ATmega328P-PU pin
A2	25
A3	26
A4	27
A5	28

To avoid confusion, retailers such as Freetronics sell adhesive labels to place over the microcontroller, like those shown in Figure 15-18 (visit the Freetronics home page and search for *labels*).

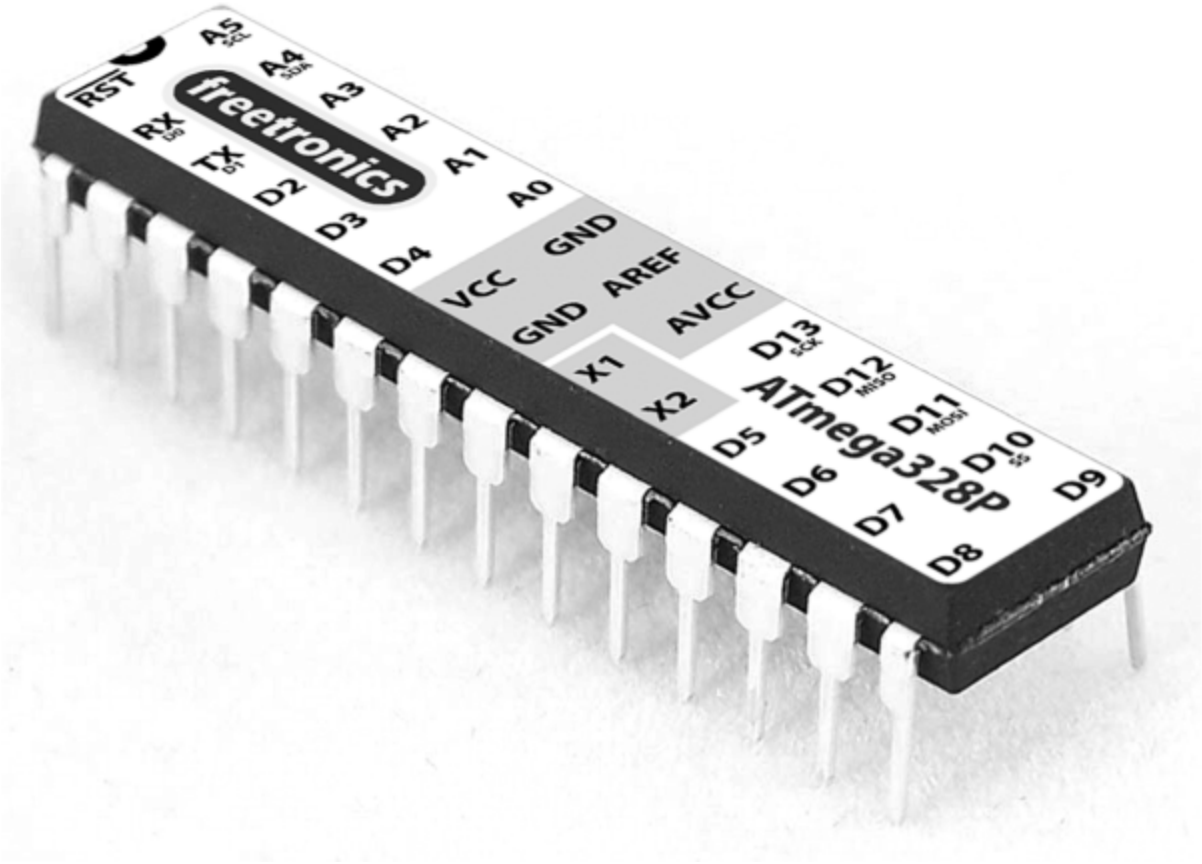


Figure 15-18: Arduino pin labels, recommended for fast prototyping

Using the labels is a great way to ensure you don't mix up the pinouts when you're in a rush.

Moving On

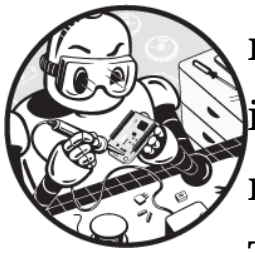
In this chapter, you learned to measure how much power your Arduino uses and how you can minimize power consumption using a variety of hardware- and software-based methods, allowing you to make battery-powered projects that will last much longer than they otherwise would. You've also learned to

use some new sensor and timing modules and the USBasp programmer. You can now create projects that can operate from batteries for an extended period of time, such as monitoring weather data over a season in a remote location or detecting nearby movements in an area without mains AC power.

In the next chapter, I'll show you how to interface your car with the Arduino using the automotive CAN data bus to monitor and record vehicle engine data and more.

16

MONITORING AUTOMOTIVE ELECTRONICS WITH THE CAN BUS



Cars use a complex network of electronics to monitor operations and control various functions, including engine management, temperature measurement, and window and lock controls. These electronics and devices communicate with each other over the *controller area network data bus (CAN bus)*.

In this chapter, you'll learn to interface your Arduino with the CAN bus to monitor data from certain automotive electronics. You'll set up and test an Arduino-to-CAN-bus data connection

and then monitor and record real-time engine statistics for later analysis.

The CAN Bus

You may be familiar with the SPI and I^2C data buses. The CAN bus is a data bus that is used to connect the engine management computer with other devices in a contemporary automobile. This reduces physical wiring between devices and allows you to connect an external device, such as a computer, a scanning tool, or your Arduino, to the automobile for monitoring operations.

For example, a car's engine management computer reads sensors in the engine and braking system. Using the CAN bus, it then broadcasts the speed, engine RPM, temperature, and other data to the computer system that controls the dashboard display. When you press the button on a remote control to lock your car doors, the radio data transceiver receives the signal from the remote control and then likewise instructs the central locking controller unit via the CAN bus to lock all the doors.

The protocol used for communication over the CAN bus is called *On-Board Diagnostic II (OBDII)*, the second generation of the OBD protocol. You can use an Arduino and matching CAN bus shield to communicate with the bus to review car statistics.

Before progressing further, if you plan to build the projects for this chapter, check whether your car has a CAN bus and an OBDII connector, as some older cars will not. To do this, look around under the dashboard of your car. The connector will be in an out-of-the-way position; for example, it appears above the pedals in my Mitsubishi Outlander Sport/ASX, as shown in Figure 16-1. If you can't initially find your OBDII connector, you may have to refer to the manual provided by the manufacturer or the service manual.



Figure 16-1: An OBDII connector socket

When it comes time to connect a cable to the OBDII connector, hold the socket firmly with one hand as you insert the plug with your other hand. The socket is usually not mounted with much support, and if you're not careful, you'll risk breaking the support bracket.

Required CAN Bus Hardware

For the projects in this chapter, you'll need an Arduino shield such as PMD Way part 13479987, shown in Figure 16-2. Whichever shield you choose, it must use the Microchip MCP2515 CAN controller and MCP2551 CAN transceiver ICs.

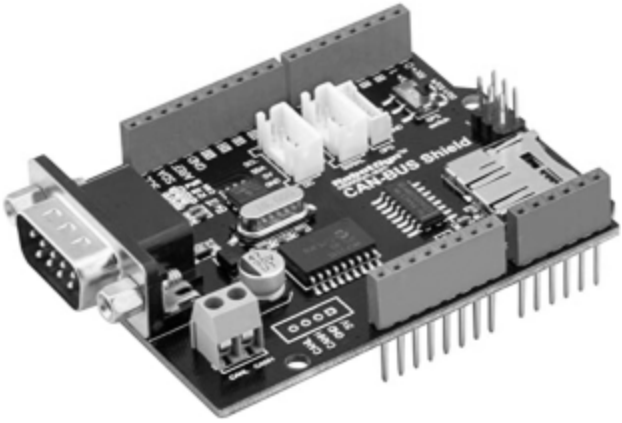


Figure 16-2: A CAN bus shield for Arduino

Next, you'll need a matching cable that runs from the vehicle's OBDII port to the nine-pin plug on the shield, such as PMD Way part 717211, shown in Figure 16-3.



Figure 16-3: An OBDII-to-shield cable

Finally, you'll need a way to power your Arduino. If you don't have a laptop computer with enough battery life to last an hour or so, consider getting a USB power bank, as shown in Figure 16-4.



Figure 16-4: A USB power bank

Alternatively, you could power your project using a USB cable and a car USB adapter, as shown in Figure 16-5.



Figure 16-5: A car USB power adapter

The OBDII connection may differ for each brand and model of car, so it's a good idea to check with enthusiast forums for more information about your car before purchasing the hardware required for this chapter. For example, Mitsubishi owners could consult the forum at *https://www.mitsubishi-forums.com* and search for OBDII posts.

Once you have the required hardware, download the CAN bus library ZIP file from *https://github.com/sparkfun/SparkFun_CAN-Bus_Arduino_Library/archive/master.zip*. Open the Arduino IDE

and select **Sketch** ▸ **Include Library** ▸ **Add ZIP Library** to install the library.

BEING SAFE WHEN WORKING WITH AUTOMOBILES

Never wire directly from your car's 12 V output to the Arduino since spikes in the power supply may damage your hardware setup. Using an external source means you can be sure of a smooth 5 V DC.

Moreover, when building the projects in this chapter, if you're running your car in your garage, keep the door open to allow exhaust gases to escape freely. If you're using a laptop computer while driving around to gather real-world data, don't look at or use the laptop while you're driving.

Also, note that this chapter is concerned with reading data from the CAN bus, not writing to it. There may be legislation in your area that doesn't permit such modifications to a motor vehicle for safety reasons.

You're now ready to test your hardware with a simple CAN bus data retrieval sketch.

Project #47: Monitoring Engine Data

In this project, you'll use the CAN bus shield to receive basic engine data and speed from the car's engine control unit (ECU) and display this in the Arduino IDE's Serial Monitor. This is a good way to check that the hardware is operating and increase familiarity with your car's operating statistics.

Your project will monitor the following:

- Car speed
- Engine RPM
- Throttle position
- Percentage of throttle application
- Engine coolant temperature
- Oxygen sensor voltage
- Mass air flow (MAF) sensor values

The ECU uses the oxygen sensor voltage to determine the amount of oxygen present in the engine exhaust gases, which indicates how fuel efficiently the engine is operating. Once your car's MAF sensor measures the amount of air entering the engine, the ECU uses this value to determine the correct amount of fuel needed for optimal engine operation.

You'll need the following parts for this project:

- An Arduino Uno or compatible board and USB cable

- A CAN bus shield for Arduino
- An OBDII-to-shield cable
- A laptop computer (preferable but not required)

If you have a laptop computer running the Arduino IDE, you can use that to monitor the project's output. Otherwise, you could set up a desktop computer in your garage. If neither of these is possible, skip to the next project.

To assemble the project, connect the CAN bus shield to the Arduino, the Arduino to the computer, and the CAN bus shield to the OBDII connector in the car. If your Arduino board has a full-sized USB socket, such as that shown in Figure 16-6, it can rest against the solder joints of the CAN bus shield. If this is the case, place some paper between the two to insulate the Arduino from the shield.

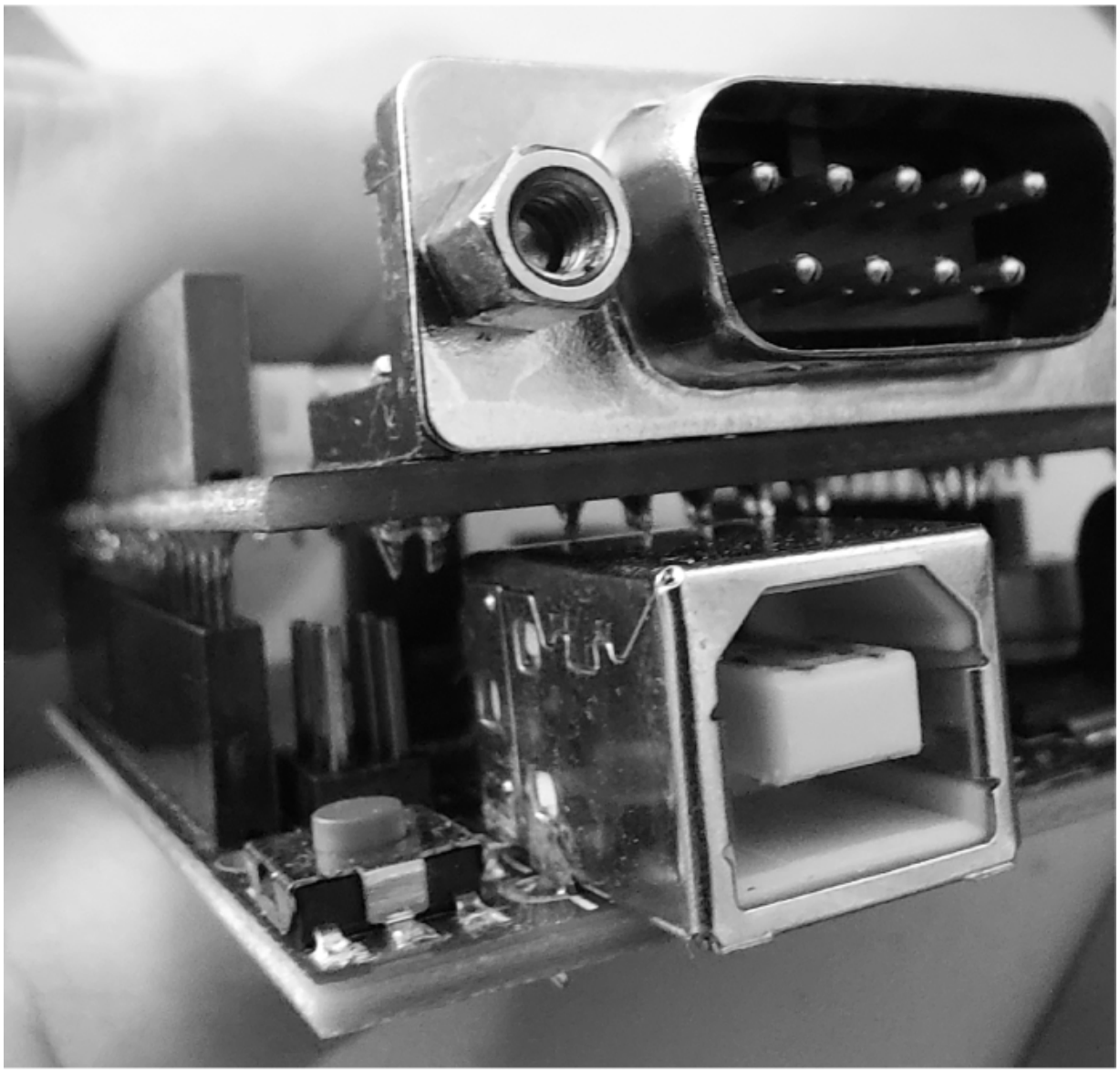


Figure 16-6: An Arduino USB port contacting the CAN bus shield

Enter and upload the sketch, and then make sure the car won't move if you start the engine. Start the engine, open the Serial Monitor in the Arduino IDE, and set the data rate to 115,200 baud. After a few moments, the engine data should be displayed, as shown in Figure 16-7.

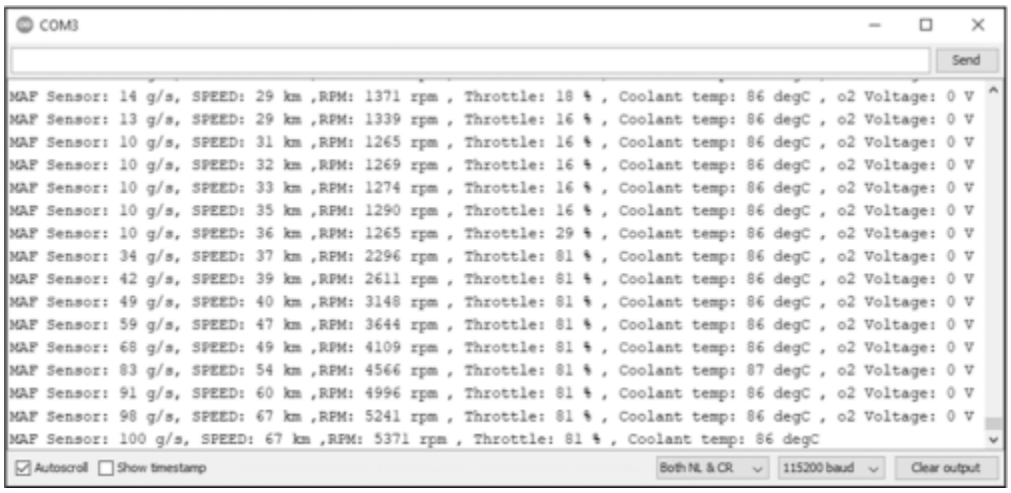


Figure 16-7: Example output from Project #47

In the figure, the speed is greater than 0 km/h. That's because I took the car for a drive with a passenger, who took the screenshot using a laptop. This test also demonstrates the potential differences in OBDII data between different cars: in the results shown in Figure 16-7, you can see that I didn't get any readings for the oxygen sensor voltage from the ECU.

Let's see how this works:

```
// Project #47 - Monitoring engine data

❶ #include <Canbus.h>
int requestDelay = 100;
char buffer[456];

void setup()
```

```
{
 ❷ Serial.begin(115200);
   // Initialize CAN bus shield with the requir
 ❸ if (Canbus.init(CANSPEED_500))
   {
       Serial.println("CAN initialization ok");
   } else
   {
       Serial.println("CAN initialization error
       while (1) {}
   }
   delay(2000);
}

void loop()
{
 ❹ Canbus.ecu_req(VEHICLE_SPEED, buffer);
   Serial.print("Speed: ");
   Serial.print(buffer);
   delay(requestDelay);

   Canbus.ecu_req(ENGINE_RPM, buffer);
   Serial.print(", RPM: ");
   Serial.print(buffer);
   delay(requestDelay);

   Canbus.ecu_req(THROTTLE, buffer);
   Serial.print(", Throttle: ");
```

```
      Serial.print(buffer);
      delay(requestDelay);

      Canbus.ecu_req(ENGINE_COOLANT_TEMP, buffer);
      Serial.print(", Coolant temp: ");
      Serial.print(buffer);
      delay(requestDelay);

      Canbus.ecu_req(O2_VOLTAGE, buffer);
      Serial.print(", o2 Voltage: ");
      Serial.print(buffer);
      delay(requestDelay);

      Canbus.ecu_req(MAF_SENSOR, buffer);
      Serial.print(" , MAF sensor: ");
      Serial.println(buffer);
      delay(requestDelay);
  }
```

The sketch first includes the CAN bus library ❶ and then creates two variables. The first is `requestDelay`, which is used to create a delay after reading data from the CAN bus. You may need to alter this variable's value depending on the car you're using, but try 100 first. If the data returned doesn't match the type or you're not seeing any data at all, you will need to experiment with values that are higher or lower. The second variable is the

large character array `buffer`, used to store data received over the CAN bus and then displayed in the Serial Monitor.

In `void setup()`, the sketch initializes the serial output ❷. It then starts CAN bus communication ❸, also checking whether it's possible to communicate between the Arduino and the CAN bus using the `if (Canbus.init (CANSPEED_500))` function. The parameter `CANSPEED_500` sets the CAN bus shield-to-data rate. The required speed varies depending on the car; the appropriate speed for my vehicle was 500 kbps.

The sketch retrieves data using the `Canbus.ecu_req()` functions in `void loop()`. These functions have two parameters: the data type and the variable in which to store the data. For example, the sketch retrieves and places the first piece of data, speed, into the `buffer` variable and then displays that data on the Serial Monitor ❹. A short delay follows, after which the next parameters are retrieved and displayed and the process repeats.

If the project doesn't work, you may need to change the data rate. You can change the parameter to `CANSPEED_125` or `CANSPEED_250` for 125 kbps or 250 kpbs, respectively, trying different speeds to see what works.

You can learn a lot about your car using this sketch. For example, I found that the ECU in my Mitsubishi limited the throttle to 18 percent when the transmission was in park or neutral, preventing drivers from unnecessarily over-revving the engine.

Now that you have a basic framework to retrieve OBDII data, you're ready to learn to save this data to a microSD card for later review. Keep the hardware for this project together, as you'll also use it in the next project.

Project #48: Logging OBDII Car Data

In this project, you'll log the data retrieved from the CAN bus to a microSD card, which allows you to analyze the data over time, whether the car is stationary or being driven. The required hardware and assembly are the same as the previous project, except that you'll also need a microSD card—the type used in Chapter 9 will be fine—and a power supply for your Arduino, as described at the start of this chapter.

Enter and upload the sketch, insert the microSD card into the CAN bus shield, and then confirm the car won't move if you start the engine. Connect the Arduino to the power supply and start the engine. The sketch should retrieve and log the data to a

text file called *CANBUS.TXT* on the microSD card, with a new entry around every second.

If you're able, take your car for a drive. After you've finished, remove the power from the Arduino, insert the microSD card into your PC, and open it for viewing. It should include the *CANBUS.TXT* file containing all your logged data, as shown in Figure 16-8.

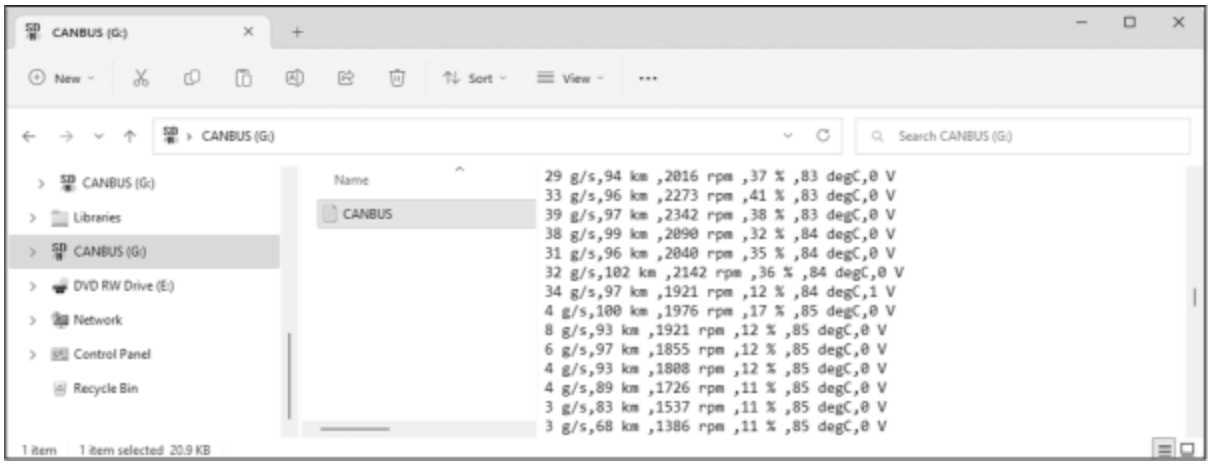


Figure 16-8: Example data file contents from a microSD card

Open *CANBUS.TXT* in a spreadsheet, and during the opening process, delimit with a comma and a space. For example, if you're using Microsoft Excel, you can do this using the Text Import Wizard that appears when you open a text file, as shown in Figure 16-9.

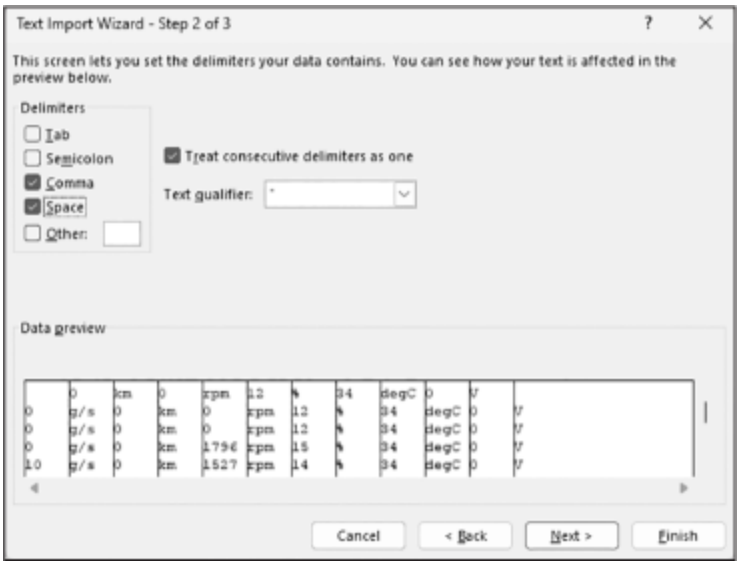


Figure 16-9: Opening the CANBUS.TXT *file in Excel*

Now that your data is in normal spreadsheet file format, you can now open it in spreadsheet software for further analysis, such as to find relationships between speed and RPM or RPM and throttle position, as shown in Figure 16-10.

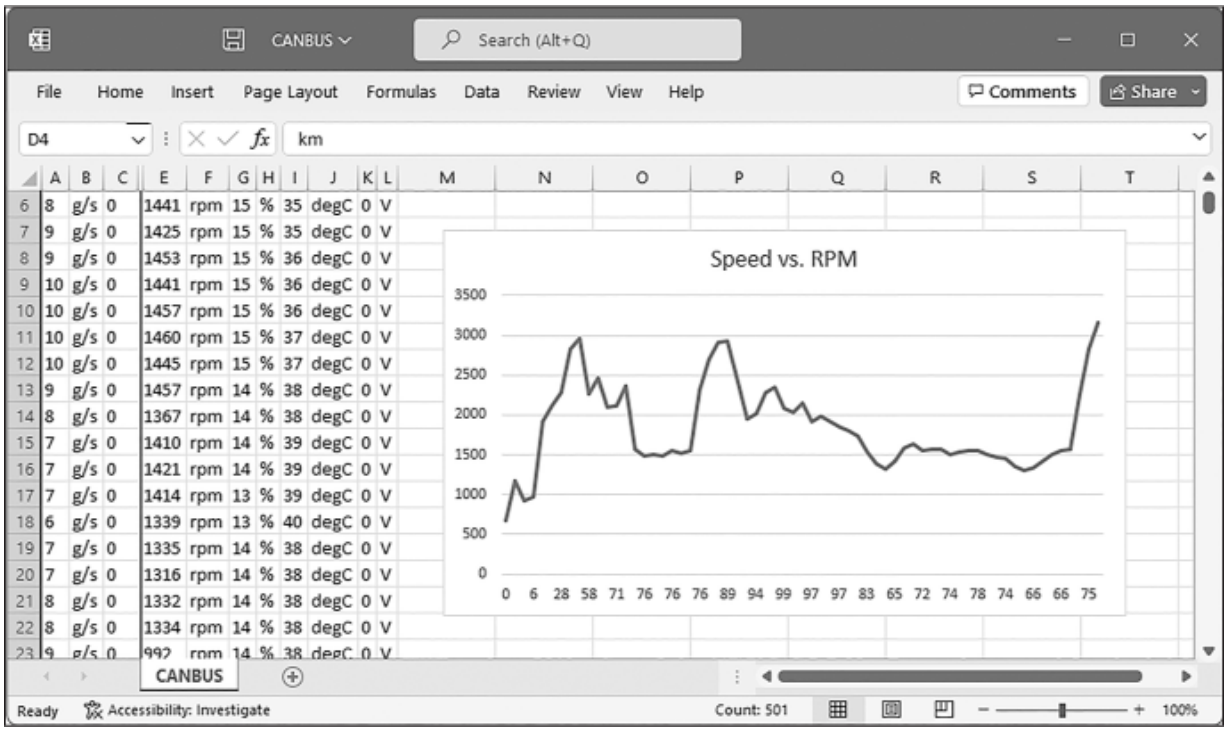


Figure 16-10: Working with example data in the spreadsheet

Let's see how the sketch works:

```
// Project #48 - Logging OBDII car data

❶ #include <Canbus.h>
int requestDelay = 1000;
char buffer[456];

#include <SPI.h>
#include <SD.h>
int chipselect = 9;

void setup()
```

```
{
    Serial.begin(115200);
    // Check uSD card
 ❷ if (!SD.begin(chipselect))
    {
        Serial.println("Card failed, or not pres
        // Stop
        while (1);
    }
    Serial.println("Micro SD card initialized.")
 ❸ if (Canbus.init(CANSPEED_500))
    {
        Serial.println("CAN initialization ok");
    } else
    {
        Serial.println("CAN initialization error
        while(1) {}
    }
    delay(2000);
}

void loop()
{
    // Open file on memory card
 ❹ File dataFile = SD.open("CANBUS.TXT", FILE_WR
 ❺ if (dataFile)
    {
     ❻ Canbus.ecu_req(VEHICLE_SPEED, buffer);
```

```
❼ dataFile.print(buffer);
  delay(requestDelay);

  Canbus.ecu_req(ENGINE_RPM, buffer);
  dataFile.print(",");
  dataFile.print(buffer);
  delay(requestDelay);

  Canbus.ecu_req(THROTTLE, buffer);
  dataFile.print(",");
  dataFile.print(buffer);
  delay(requestDelay);

  Canbus.ecu_req(ENGINE_COOLANT_TEMP, buff
  dataFile.print(",");
  dataFile.print(buffer);
  delay(requestDelay);

  Canbus.ecu_req(O2_VOLTAGE, buffer);
  dataFile.print(",");
  dataFile.print(buffer);
  delay(requestDelay);

  Canbus.ecu_req(MAF_SENSOR, buffer);
  dataFile.print(",");
  dataFile.println(buffer);

  // Close file on memory card
```

```
        ❽ dataFile.close();
          delay(1000); // Around 1-second delay be
      }
  }
```

The sketch includes the requisite CAN bus library and variables ❶, followed by the library to use the microSD card reader and the `chipselect` variable. This variable indicates which CS pin the microSD card socket uses, in this case using a value of 9 for Arduino digital pin 9. In `void setup()`, the sketch checks that the microSD card subsection is working ❷ and then initializes the CAN bus interface with the required bus speed ❸. As in the previous project, if the sketch doesn't log any data, try changing the value for `requestDelay`.

If the microSD card and the CAN bus initialize successfully, the sketch moves on to the main loop and opens the file on the microSD card for writing ❹. If a file to be written to exists ❺, the sketch moves on to retrieve the first piece of data from the ECU ❻; then, instead of sending the data to the Serial Monitor, the sketch writes it to the microSD card ❼. The process then repeats for the other five pieces of data from the ECU. Once finished, the sketch closes the file for writing ❽. You can change the period between logging data using `delay()`.

For a challenge, you could add a real-time clock IC such as the popular DS3231 series to log time and date information along with the engine parameters. You could also add a GPS module to log speed from both the ECU and the GPS, allowing you to compare the inaccuracy of your car's speedometer.

Moving On

This chapter gave you the tools to monitor and record various parameters of information made available by your vehicle's systems such as speed, engine RPM, throttle position, coolant temperatures, and so on that give greater insight into the operation of the engine and its sensors. Each type of vehicle is different, so by using the projects you built in this chapter, you can find data specific to your own ride.

In the next chapter, you'll learn how to make an Arduino communicate with other devices using the RS232 data bus.

17

ARDUINO-TO-RS232 COMMUNICATION



The RS232 data bus is a popular wired method for electronic devices to communicate with one another. It was originally devised in the late 1960s and is still used today in industrial programmable logic controllers (PLCs) and applications that require reliable wired point-to-point data transfer. It's also often used to interface computers with older devices.

This chapter shows you how to interface an Arduino with the RS232 bus for data transmission and remote-control applications. You'll learn to:

- Add an RS232 interface to your Arduino using a board or shield
- Build a PC-to-Arduino remote control that operates via RS232
- Set up communication over RS232 between two Arduino boards

You can use the projects in this chapter as the framework for your own future Arduino-to-RS232 communication needs.

The RS232 Bus

The RS232 bus was originally designed to allow connections between data terminals and modems that would then be connected to remote mainframes and minicomputers. The standard evolved over time, with each version being noted by a letter suffix, such as RS232-C. For the purposes of this chapter, I'll refer to all versions of the standard as RS232.

Though the RS232 bus is an older technology, there are still many reasons to use it, including interfacing with older hardware and creating long, reliable wired data connections between devices such as your Arduino and a PC. RS232 data lines can stretch up to 15 meters in length before reliability becomes an issue, though you can extend the possible length over 15 meters by reducing the data speed. The examples in this

chapter use 9,600 bps, but in your own projects, you can use lower speeds supported by the Arduino if you want.

RS232 is a serial data bus that operates in a similar manner to the Arduino's serial port (usually found on pins D0 and D1). The basic RS232 connection has a TX (transmit) and RX (receive) wire, along with a common ground. The full RS232 standard has additional lines used for telephony, controlling data start/stop, and sending status between two devices, but you won't use these lines in this chapter.

Although the RS232 bus sends sequential bits of data representing ones and zeros, the signal type is different from the TTL serial used with Arduino. For example, Figure 17-1 shows data sent from an Arduino's serial port. You can see that the ones (known as *marks* in RS232 parlance) are at 5 V, while the zeros (*spaces*) are at 0 V.

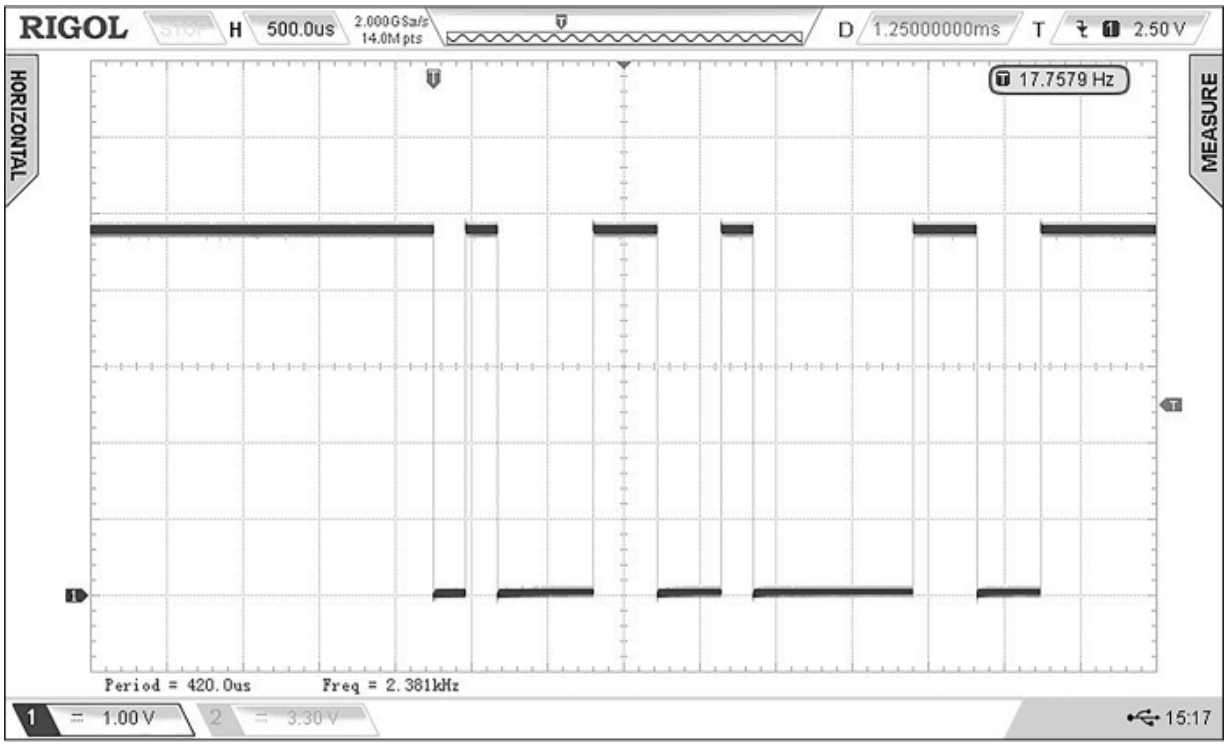


Figure 17-1: Time to live (TTL) logic data from the Arduino serial port

Now consider the same piece of data being sent along RS232, as shown in Figure 17-2. RS232 requires marks to be a negative voltage and spaces to be a positive voltage.

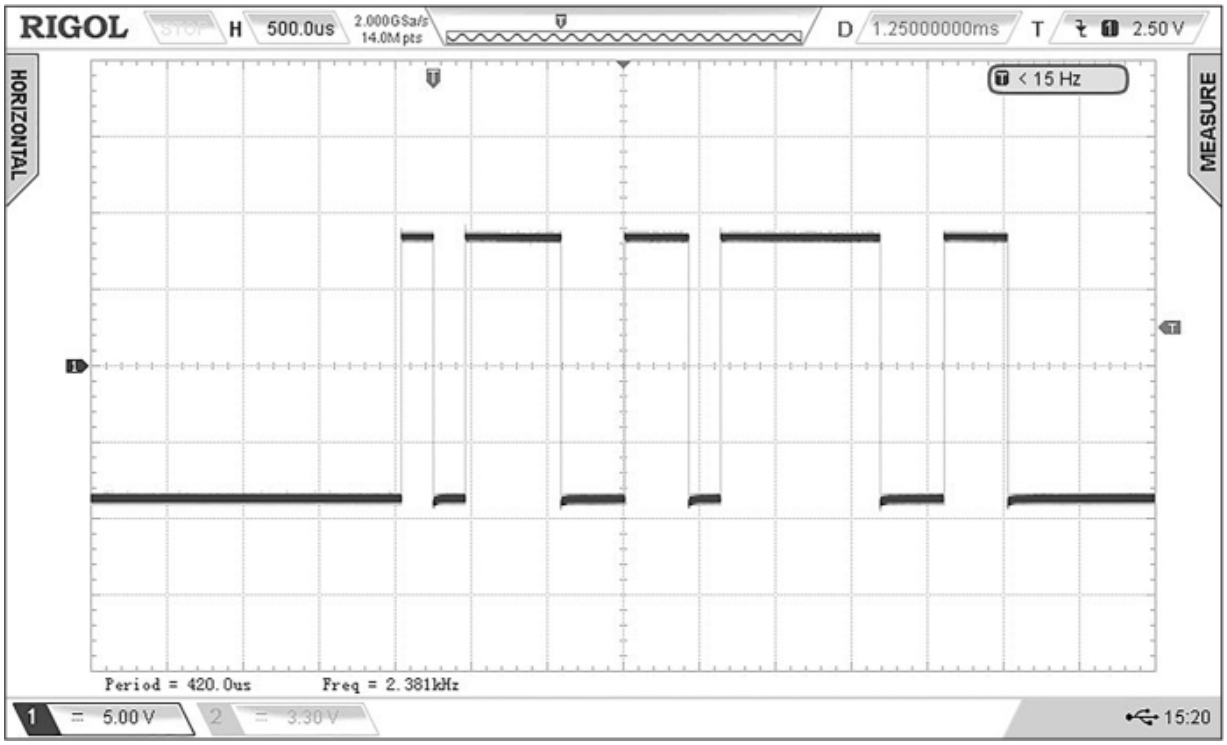


Figure 17-2: RS232 logic data from an RS232 port

Using opposite voltages for marks and spaces clearly defines the 2 bits, avoiding potential confusion and thus increasing data accuracy. For each byte of data sent, RS232 includes a start bit and a stop bit on the ends of each 8 bits of data. (Later in this chapter, I'll show you how to change the number of bits sent for each transmission.)

The voltages for a mark can range from 3 V to 15 V DC and for a space between −3 V and −15 V DC. Though long cables mean greater variation in signal voltage—the longer the cable, the larger the voltage drop, because of the wire's resistance—the wide voltage range between marks and spaces means the

length of the RS232 cable affects signal integrity less. If the surrounding electrical environment is noisy, there is less chance of confusion between a mark and a space.

The increased voltage required for the RS232 signals is generated by a TTL-to-RS232 serial converter IC, the MAX3232, found on RS232 shields and modules for Arduino. This IC uses a charge-pump circuit to convert the 5 V DC to +/—10 V DC or more, but with a reduced current. Therefore, you don't need a separate dual-rail power supply to use RS232 and the Arduino.

Connecting to RS232

There are several easy ways to interface your Arduino with an RS232 connection on a device or PC. Whichever interface device you use will have a nine-pin connector for RS232 and a way to electrically connect the onboard MAX3232 IC to the Arduino's TX/RX serial pins.

The most convenient interfacing method is to use an RS232 shield for Arduino (PMD Way part 954042), as shown in Figure 17-3. The shield has a convenient switch that selects between RS232 use and Arduino-to-PC communication; you turn the switch off when uploading sketches and on when using RS232.

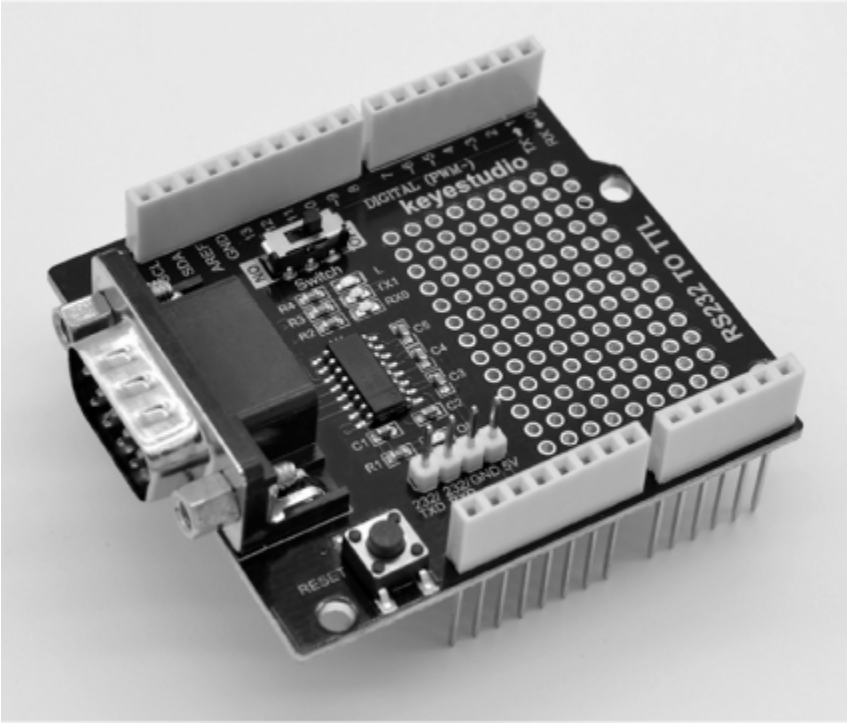


Figure 17-3: An RS232 shield for Arduino

A more compact alternative to the RS232 shield for Arduino, especially useful when making smaller Arduino-based circuits, is the RS232-to-TTL module shown in Figure 17-4.

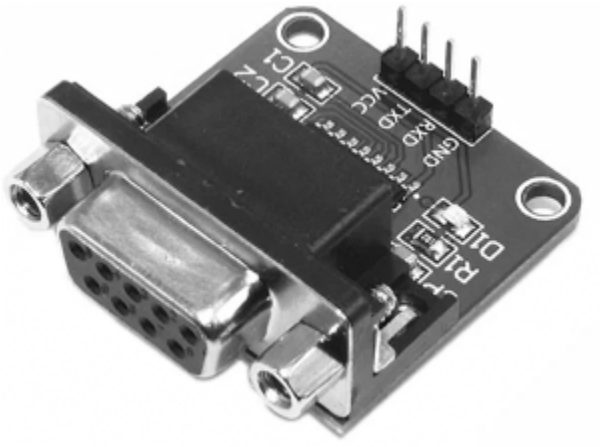


Figure 17-4: An RS232-to-TTL module

This module connects to the Arduino using the inline header pins: connect V_{CC} and GND to Arduino 5V and GND, module TXD to Arduino RX (D0), and module RXD to Arduino TX (D1). The projects in this chapter use the RS232 shield for Arduino, but you could use the module instead for the same projects.

For this chapter, you'll also need an RS232 cable to bridge the Arduino and RS232 hardware with a device or computer. Based on the standard connector on RS232 shields and modules and the standard PC RS232 socket (shown at the top of Figure 17-5), you'll need a double-ended nine-pin female cable.

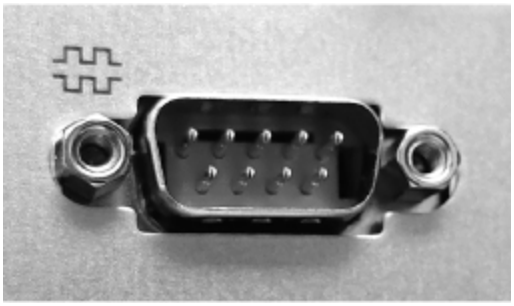


Figure 17-5: The back of a desktop PC showing an RS232 socket

If you're using a Mac or laptop computer or if your PC doesn't have an RS232 port, you'll need a USB-to-RS232 cable such as PMD Way part 514539A, as shown in Figure 17-6.



Figure 17-6: A USB-to-RS232 cable

The RS232 connectors on your PC or Arduino shield are the nine-pin variety with two rows of pins, in either a male or female configuration. When working with old devices with 25-pin connectors, you should be able to buy a converter plug or cable like the one shown in Figure 17-7 from the supplier of your other RS232 gear.

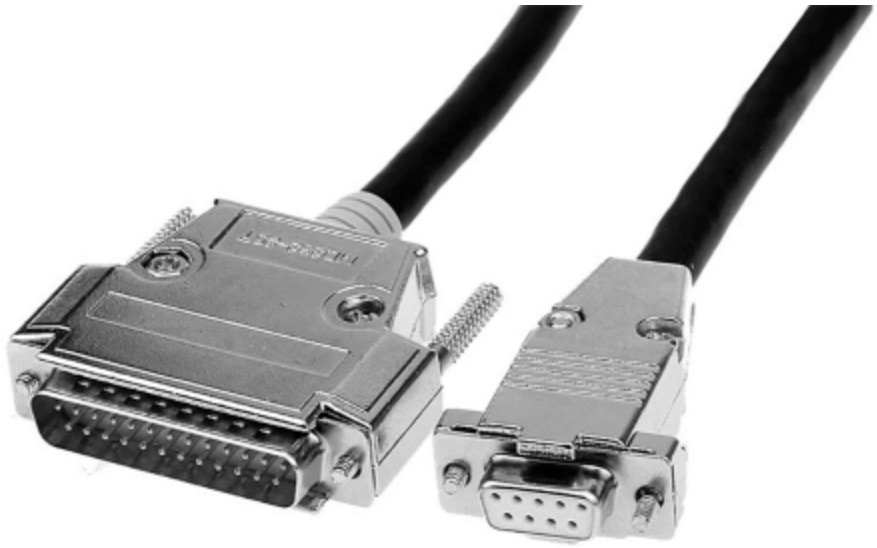


Figure 17-7: A DB9-to-DB25 serial cable

Once you've collected the required hardware, you're ready to set up and test a PC-to-Arduino connection via RS232.

Testing Arduino-to-PC Connections via USB

To test your Arduino-to-PC connection, upload Listing 17-1 to your Arduino board.

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

void loop()
```

```
{
    Serial.print("Hello ");
    delay(1000);
}
```

Listing 17-1: Testing the Arduino-to-PC connection

Next, remove the USB cable from the Arduino and then connect the RS232 shield or module as described in the previous section. Reconnect the Arduino to USB or external power and then open your terminal software. As in previous chapters, the projects in this chapter use the free terminal emulator CoolTerm by Roger Meier, which you can download at *http://freeware.the-meiers.org*.

Once you have opened the terminal software, click **Options**. You should be presented with a menu of serial port options, as shown in Figure 17-8.

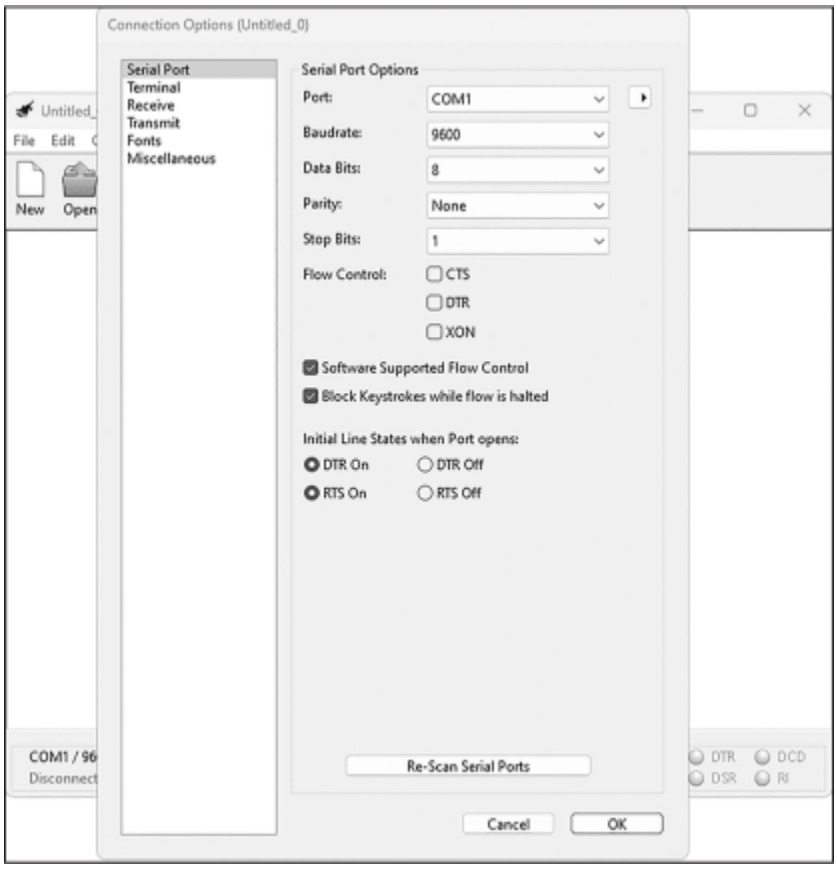


Figure 17-8: The CoolTerm options menu

Choose your serial port (for example, COM1) from the Port drop-down menu and ensure the rest of the serial port settings match those shown in the figure. Click **OK** to close the menu and then click **Connect** to begin data transfer from the Arduino to the PC via RS232. The PC should repeatedly receive the word "Hello" and display it on the terminal window, as shown in Figure 17-9.

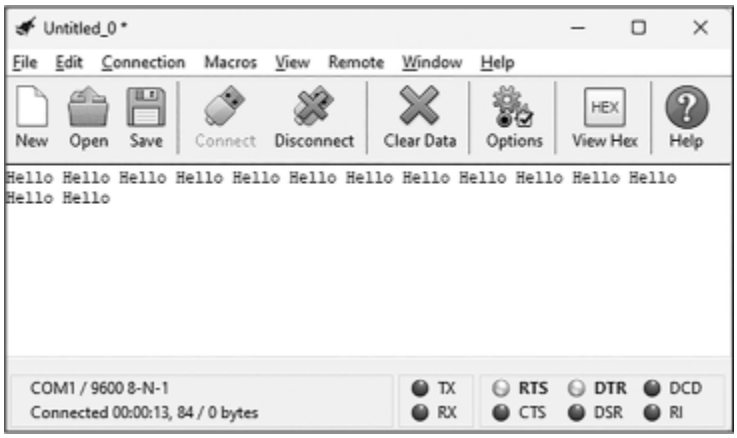


Figure 17-9: The results of Listing 17-1 in the CoolTerm window

You can use this simple setup to capture data from your Arduino over a longer distance than is possible using USB cables. In CoolTerm, press CTRL-R to start recording any output received to a text file, and press CTRL-SHIFT-R to stop recording.

Keep your hardware connected; you'll use it in the following project to make a PC-to-Arduino remote control over RS232.

Project #49: Creating a PC-to-Arduino Remote Control

This project demonstrates the basic framework to remotely control your Arduino over RS232 to control devices connected to digital outputs, request information from sensors, or even add computer control to your Arduino projects with PC-based software that writes to the serial port.

You'll need the following parts:

- An Arduino Uno or compatible board and USB cable
- An RS232 shield or module for Arduino
- A nine-pin cable for PC-to-RS232 connection

To assemble the project, connect the Arduino to the RS232 hardware and PC as described in the previous section. Uploaded the sketch for Project #49, open the CoolTerm software, and click **Connect** to begin data communication between the Arduino and the PC via RS232.

Press **0** or **1** on the PC keyboard to request the values of the Arduino's analog inputs 0 and 1, respectively, and turn the onboard LED on and off by pressing **2** and **3**. If any other key is pressed, the terminal should display the error message `Unrecognized command`, as shown in Figure 17-10.



Figure 17-10: Example output from Project #49

Let's see how this works:

```
  // Project #49 - PC-to-Arduino remote control

❶ void doSomething0()
  {
      Serial.print("A0 value is: ");
      Serial.println(analogRead(0));
  }

  void doSomething1()
  {
      Serial.print("A1 value is: ");
      Serial.println(analogRead(1));
  }
```

```
void doSomething2()
{
    digitalWrite(13, HIGH);
    Serial.println("LED On");
}

void doSomething3()
{
    digitalWrite(13, LOW);
    Serial.println("LED Off");
}

void setup()
{
 ❷ Serial.begin(9600);
    pinMode(13, HIGH);
    digitalWrite(13,LOW);
}

void loop()
{
    char command;
 ❸ if(Serial.available()) // If data received by
    {
        command = Serial.read(); // Save the data
     ❹ switch (command)
        // Act on the received data:
        {
```

```
            case '0': doSomething0(); break;
            case '1': doSomething1(); break;
            case '2': doSomething2(); break;
            case '3': doSomething3(); break;
          ❺ default: Serial.println("Unrecognized
        }
    }
}
```

Starting at ❶, the sketch creates four custom functions to be called upon receiving a matching command. The sketch then initializes the Arduino serial port ❷ and sets the onboard LED pin to output and turns it off.

In the main loop, the Arduino waits for a character of text to come in from the RS232 connection via serial ❸ and then acts on this character using a `switch…case` function ❹. If the character falls outside of the range of choices, the sketch sends an error message back to the PC ❺. The actions in this sketch are simply demonstrations, and you can replace the contents with your own requirements.

This project demonstrates that you can control or monitor any Arduino operation with a PC over an RS232 connection. You could also write your own custom PC software for remote

control. Any code that can write to a COM: or serial port on the machine can control the Arduino, and any code that can receive text from the serial port can act upon it.

In the next project, I'll show you how to set up communication between two Arduinos via RS232. Since RS232 is an extension of the Arduino's serial communication, two Arduinos can communicate with each other using RS232. However, the RS232 cable required for this purpose is slightly different than a normal cable: the TX and RX data lines need to be crossed over. That is, the TX pin on one end of the cable needs to be connected to the RX pin at the other end, and vice versa. Otherwise, you'd be trying to connect the TX pins of the two Arduinos, which won't work.

The solution to this problem is to use a *crossover*, or *null modem*, cable, which has the internal wiring required for direct RS232-to-RS232-device communication, such as PMD Way part 6045480L15 shown in Figure 17-11.



Figure 17-11: An RS232 crossover cable

If you're experimenting over just a short distance, you can instead just use jumper wires to connect between the two RS232 shields or modules for Arduino: just connect the TX and RX on one side to the RX and TX on the other, and connect GND across the two.

Project #50: Enabling Arduino-to-Arduino Communication

This project demonstrates how one Arduino talks to another over RS232. One Arduino board with a BMP180 temperature sensor will send the current temperature to another Arduino

board via RS232, and the I²C LCD connected to the receiving Arduino will display the value.

You'll need the following parts for this project:

- Two Arduino Uno or compatible boards and USB cables
- Two RS232 shields or modules for Arduino
- A nine-pin RS232 crossover cable
- A solderless breadboard
- Various jumper wires
- A BMP180 temperature and air pressure sensor board
- A PCF8574 LCD module

One Arduino will act as the transmitter board, the other as the receiver board. To assemble the hardware, attach an RS232 shield or module to each Arduino, then connect both via the crossover cable or jumper wires. Add the BMP180 sensor to the transmitter board, as shown in the schematic in Figure 17-12.

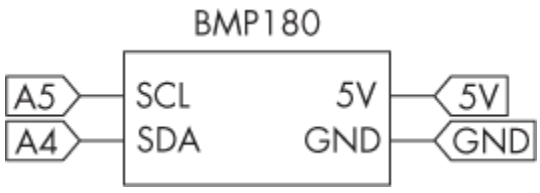


Figure 17-12: The connection diagram for a BMP180 sensor to a transmitter Arduino

Next, add the LCD module to the receiver board, as shown in Figure 17-13.

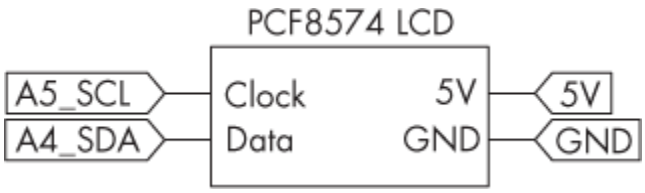


Figure 17-13: The connection diagram for an LCD to a receiver Arduino

If you haven't used the BMP180 sensor before, install the library as described in Chapter 10, and install the I^2C LCD as described in Chapter 13.

Now that the Arduinos are connected, if you're using the RS232 shields, be sure they're switched on. To power the project, you can use either USB cables or external power if there's a distance between the PC and Arduinos.

Next, enter and upload the transmitter sketch to the transmitter Arduino and then the receiver sketch to the receiver Arduino. A moment or two after this, the LCD should display the current temperature on the LCD, an example of which is shown in Figure 17-14.



Figure 17-14: Example output from Project #50

Let's see how this works, starting with the transmitter sketch:

```
// Project #50 - Arduino-to-Arduino communicatio

#include "Adafruit_BMP085.h"
Adafruit_BMP085 bmp;

❶ int temperature;

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

void loop()
```

```
{
  ❸ temperature = bmp.readTemperature();
    Serial.print("Temp (C)= ");
    Serial.print(temperature);
    Serial.print("\n");
    delay(1000);
}
```

The transmitter sketch includes and initializes the BMP180 library and then creates a variable to store the temperature ❶. It starts the serial and the BMP180 ❷. The loop takes a temperature reading from the BMP180 ❸ and then sends the text to the receiving Arduino, starting with the description `Temp (C) =` and followed by the value for the temperature. This is followed by `\n`, which tells the receiving Arduino to add a new line. There's a small delay before the process repeats.

Now let's examine the receiver sketch:

```
// Project #50 - Arduino-to-Arduino communicatio

#include <LiquidCrystal_PCF8574.h>
#include <Wire.h>
LiquidCrystal_PCF8574 lcd(0x27);
```

```
int length = 16;
char newLine = '\n';
char buffer[16];

void setup()
{
    Serial.begin(9600);
    Wire.begin();
    Wire.beginTransmission(0x27);
    lcd.begin(16, 2);
    lcd.setBacklight(255);
    lcd.clear();
}

void loop()
{
    if (Serial.available())
    {
        lcd.home();
        lcd.setCursor(0, 0);
        Serial.readBytesUntil(newLine, buffer, l
        lcd.print(buffer);
    }
}
```

The receiver sketch is tasked with displaying the line of serial data received over RS232 on the LCD. It includes and initializes

the LCD library and I²C bus and then creates three required variables: `length`, the number of characters that fit on one line of the LCD; `endOfLine`, which holds a newline character; and a character array `buffer`, which holds incoming data from the RS232 bus.

In `void setup()`, the sketch sets up serial communications, activates the I²C bus, and initializes the LCD for use by turning on the backlight and setting the cursor to the top left of the display. In the main loop, if incoming characters are detected, the LCD cursor is reset to the top left with `lcd.home();` and `lcd.setCursor(0, 0);`. The `Serial.readBytesUntil()` function captures up to 16 inbound characters from serial, until the newline character is received. The sketch stores these characters in the array buffer. Finally, it displays the characters of the buffer array on the LCD. The process repeats whenever new data is received from the transmitter Arduino.

For more permanent projects that involve two Arduino boards at a distance from one another, the simple demonstration in this project provides an ideal framework for reliable data transmission. For a challenge, you might create a project in which two Arduinos “talk” to each other by sending random data or sensor values.

Other RS232 Data Configurations

Up until now in this chapter, you've used a default 9,600 bps for data speed and 8 bits in each piece of data sent, with no parity and 1 stop bit. (This is often written as "8, None, 1" after the speed in device specification sheets or user manuals.) However, in future projects, you may want to work with devices that use a different data speed or method of sending and receiving data. For example, an old terminal might use 7 bits instead of 8, and 2 stop bits instead of 1.

To solve this problem, you can modify the `Serial.begin()` function's parameters used in the projects in this chapter. Set the first parameter to the data speed you require and the second parameter as the new data configuration:

```
Serial.begin(speed, SERIAL_XYZ)
```

In the second parameter, *X* stands for the requisite number of data bits, which can fall between 5 and 8; *Y* stands for parity, which can be `N` for none, `E` for even, or `O` for odd; and *Z* stands for the number of stop bits, which can be 1 or 2.

For example, to start the serial (and thus RS232) at 4,800 bps with 7-bit data, no parity, and 1 stop bit, use the following

function in your sketch:

```
Serial.begin(4800, SERIAL_7N1)
```

This may come in handy if you decide to interface any vintage equipment with your Arduino. For some retro computing fun, keep an eye out on eBay or computer marketplaces for old computer terminals or printers. For example, check out the Arduino-controlled RS232 to parallel printer interface at *https://hackaday.io/project/18140-serial-to-parallel-printer-interface*.

Moving On

You learned how to communicate between a PC and Arduino via RS232 and between Arduinos via RS232. You now have the tools to harness the RS232 bus for remote control, data communication between Arduinos, and data capture to a PC for analysis and to communicate with RS232-based devices with an Arduino.

In the next chapter, you'll learn how to use an Arduino to communicate with other devices using another data bus, the RS485 data bus.

18

ARDUINO-TO-RS485 COMMUNICATION



In the previous chapter, you used the RS232 bus to create long, reliable wired data connections between Arduino microcontrollers and other devices. This chapter introduces the RS485 bus, which you can use for even longer wired connections.

The RS485 data bus is used for a wide range of devices, from device controllers for heating, ventilation, and air conditioning (HVAC) systems to remote-sensor communications and security system networks. It requires a twin-core shielded wire. Depending on the data speed, your cable length can extend more than 800 meters, and you can connect up to 256 devices

on a single bus. This secure, interference-free communication between devices isn't possible with wireless data modules. RS485 modules are quite inexpensive, allowing you to easily use them in your own projects.

In this chapter, I'll explain how to connect RS485 interface modules with Arduino. You'll learn to:

- Send data from a remote Arduino to your PC via a circuit
- Create a remote thermometer with LCD readout
- Build remote-controlled Arduino projects with one primary controller and two or more secondary remote-controlled Arduinos

You can use these projects as a framework for your future Arduino-to-RS485 communication needs.

The RS485 Bus

The RS485 data bus was designed to connect two or more data terminal equipment devices directly without needing modems over a longer distance than is possible with other data buses like the RS232. It's ideal for connecting two microcontroller-based devices that need to communicate with each other, as it's

more reliable than wireless data links and works over a much greater distance between units.

Like RS232, RS485 is a serial data bus that operates in a similar manner to the Arduino's serial port (usually found on pins D0 and D1). However, data doesn't simply pass from the Arduino through an RS485 interface to the RS485-based device at the other end. The Arduino must tell the RS485 device whether it's transmitting or receiving data before the data can move along the bus as required. The data then travels along the RS485 bus as required to another RS485 device, then to the attached Arduino.

Although the RS485 bus sends sequential bits of data representing ones and zeros, the signal type differs from the TTL serial you're used to with Arduino. For comparison, the image in Figure 18-1 shows some data being sent from an Arduino's serial port, as captured on an oscilloscope. As you can see, the ones are at 5 V and the zeros are at 0 V.

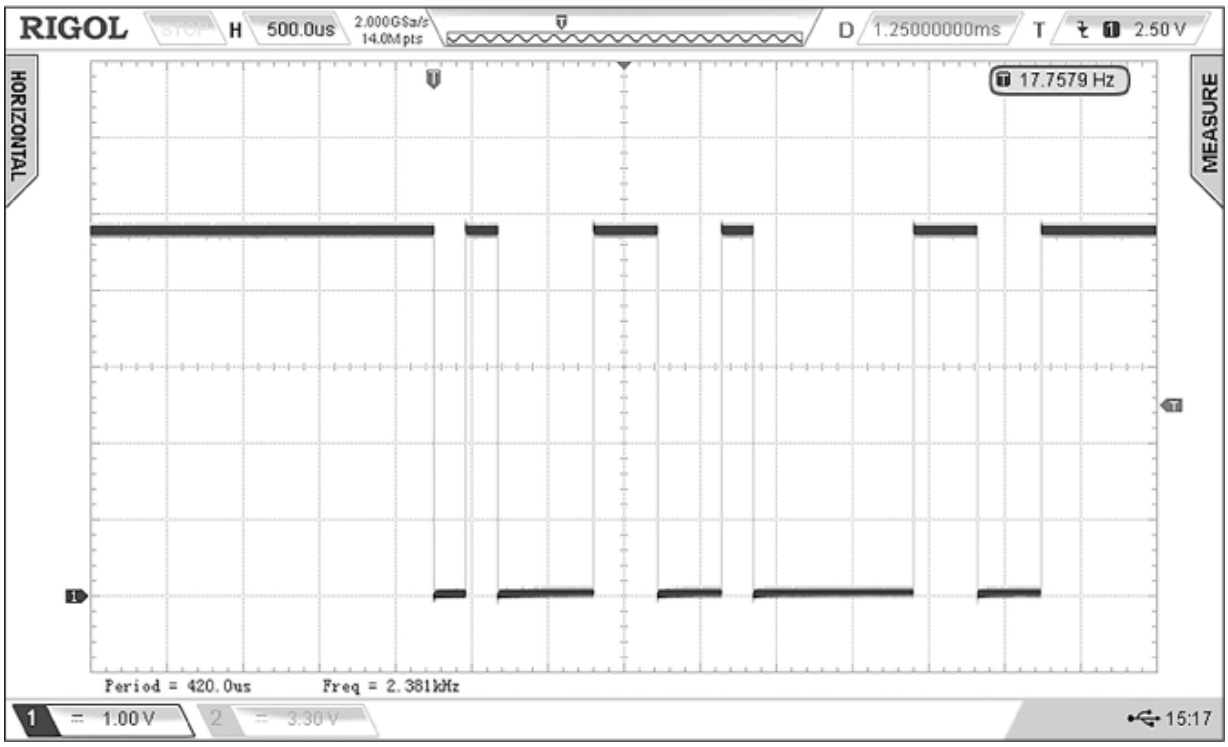


Figure 18-1: Data being sent from an Arduino serial port

Now consider the same piece of data being sent along RS485, as shown in Figure 18-2. RS485 uses two wires, A and B. When a 1 is sent, the A line has a voltage higher than the B line, and when a 0 is sent, the A line has a voltage lower than the B line. The voltages used can vary between −7 V and 12 V DC or more, depending on the RS485 interface hardware used.

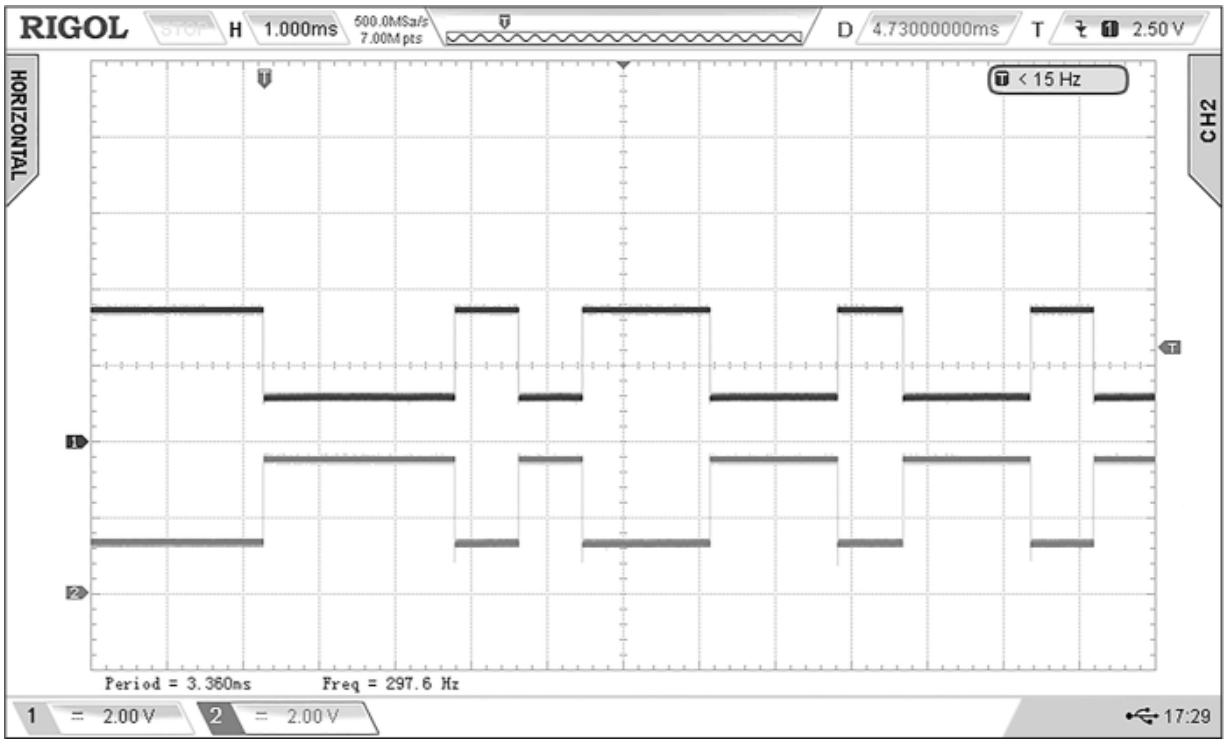


Figure 18-2: Data being sent from an RS485 interface

Using different voltages for ones and zeros over two wires enables clear definition of the two bits, removing chance of confusion and thus increasing data accuracy. Thanks to the wide voltage range, the length of the RS485 cable has less effect on the signal integrity, as there's more scope for a change in signal voltage due to cable length (the longer the cable, the larger the voltage drop over the cable due to the resistance of wire). Combined with the use of shielded cable, this voltage range means you can use RS485 over much longer distances than other data buses, such as RS232.

Connecting to RS485

The basic RS485 interface board or module has a transmit terminal (called RO) and receive terminal (called DI) used for communication between the Arduino, along with power (usually 5V) and GND. An additional signal from the microcontroller to the RS485 interface, usually labeled DE/RE, indicates whether data will be received or transmitted. On some modules, DE and RE are separate pins which you must connect with one other. Finally, as described in the previous section, connection between the RS485 interfaces uses two wires, A and B. This allows for *half-duplex communication*, in which data travels in one direction at a time.

NOTE

The RS485 bus also supports full-duplex communication over four wires, but describing this in detail is outside the scope of this book.

You can see all the pins clearly labeled on an inexpensive RS485 module, such as PMD Way part number 600197, shown in Figure 18-3. These modules can be placed conveniently in

solderless breadboards for experimenting or rapid prototyping. You'll use them in projects throughout this chapter.

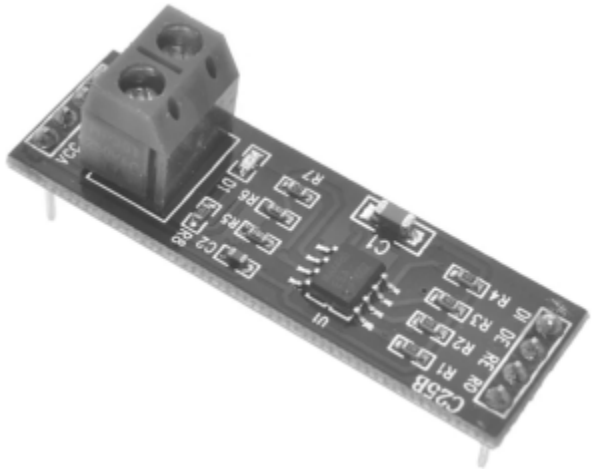


Figure 18-3: An inexpensive RS485 interface module

Figure 18-4 shows the schematic symbol used for these modules.

The 5V and GND pins in the schematic connect to a power source. The A and B pins of the interface modules connect to other interface modules; A pins are connected to A pins, and B pins to B pins. The DI (data in) pin receives data from the Arduino and sends it out along RS485. The R0 pin sends data received over RS485 to the Arduino. Finally, use the DE and RE pins in your circuits to set the interface module to transmit or receive data, setting them HIGH to transmit and LOW to receive data, as you'll do in the following project.

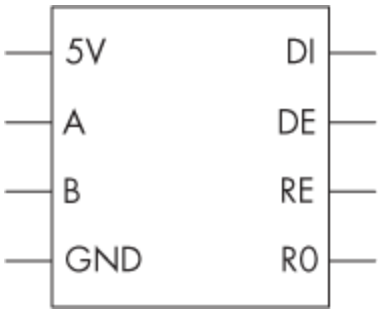


Figure 18-4: The RS485 interface module schematic symbol

Project #51: Creating an Arduino-to-PC Data Link

This project demonstrates how easy it is to send data from one Arduino, to another Arduino, to a PC with the RS485 bus. You can also use it to capture data to a PC from a remote Arduino.

You'll need the following parts:

- Two Arduino Uno or compatible boards and one USB cable
- Two RS485 interface modules
- Two solderless breadboards
- Assorted jumper wires
- A suitable power supply for an Arduino Uno or compatible board
- A twin-core wire for a long-distance communication (optional)

As with all the projects in this chapter, you'll create two Arduino-to- RS485 circuits: a receiver circuit and a transmitter

circuit. The transmitter's Arduino will send data to the receiver's Arduino via RS485, and that data will then travel to the PC via the usual USB connection. You may find it easier to upload the receiver and transmitter sketches for this project to their respective Arduinos before constructing the circuits, as it is then easier to build out the circuits, especially over a distance.

If you want to run a long RS485 cable, use shielded twin-core cable and use the cable shield to connect the GND of the transmitter and receiver. However, you can also test this project with short hook-up wire.

Figure 18-5 shows the schematic diagram for the receiver circuit.

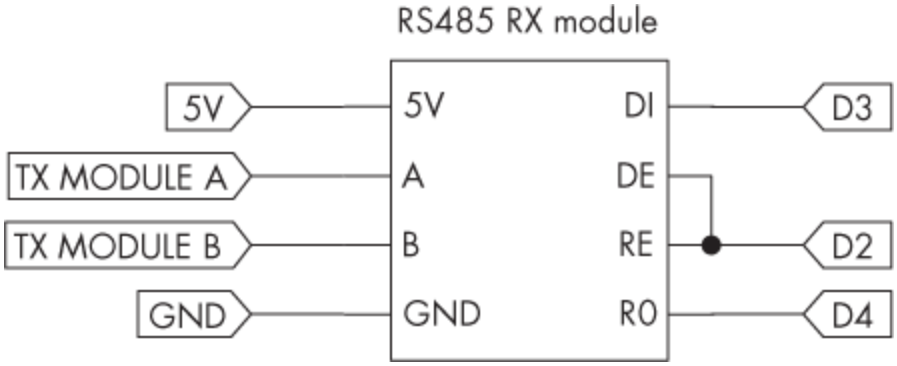


Figure 18-5: The Project #51 receiver circuit schematic

Figure 18-6 shows the schematic diagram for the transmitter circuit.

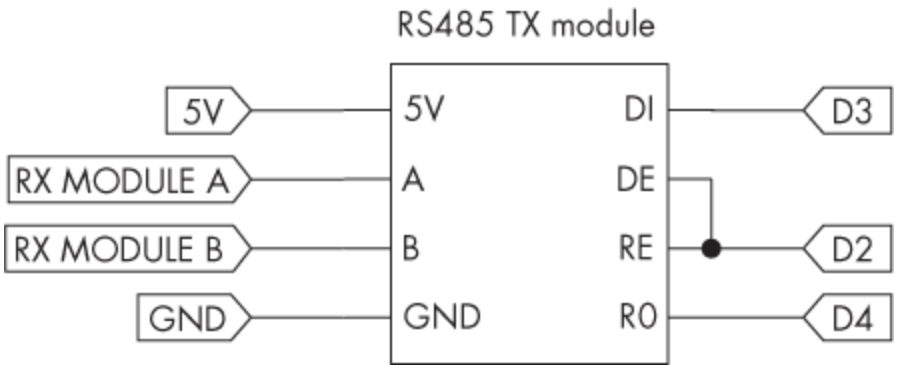


Figure 18-6: The Project #51 transmitter circuit schematic

Once you've built the circuits and uploaded the sketches, connect the receiver unit to the PC via USB and open the Arduino's Serial Monitor or terminal software such as CoolTerm (as first explained in Chapter 14). The terminal should display values for analog pin 0 on the transmitter circuit's Arduino, as shown in Figure 18-7.

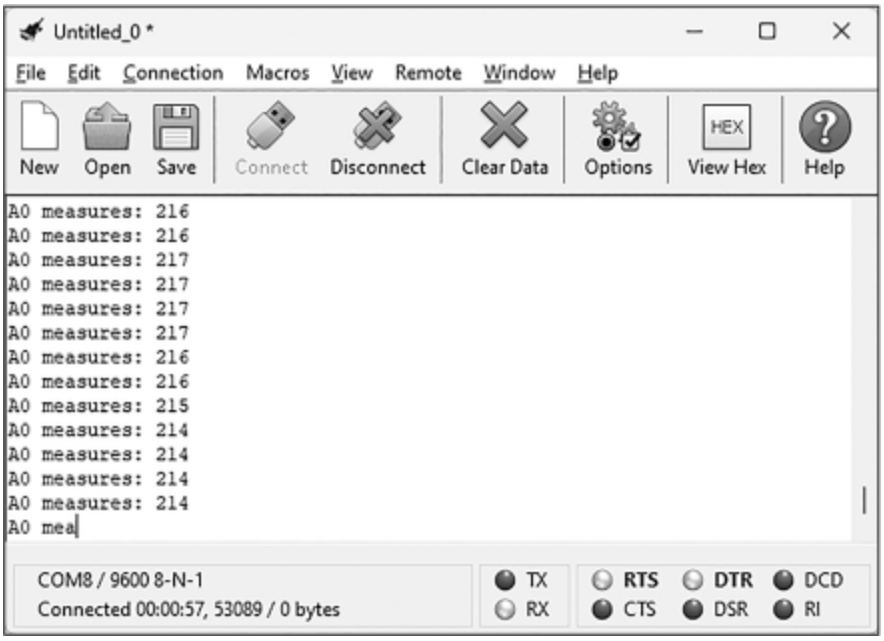


Figure 18-7: Example output from Project #51

Let's see how this works, first taking a closer look at the sketch for the transmitter circuit:

```
  // Project #51 - Transmitter

❶ #include <SoftwareSerial.h>
  SoftwareSerial RS485(4, 3);

❷ #define TXRX 2 // Using D2 for RS485 DE/RE

  void setup()
  {
      RS485.begin(9600);
      pinMode(TXRX, OUTPUT);
      digitalWrite(TXRX, HIGH); // RS485 transmit
  }

  void loop()
  {
      RS485.print("A0 measures: ");
      RS485.println(analogRead(0));
  }
```

The sketch configures software serial to communicate with the RS485 interface modules using digital pins D4 and D3 ❶. It uses Arduino digital pin D2 to control the DE/RE pins that determine

the data direction on the RS485 bus ❷. In `void setup()`, the sketch starts the software serial that will communicate with the RS485 module, then sets pin D2 to output to control the data direction on the RS485 module. It then sets the direction to transmit. Finally, in `void loop()`, the sketch continuously sends the example data in the form of the value for analog input 0 along the RS485 bus via software serial.

Now consider the receiver circuit sketch:

```
// Project #51 - Receiver

#include <SoftwareSerial.h>
SoftwareSerial RS485(4, 3);

#define TXRX 2 // Using D2 for RS485 DE/RE

void setup()
{
    Serial.begin(9600);
    pinMode(TXRX, OUTPUT);
 ❶ digitalWrite(TXRX, LOW); // RS485 receive
}

void loop()
{
    if (RS485.available())
```

```
    {
      ❷ Serial.write(RS485.read());
    }
}
```

Configuration for the receiver sketch is identical to the transmitter sketch, except that the program sets the RS485 module to receive mode by setting digital pin D2 to LOW ❶. The Arduino then waits for a character of text to come from the RS485 bus via software serial. When a character arrives, the Arduino sends it to the attached computer via hardware serial and USB ❷. You can use this project to send any type of data generated by devices connected to an Arduino over an RS485 data link and back to a PC for recording.

The next project demonstrates a way of viewing data received over an RS485 data link that is independent of using a computer for operation.

Project #52: Creating an Arduino-to-Arduino Data Link

In this project, you'll build an RS485 data link that sends temperature data from a BMP180 sensor over the bus, to be displayed using an I^2C PCF8574 LCD module. Apart from being

another good demonstration of using the RS485 data bus, this project serves as a framework for building remote real-time monitoring systems that don't require a host computer at the receiving end, such as operating temperature, machine speed, or other sensor data.

NOTE

If you're not familiar with I^2C PCF8574 module LCDs, review their use as described in Chapter 13. I also cover the BMP180 sensor in Chapter 10.

You'll need the following parts for this project:

- Two Arduino Uno or compatible boards and one USB cable
- Two RS485 interface modules
- Two solderless breadboards
- Assorted jumper wires
- Two suitable power supplies for an Arduino Uno or compatible board
- A PCF8574 LCD module
- A BMP180 temperature and air pressure sensor board
- A twin-core wire for a longer distance communication (optional)

You'll again build two Arduino-to-RS485 circuits, a receiver and a transmitter. In this project, the transmitter's Arduino will send temperature data to the receiver's Arduino via RS485 to be displayed on the LCD. Again, you may find it easier to upload the two sketches for this project to their respective Arduinos before constructing the circuits.

Figure 18-8 shows the schematic diagram for the receiver circuit.

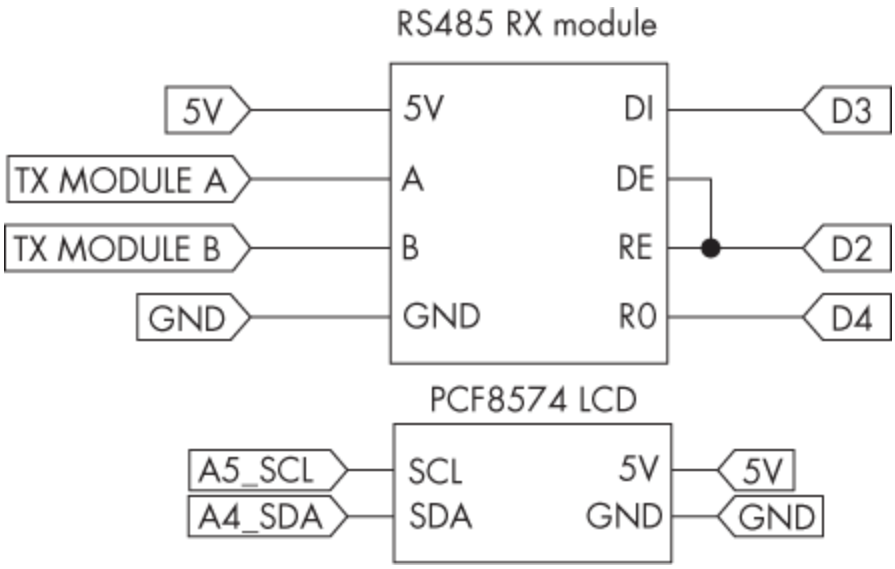


Figure 18-8: The Project #52 receiver circuit schematic

Figure 18-9 shows the schematic for the transmitter circuit.

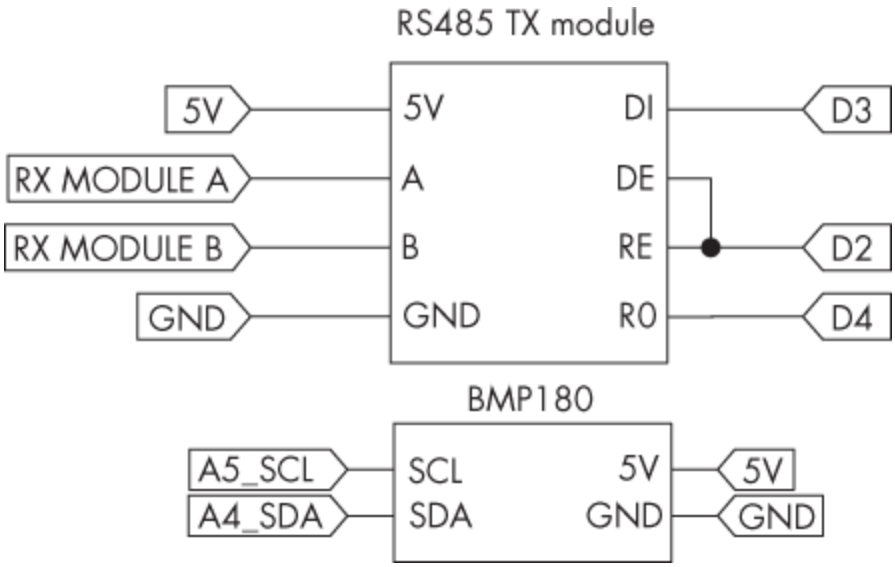


Figure 18-9: The Project #52 transmitter circuit schematic

Once you have uploaded the sketches to their respective Arduino boards and the circuits are complete, power both circuits up. The current ambient temperature should appear on the LCD, as shown in Figure 18-10.



Figure 18-10: Example output from Project #52

Let's see how this works, first taking a closer look at the transmitter sketch:

```
  // Project #52 - Transmitter

❶ #include <Adafruit_BMP085.h>
  Adafruit_BMP085 bmp;

❷ #include <SoftwareSerial.h>
  SoftwareSerial RS485(4, 3);

❸ #define TXRX 2 // Using D2 for RS485 DE/RE

  void setup()
  {
      RS485.begin(9600);
      pinMode(TXRX, OUTPUT);
      digitalWrite(TXRX, HIGH); // RS485 transmit
      bmp.begin();
  }

  void loop()
  {
      RS485.print(bmp.readTemperature());
      delay(500);
  }
```

The sketch first includes the library required to use the BMP180 temperature sensor and creates the instance ❶. Again, the

sketch configures software serial to communicate with the RS485 interface modules using digital pins D4 and D3 ❷. The sketch uses Arduino digital pin D2 to control the DE/RE pins that determine the data direction on the RS485 bus ❸.

In `void setup()`, the sketch starts the software serial that will communicate with the RS485 module, then sets pin D2 to output to control the data direction on the RS485 module. It then sets the direction to transmit and starts the BMP180 sensor. Finally, in `void loop()`, it sends temperature data from the BMP180 sensor every half a second along the RS485 bus via software serial.

Now consider the receiver circuit sketch:

```
  // Project #52 - Receiver

❶ #include <Wire.h>
  #include <LiquidCrystal_PCF8574.h>
  LiquidCrystal_PCF8574 lcd(0x27);

  #include <SoftwareSerial.h>
  SoftwareSerial RS485(4, 3);

  #define TXRX 2 // Using D2 for RS485 DE/RE
```

```
float temperature = 0;

void setup()
{
    Wire.begin();
    Wire.beginTransmission(0x27);
 ❷ lcd.begin(16, 2);
 ❸ lcd.setBacklight(255);
 ❹ lcd.clear();
    RS485.begin(9600);
    pinMode(TXRX, OUTPUT);
    digitalWrite(TXRX, LOW); // RS485 receive
}

void loop()
{
    byte incoming = RS485.available();
 ❺ if (incoming != 0)
    {
        temperature = RS485.parseFloat();
     ❻ lcd.home();
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Temperature:");
        lcd.setCursor(0, 1);
        lcd.print(temperature);
        lcd.print(" Celsius");
```

```
    }
}
```

Since the receiver uses the PCF8574 I^2C LCD, the required library for the I^2C bus, the sketch includes the LCD display and instance ❶. In `void setup()`, it starts the I^2C bus and begins communication to the LCD in order to configure the LCD display size ❷, turn on the backlight ❸, and clear the display ❹. The sketch then includes the RS485 configuration for the receiver sketch. This is identical to that of the transmitter sketch, except that the sketch sets the RS485 module to receive mode by setting digital pin D2 to LOW.

The Arduino waits for text to come from the RS485 bus via software serial ❺. Once the text arrives, the sketch converts it into a floating-point number using the `.parsefloat()` function from the serial library. The temperature information is then displayed on the LCD ❻. The process repeats when more temperature data arrives from the transmitter circuit.

Project #53: Remote Control Operation

Since it supports longer cable runs, the RS485 bus also lends itself well to remote-control applications. This project demonstrates remote-control operation over RS485, in which

one Arduino circuit (the primary) sends signals to another Arduino (the secondary) over RS485 to take various actions.

You'll need the following parts for this project:

- Two Arduino Uno or compatible boards and one USB cable
- Two RS485 interface modules
- Two solderless breadboards
- Assorted jumper wires
- Two suitable power supplies for an Arduino Uno or compatible board
- A twin-core wire for a longer distance communication (optional)

In this project, the primary Arduino sends single characters to the secondary Arduino via RS485. You'll program the secondary Arduino to take various actions depending on the character received; if an unknown command is received, a default action will occur. Again, you may find it easier to upload the sketches to each Arduino before constructing the circuits.

Figure 18-11 shows the schematic diagram for the primary circuit.

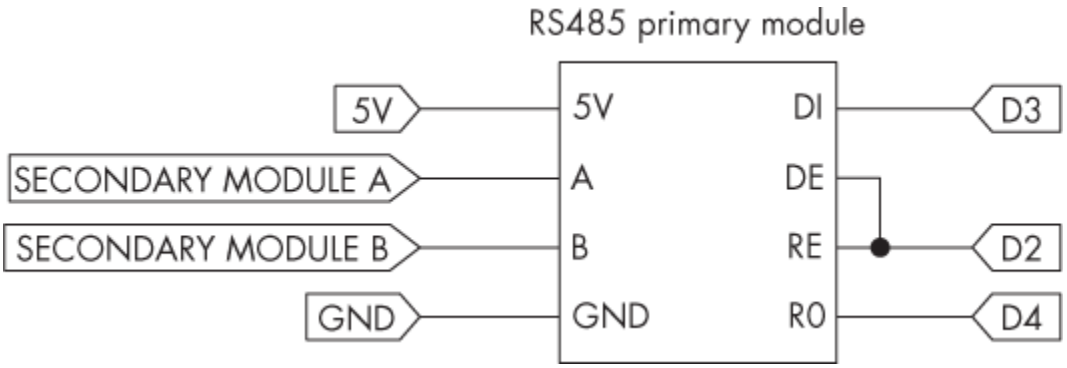


Figure 18-11: The Project #53 primary circuit schematic

Figure 18-12 shows the schematic diagram for the secondary circuit.

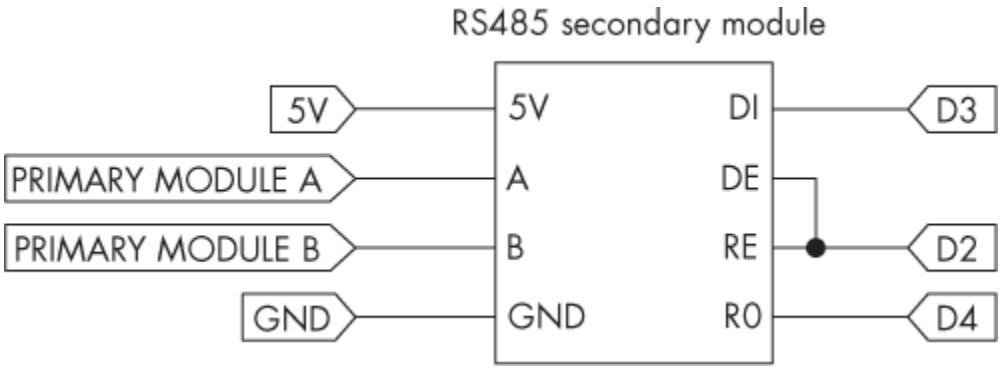


Figure 18-12: The Project #53 secondary circuit schematic

Once you have uploaded the project sketches to their respective Arduino boards and built the circuits, power them both up. The primary Arduino should start sending commands to the secondary, which will act on those commands by blinking the Arduino's onboard LED in various sequences.

Let’s see how this works, first examining the sketch for the primary circuit:

```
// Project #53 - Primary

#include <SoftwareSerial.h>
SoftwareSerial RS485(4, 3);

#define TXRX 2 // Using D2 for RS485 DE/RE

void setup()
{
    RS485.begin(9600);
    pinMode(TXRX, OUTPUT);
    digitalWrite(TXRX, HIGH); // RS485 transmit
}

void loop()
{
    RS485.print("0");
    delay(1000);
    RS485.print("1");
    delay(1000);
    RS485.print("2");
    delay(1000);
    RS485.print("3");
    delay(1000);
    RS485.print("8");
```

```
    delay(5000);
}
```

This sketch configures the primary circuit to transmit over RS485, as did previous projects. The point of difference is the example commands sent in `void loop()`. The secondary Arduino is configured to act upon the commands 0, 1, 2, and 3, so the primary sketch sends those commands as a demonstration. It also sends the number 8 to demonstrate the function on the receiver which deals with unknown commands. Each command is sent in turn with a delay between each.

Here's the sketch for the secondary circuit:

```
// Project #53 - Secondary

#include <SoftwareSerial.h>
SoftwareSerial RS485(4, 3);

#define TXRX 2 // Using D2 for RS485 DE/RE

void blinkLED(int i)
{
    for (int j = 0; j < i; j++)
    {
        digitalWrite(13, HIGH);
```

```
        delay(100);
        digitalWrite(13, LOW);
        delay(100);
    }
}

void setup()
{
    RS485.begin(9600);
    pinMode(TXRX, OUTPUT);
    digitalWrite(TXRX, LOW); // RS485 receive
    pinMode(13, OUTPUT);
}

void loop()
{
 ❶ if (RS485.available() > 0)
    {
        char receivedChar = RS485.read();
     ❷ switch (receivedChar)
        {
            case '0': blinkLED(5); break;
            case '1': blinkLED(10); break;
            case '2': blinkLED(15); break;
            case '3': blinkLED(20); break;
         ❸ default: blinkLED(2); break;
        }
```

```
    }
}
```

The secondary circuit is configured to receive commands over RS485 then act based on those commands. To demonstrate this, the sketch uses the `blinkLED()` function, which accepts an integer and blinks the onboard LED that number of times. The secondary circuit awaits a character of text (the command) from the RS485 bus via the software serial port ❶. The secondary unit uses a `switch…case` function ❷ to determine which function to call, depending on the command received from the primary unit. If it receives a command that isn't covered in the `switch…case` function, it blinks the LED twice ❸.

You can modify this project for your own remote-control needs by replacing the calls to the `blinkLED()` function with your own actionable requirements.

Controlling Two or More Secondary RS485 Devices

So far, you've learned how to communicate between two RS485-connected Arduino boards. As mentioned earlier, however, you can control more than one secondary device with the RS485

bus, up to 256 devices in total. In this section and the following project, I'll show you how to do so.

The wiring of the RS485 bus is slightly different when using three or more devices. The transceiver IC found on the modules you've been using also requires a 120 Ω terminating resistor across the A and B wires at each end of the RS485 bus. This resistor is fitted with the required terminating resistor, so if you're just using the two modules, you don't need to change anything. If you're using three or more of the RS485 modules covered in this chapter, though, you'll need to desolder the terminating resistor labeled R7 on the PCB (with the value 121 on the resistor, which is interpreted as 12 Ω, with a 0 added on the end for 120 Ω) from the module. This resistor is circled in Figure 18-13.

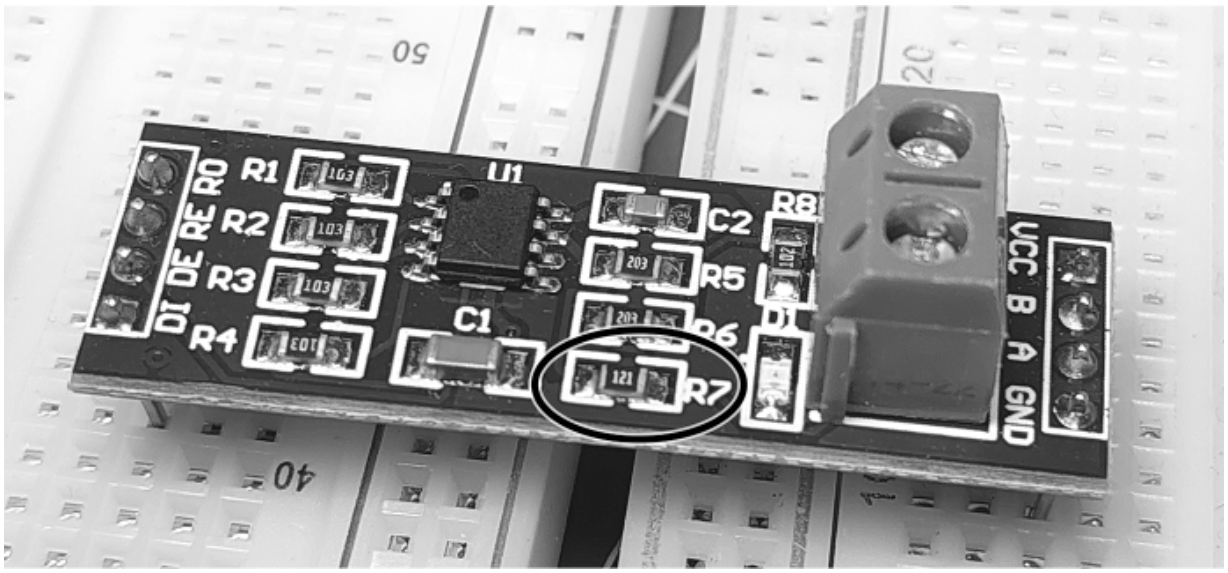


Figure 18-13: An RS485 terminating resistor

When wiring your devices over distances longer than that used for demonstration on your workbench, ensure the A/B pair of wires travel from device to device in a “daisy-chain” configuration, as shown in Figure 18-14. Don’t create a separate run from each device back to a main A/B pair.

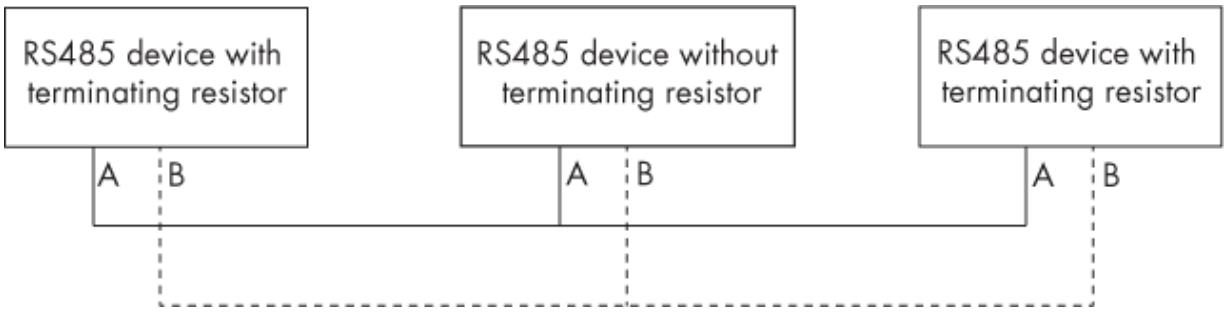


Figure 18-14: Example RS485 wiring configuration for three devices

You’ll put this technique to work in the next project.

Project #54: Controlling Two or More Secondary Arduino Boards

This project demonstrates controlling two or more secondary Arduino boards over RS485 from one primary board. One Arduino circuit (the primary) will send commands to other Arduinos (the secondary devices) over RS485, telling them to take various actions.

The hardware for this project is the same as that used in Project #53, except that you'll use two or more secondary devices. Start by building one primary device and two secondary devices, connecting them via the RS485 bus as shown in Figure 18-14 in the previous section. Again, it's a good idea to upload the sketches to each Arduino before building the circuits. You can add more secondary devices to the project later, if you like, once you've got your three-device setup working.

If you're just testing this project on your workbench, the distance of the RS485 bus will be short enough that you may not need to remove the terminating resistors from the RS485 modules, so test the code before making any changes to your hardware.

In this project, the primary board sends a three-digit number across the RS485 bus, which is received by all the secondary boards. The first digit of the command represents the number of the secondary unit to control (units 1 through 9), and the final 2 digits represent the command being sent to the specific secondary unit. For example, to send command 12 to secondary unit 2, the primary would send 212 across the RS485 bus, since the sketch loaded onto the second secondary unit is programmed to act based on receiving 12. If you add 10 or more secondary units, you can change the commands in the sketch so they send four-digit numbers, or modify the command coding method for your own needs.

This project is just a demonstration of multiple-secondary-device control over RS485. In its current form, each command sent from the primary will trigger the matching secondary device to blink its onboard LED a certain number of times. However, in your own future projects, you can replace the simple example actions in the secondary units with your own requirements.

Let’s see how this works in the sketches, starting with the primary device sketch:

```
// Project #54 - Primary

#include <SoftwareSerial.h>
SoftwareSerial RS485(4, 3);

#define TXRX 2 // Using D2 for RS485 DE/RE

void setup()
{
    RS485.begin(9600);
    pinMode(TXRX, OUTPUT);
    digitalWrite(TXRX, HIGH); // RS485 transmit
}

void loop()
{
 ❶ RS485.println(105); // Command for secondary
    delay(5000);
 ❷ RS485.println(205); // Command for secondary
    delay(5000);
}
```

In a sequence that should be familiar to you by now, the sketch first configures the primary unit to send data to the RS485 bus. In `void loop()`, it sends two new example commands: 105 ❶, which is command 5 for the secondary unit 1, and 205 ❷, which

is command 5 for secondary unit 2. Remember that the commands are three digits, with the first being the secondary unit to address, and the following two digits (00 to 99) being the actual command.

The following is the sketch for secondary device 1:

```
// Project #54 - Secondary device 1

#include <SoftwareSerial.h>
SoftwareSerial RS485(4, 3);

#define TXRX 2 // Using D2 for RS485 DE/RE

void blinkLED(int i)
{
    for (int j = 0; j < i; j++)
    {
        digitalWrite(13, HIGH);
        delay(250);
        digitalWrite(13, LOW);
        delay(250);
    }
}

void setup()
{
```

```
    RS485.begin(9600);
    pinMode(TXRX, OUTPUT);
    digitalWrite(TXRX, LOW); // RS485 receive
    pinMode(13, OUTPUT);
}

void loop()
{
    while (RS485.available() == 0) {}
 ❶ int commandType = RS485.parseInt();
 ❷ if (commandType >= 100 && commandType < 200)
    {
        switch (commandType)
        {
         ❸ case 105: blinkLED(5); break;
           case 110: blinkLED(10); break;
           case 115: blinkLED(15); break;
           case 120: blinkLED(20); break;
         ❹ default: blinkLED(2); break;
        }
    }
}
```

This sketch is configured so that commands received from the primary unit call the `blinkLED()` function with a varying parameter to blink the LED a different number of times. In `void loop()`, the secondary awaits an incoming integer from the

RS485 bus ❶. It then checks whether the received integer falls between 100 and 199 inclusive ❷. If so, meaning the command is intended for this unit, the sketch compares the integer against the preset commands using the `switch…case` function. For example, if the command 105 is received, the `blinkLED()` function is called to blink the onboard LED five times ❸. If a command is received but not recognized ❹, the LED blinks twice.

The first part of the sketch for secondary device 2 is the same as for secondary device 1. Only the `void loop()` section of the sketch differs:

```
void loop()
{
    while (RS485.available() == 0) {}
    int commandType = RS485.parseInt();
 ❶ if (commandType >= 200 && commandType < 300)
    {
        switch (commandType)
        {
          ❷ case 205: blinkLED(5); break;
            case 210: blinkLED(10); break;
            case 215: blinkLED(15); break;
            case 220: blinkLED(20); break;
          ❸ default: blinkLED(2); break;
```

```
            }
        }
    }
```

The sketch tests the received command to see whether it falls between 200 and 299 inclusive ❶, as the first digit of the command (2) notes the command is for this secondary unit. If the command is for this unit, the sketch compares it against the preset commands using the `switch…case` function. For example, if the device receives the command 205, the sketch calls the `blinkLED()` function to blink the onboard LED five times ❷. If a command is received but not recognized ❸, the LED blinks twice.

To add additional secondary units, you'd simply change the test line ❶ so the new unit can check whether a command from the primary device falls within its required numeric range. You'd also need to add the appropriate commands and required actions in the `switch…case` function.

For a challenge, you might try creating a system that returns data from the secondary devices back to the primary device.

Moving On

This chapter has given you the basic building blocks for implementing the RS485 data bus with Arduino. You can use this knowledge to make your own data transfer and control applications over longer distances, such as monitoring sensors or controlling devices in other rooms in an office or factory.

In the next chapter, you'll get started with popular Wi-Fi-capable Arduino-compatible boards to make remote-control devices that can be controlled from a cell phone or any web-enabled device.

19

THE ESP32 MICROCONTROLLER PLATFORM AND IOT



The phrase *Internet of things (IoT)* describes a wide variety of devices that communicate or are controlled over the internet. In the past, building IoT Arduino projects that communicate using Wi-Fi was complex and expensive, as Wi-Fi shields for Arduino were cumbersome and had limited features when combined with the Arduino's microcontroller. However, this is no longer the case, thanks to the Espressif ESP32 microcontroller platform, which has built-in Wi-Fi and Arduino compatibility.

We'll use ESP32-based boards in the final six chapters of this book. For the best learning experience, you may want to read these chapters in order; however, you can still skip around between them if you prefer.

In this chapter, I'll introduce you to the ESP32-based Arduino-compatible development board. You'll learn to install new board profiles in the Arduino IDE, test ESP32 Wi-Fi connectivity, and use PWM with the ESP32 board. You'll learn to:

- Create a simple web page with buttons
- Build a remote control for four outputs with state display
- Build a remote-control-adjustable PWM output

You'll use the skills you develop in this chapter to work with other internet-connected devices in the later chapters of this book.

The ESP32

The ESP32 is a microcontroller with built-in Wi-Fi and Bluetooth, as well as a higher operating speed than a typical Arduino or compatible board. It's not a simple one-chip configuration; instead, it requires an external flash memory IC

and an antenna for the wireless operations. The IC and antenna are usually grouped closely together, as shown in Figure 19-1.

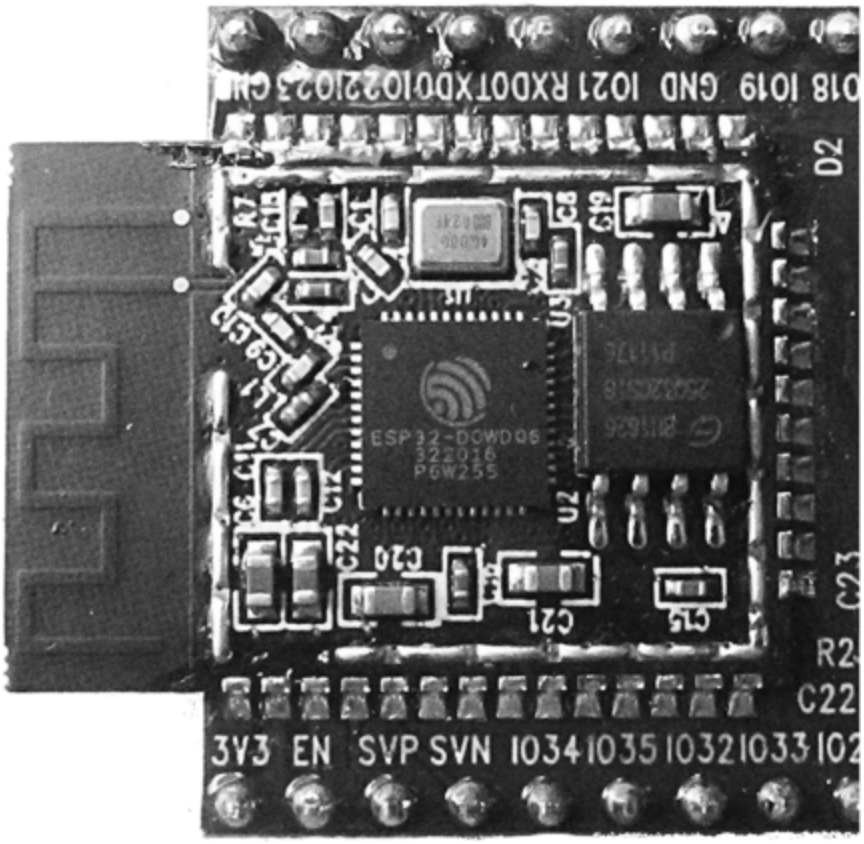


Figure 19-1: Example ESP32 chipset in a module with the top can removed

To avoid signal interference, the required circuitry is supplied in a module, which has a metal can on top. This module is then fitted to boards with the circuitry used to interface with other devices or is added to boards to become Arduino-compatible, as shown in Figure 19-2.

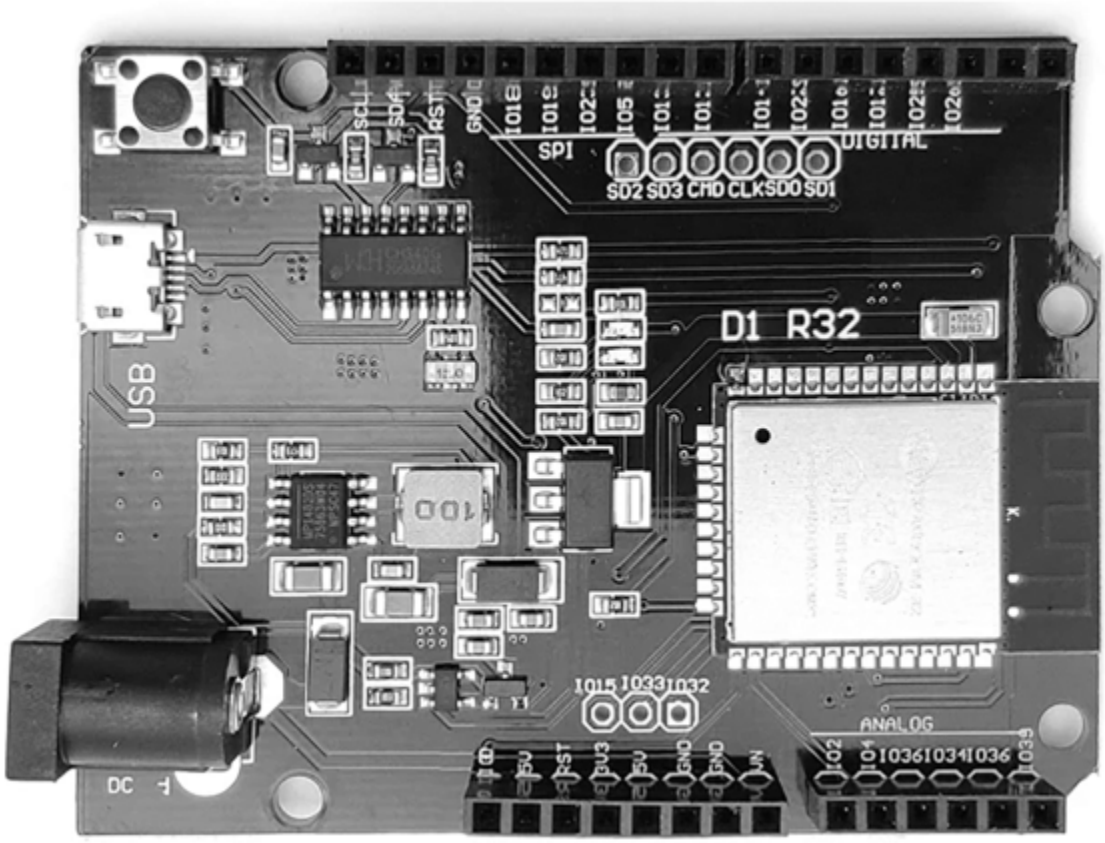


Figure 19-2: An ESP32 development board

Boards like PMD Way part 51045204, shown in Figure 19-2, are known as *ESP32 development boards* (or *dev boards*). You'll use this board for projects from this chapter through Chapter 24. Make sure to order the required USB cable if it isn't included with the board you purchase.

There are four major differences between the Arduino Uno and the ESP32 dev board. The first is the large external flash memory IC; the ESP32 offers much more space for Arduino

code than offered by an 8-bit Arduino. The second difference is the faster CPU speed of up to 240 MHz.

The third difference is that the ESP32 is a 3.3 V device, so all connections to external devices via input or output pins must either operate at 3.3 V or be 3.3 V tolerant. If the connection isn't at this voltage by default, you'll need to use a level converter such as PMD Way part 441079, shown in Figure 19-3. Connect the converter between the wires from the 5 V device to the HV pads and the matching lower voltage wires on the LV pads to the ESP32, and connect GND of both sides to the ESP32 and the 5 V device.



Figure 19-3: A four-channel level converter board

When shopping for a level converter, get a four-channel unit like the unit in Figure 19-3. That way, you can use both SPI and

I²C data buses without running out of converter pins.

The final difference between the Arduino Uno and the ESP32 is that, while the ESP32 shares the same physical form factor as an Uno, the pinouts are different, as shown in Table 19-1. This also means that some Arduino shields may not work with the ESP32 board, so investigate the required connections of a shield before purchasing to use with your ESP32.

Table 19-1: ESP32 and Arduino Pinout Comparison

Arduino label	**Uno R3**	**ESP32**	**Arduino label**	**Uno R3**	**ESP**
I²C clock	SCL	SCL	Digital I/O	D3	IO25
I²C data	SDA	SDA	Digital I/O	D2	IO26
Analog reference	AREF	RESET	Digital I/O	D1	TX only

Arduino label	Uno R3	ESP32	Arduino label	Uno R3	ESP
GND	GND	GND	Digital I/O	D0	RX only
Digital I/O	D13	IO18	Analog input	A5	IO39
Digital I/O	D12	IO19	Analog input	A4	IO36
Digital I/O	D11	IO23	Analog input	A3	IO34
Digital I/O	D10	IO5	Analog input	A2	IO35
Digital I/O	D9	IO13	Analog input	A1	IO4
Digital I/O	D8	IO12	Analog input	A0	IO2

Arduino label	Uno R3	ESP32	Arduino label	Uno R3	ESP
Digital I/O	D7	IO14	Board voltage	IOREF	IO0
Digital I/O	D6	IO27	Reset MCU	RESET	RESE
Digital I/O	D5	IO16	3.3 V output	3.3V	3.3V
Digital I/O	D4	IO17	5 V output	5V	5V
			GND	GND	GND
			GND	GND	GND
			Raw voltage in	Vin	Vin

ESP32 pins labeled IO*xx* are (3.3 V) digital input and output pins. When addressing these pins in your Arduino sketches, don't include the IO, just use the number. I'll demonstrate that shortly.

NOTE

Port manipulation doesn't work with the ESP32 board. Pins may also have other functions beyond those shown in the table, but for the purposes of the projects in this chapter, you'll use only the functions shown in the table.

Configuring the Arduino IDE for the ESP32

The Arduino IDE doesn't come with the ESP32 boards preinstalled, so you'll need to install the required files. With your ESP32 on hand, open the Arduino IDE and select **File** ▸ **Preferences**. When the Preferences dialog appears, click the small button next to the Additional Boards Manager URLs field, and the Additional Boards Manager URLs dialog should appear, as shown in Figure 19-4.

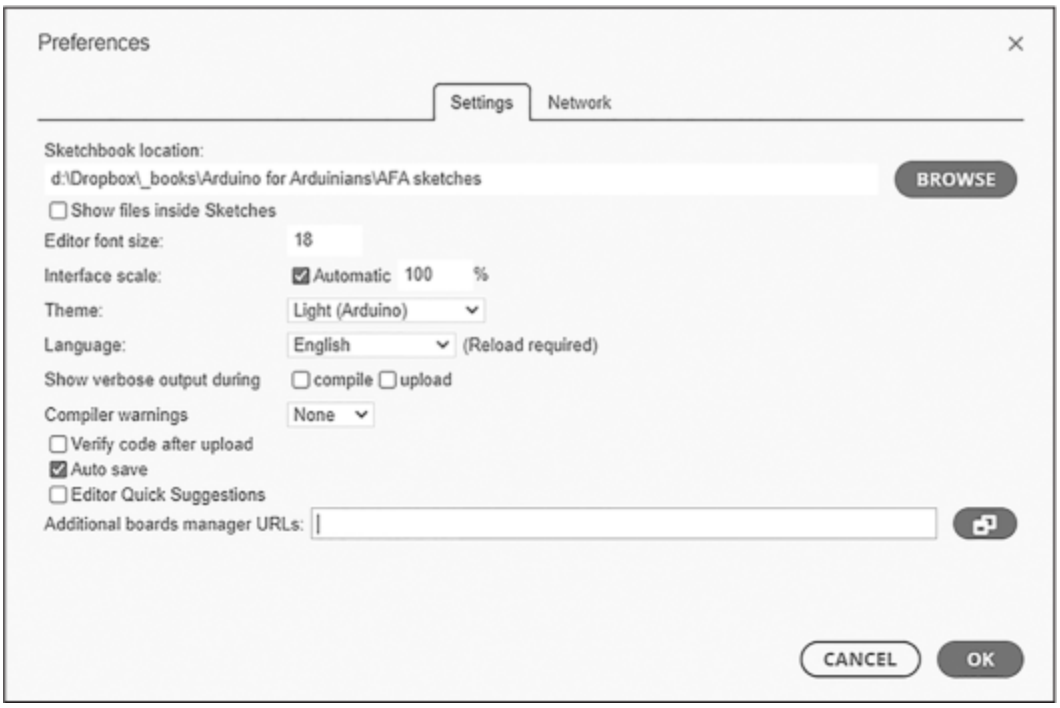


Figure 19-4: The Arduino IDE Additional Boards Manager URLs dialog

Enter the following URL into the dialog:

```
https://dl.espressif.com/dl/package_esp32_index.
```

If you already have other URLs in the dialog, place a comma at the end of the last URL and paste the new one after the comma.

Click **OK** to close the Preferences dialog. Now close and reopen the IDE. If your computer is not connected to the internet, connect it now. Select **Tools** ▸ **Boards Manager**. When the Boards Manager opens, enter **ESP32** in the search dialog. The

ESP32 package should appear. Always use the latest version available.

Click **Install** and wait a few moments for the process to complete, as shown in the progress bar at the bottom of the Boards Manager dialog.

Since the ESP32 uses the CH340 USB-to-serial interface, which differs from that used on an Arduino Uno or compatible board, you'll need to install drivers for the interface for your computer. To do this, follow the guide provided by SparkFun at *https://learn.sparkfun.com/tutorials/how-to-install-ch340-drivers/all*. Once you've installed the driver, connect your ESP32 dev board to the PC via a USB cable.

If you don't already have the latest version of Python on your PC, you'll also need to download and install it, since the Arduino IDE requires it as part of the toolchain to upload code to an ESP32 board. If necessary, visit *https://www.python.org/downloads* and follow the links for your operating system and install.

After installing Python, macOS users may need to open the terminal and run the following command before re-opening the Arduino IDE:

```
Open -a Arduino
```

Now open the Arduino IDE, select **Tools**, and change all the settings from Board onward to match those shown in Figure 19-5.



Figure 19-5: Board settings in Arduino IDE for the ESP32

The only parameter that may vary on your computer is the USB port; select the appropriate port shown in that option.

Testing the ESP32

Before moving on to the first project of the chapter, test the dev board to check you can upload a sketch, operate GPIO pins, and operate your Wi-Fi.

The GPIO Pins

First, try controlling some GPIO pins by building the classic test circuit that blinks some LEDs. You will need the following:

- An ESP32 dev board and matching USB cable
- A solderless breadboard
- Assorted jumper wires
- Four LEDs
- One 560 Ω, 0.25 W, 1 percent resistor

Assemble the circuit as shown in Figure 19-6.

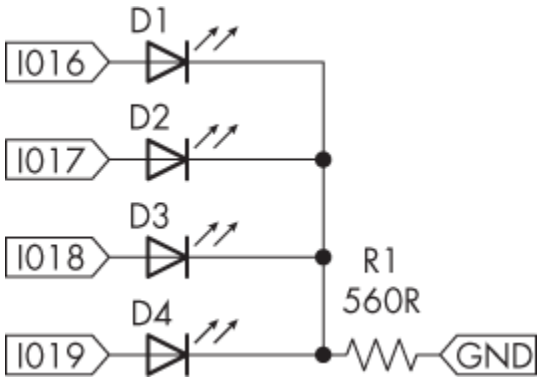


Figure 19-6: A basic four-LED circuit for testing purposes.

Now enter and upload Listing 19-1 to your ESP32 dev board. A moment after uploading the sketch, each LED should turn on and off in turn.

```
void setup()
{
  ❶ pinMode(16, OUTPUT);
    pinMode(17, OUTPUT);
    pinMode(18, OUTPUT);
    pinMode(19, OUTPUT);
}

void blinkLEDs()
{
    int i;
    int d = 250;
    for (i = 16; i < 20; i++)
    {
      ❷ digitalWrite(i, HIGH);
        delay(d);
      ❸ digitalWrite(i, LOW);
    }
}

void loop()
{
    blinkLEDs();
}
```

Listing 19-1: Testing outputs from the ESP32 dev board

Although the hardware descriptions for the GPIO pins are IO*xx*, as shown in Table 19-1, the sketch references the pins by just their numerical part identifiers when setting up the GPIO pin status ❶ and addressing them to turn on and off ❷ ❸.

If your sketch didn't upload, check the settings for the board type and parameters as described in the previous section. Once you've finished this test, keep the circuit assembled, as you'll use it for the following projects.

The Wi-Fi Connectivity

You'll now test the ESP32's Wi-Fi connectivity, using the WiFiScan sketch included with the ESP32 board profile that scans for the existence of Wi-Fi networks in your immediate vicinity.

You won't need any extra hardware beyond the board, the USB cable, and a Wi-Fi network. Upload the sketch from the IDE's examples via **File** ▸ **Examples** ▸ **WiFi** ▸ **WiFiScan**, then open the Serial Monitor in the Arduino IDE. After a few moments, the board should search for the existence of Wi-Fi access points and list them, along with their signal strength, as shown in Figure 19-7.



Figure 19-7: Example results of scanning for Wi-Fi access points

The Serial Monitor simply shows the access point names, not whether the connection is open. If you'd like more information on the meaning of the signal strength, visit *https://en.wikipedia.org/wiki/Received_signal_strength_indication*.

Port Forwarding

To control your project from a device that isn't connected to your local network, such as a tablet or phone using a cellular connection that doesn't have a static IP address, you'll need to use a technique called *port forwarding* in your network router or modem. Port forwarding gives you a static IP address from an external provider that then diverts data directly to your device.

Port forwarding services are provided by organizations such as No-IP (*https://www.noip.com*) and Dyn (*https://account.dyn.com*). Setup is often unique to the make and model of your router, so do an internet search for *router port forwarding* for your model of router. Alternatively, you could ask your ISP, if it provided the router, or visit a tutorial site such as *http://www.wikihow.com/Port-Forward* for more information.

Once you've set up port forwarding or otherwise confirmed your static IP address, you're ready to start building a framework for remote-control applications. However, if you can't organize port forwarding, you can still control your projects from your local-area Wi-Fi network.

Project #55: Remote-Controlling a Single GPIO Pin

In this project, you'll remotely control a GPIO pin on your ESP32 dev board via any device with a web browser. This includes building a simple web page hosted by the ESP32 using HTML and CSS. You can then use this page to control anything you normally would control with an Arduino 3.3 V (or 5 V with a level converter) digital output, such as a relay, MOSFET, LED, or so on.

For the hardware, use the circuit for testing the GPIO pins described in the previous section. Enter and upload the Project #55 sketch to your ESP32 dev board. Don't forget to update your Wi-Fi network credentials in the sketch before uploading. A moment after that, open the Serial Monitor in the Arduino IDE to watch the ESP32's progress as it connects to the Wi-Fi network. The sketch should then display your project's IP address, as shown in Figure 19-8.

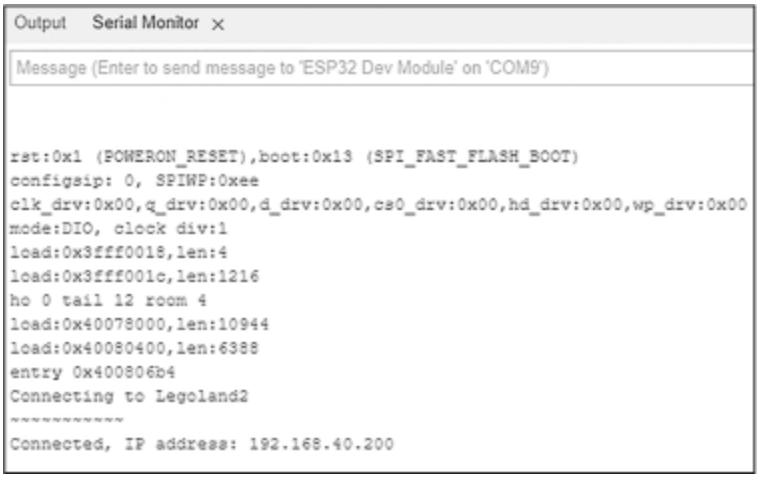


Figure 19-8: The Serial Monitor showing that the ESP32 has connected to the Wi-Fi network

Next, using the web browser on a web-enabled device, visit the IP address shown in the Serial Monitor. You should see the simple web page hosted by the ESP32 dev board, as shown in Figure 19-9.

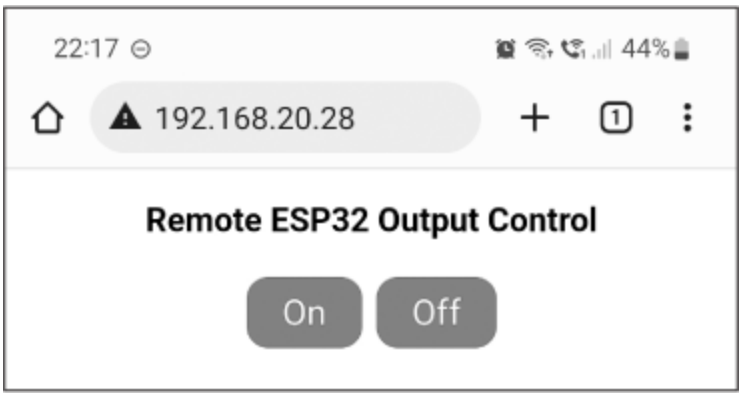


Figure 19-9: A remote-control web page

When you click the On or Off buttons, the LED connected to GPIO 16 should turn on and off. For this project, the ESP32 dev board has two jobs: to act as a web server that hosts the simple web page for the user interface (including the On/Off buttons) and to respond to HTML requests from the web browser (to turn the GPIO pin on and off).

Let's take a closer look at how this works:

```
// Project #55 - Digital output control via web

#include <WiFi.h> ❶
WiFiServer server(80); // Set web server port to

// Store your Wi-Fi network credentials:
char* ssid = "name"; // Insert name of Wi-Fi acces
char* password = "password"; // Insert password
```

```
String request; // Stores HTTP request from clie
int outputPin = 16; // Using GPIO pin 16 on the

unsigned long currentTime = millis(); ❷
unsigned long previousTime = 0;
const long timeoutTime = 2000; // Allowed client

void setup()
{
    Serial.begin(115200);
    // Set up digital pin to control
    pinMode(outputPin, OUTPUT);
    digitalWrite(outputPin, LOW);

    // Connect to Wi-Fi network:
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    // Show indicator while waiting for connecti
    {
        delay(500);
        Serial.print("~");
    }
    Serial.println(); // Display local IP addres
    Serial.print("Connected, IP address: ");
    Serial.println(WiFi.localIP());
    // Start web server
```

```
    server.begin();
}

void loop()
{
    // Listen for incoming clients from web brow
    WiFiClient client = server.available();
    if (client) // If a request received
    {
        currentTime = millis();
        previousTime = currentTime;
        Serial.println("New client connected");
        String currentLine = " ";
        while (client.connected() && currentTime
        {// Stay connected for timeoutTime
            currentTime = millis();
            if (client.available())
            {
                // Display data from client on S
                char c = client.read(); ❹
                Serial.write(c);
                request += c;
                if (c == '\n') // Client request
                {
                    if (currentLine.length() == 
                    {
                        // Send HTTP response to
                        client.println("HTTP/1.1
```

```
                client.println("Content-
                client.println("Connecti
                client.println();
                // Now do things based o
                if (request.indexOf("GET
                {
                    // If request was IP
                    Serial.println("outp
                    digitalWrite(outputP
                } else if (request.index
                // If request was IP add
                {
                    Serial.println("outp
                    digitalWrite(outputP
                }
                // Build web page for di
                client.println("<!DOCTYP
                client.print("<head><met
                client.println(" content
                client.println("<link re
                // Build buttons using C
                client.print("<style>htm
                client.print("inline-blo
                client.println();
                client.print (".button {
                client.println("none; co
                client.print ("text-deco
                client.println("2px; cur
```

```
                    client.println("</style><
                    // Now HTML for layout:
                    client.println("<body><h1
                    // Display "On" button wi
                    client.print ("<p><a href
                    client.println("class=\"b
                    // Display "Off" button w
                    client.print ("<a href=\
                    client.println("=\"button
                    client.println("</body></
                    // End the response to cl
                    client.println();
                    break;
                } else
                {// If you got a newline, the
                    currentLine = " ";
                }
            } else if (c != '\r')
            {
                currentLine += c;
            }
        }
    }
    request = " "; ❼
    // Close connection to client:
    client.stop();
    Serial.println("Disconnected.");
```

```
        }
    }
```

First, the sketch includes the Arduino Wi-Fi library ❶ and creates an instance of a server with a port of 80. Following this, it stores the SSID (Wi-Fi access point name) and password in the respective fields. It declares a string variable to contain the text of the HTTP request from the client (the web browser displaying the project's web page) and then sets the GPIO pin it will control.

The sketch then declares variables to take care of the connection time allowed for each client to the server (the ESP32 dev board) ❷. In `void setup()`, it configures the Serial Monitor and then sets GPIO16 as an output and to LOW. The ESP32 attempts to connect to the Wi-Fi network using the credentials entered earlier, with the code looping and displaying a tilde (~) every half second until the network connection is successful. Once the ESP32 has made a Wi-Fi connection, the sketch displays the IP address in the Serial Monitor and the project begins to operate as a web server, thanks to `server.begin()`.

The ESP32 now runs the code in `void loop()` until a reset or removal of power. Each loop begins by checking to see if a client

(a person using a web browser) is trying to connect to the project ❸. If so, the code takes note of the point in time at which the connection starts using the `millis()` function to allow each individual client to connect to the ESP32 for only up to two seconds (as set with `timeoutTime`). This is because other users may want access as well—you don't want one person hogging access to the project. The code also empties the string variable `currentLine`, awaiting text from the client.

If the client has been connected for less than 2,000 milliseconds and has made a request, the ESP32 receives it. The code feeds the `currentTime` variable with the latest value of `millis()`. The string request receives the client request ❹, one character at a time. This request is displayed on the Serial Monitor so that, if you're curious, you can see the information the client sends out (browser type, device operating system, and so on). Once the client has finished sending the request, as denoted by a newline character (`\n`), the ESP32 sends a response in HTML back to the client browser. To do so, it uses `client.println()` functions, which send the text out via the Wi-Fi connection in the same way you may use `Serial.println()` in other projects.

The ESP32 can finally act on the client request and take an action. Client requests are received in the form `/16/on` or `/16/off`, so the string request is searched in turn for an on

command ❺ and acted upon by sending this to the Serial Monitor and then turning GPIO16 on; if an off command is received, the opposite occurs ❻. This is the point where you can modify the actions to suit your own purposes. This project simply calls a `digitalWrite()` function, and you can replace this with your own function to have the ESP32 take an action of your own choosing.

The final section of the code enables ESP32 to host the simple web page that is the user interface shown in Figure 19-10. Using many `client.println()` functions, the code creates HTML that creates the framework for a web page, defines using CSS the buttons to be displayed, positions the buttons, and sends HTML to define the end of the web page. Once the client request is complete, the sketch flushes the request from the variable ❼ and closes the connection, ready for the next user.

Let's take a closer look at the HTML contained in the `client.println()` functions:

```
<!DOCTYPE html><html>
<head><meta name=\"viewport\" content=\"width=de
<link rel=\"icon\" href=\"data:,\">
<style>html {font-family: Helvetica; display: in
 margin: 0px auto; text-align: center;}
```

```
  .button {background-color: #ff0000; border: none
  text-decoration: none; font-size: 20px; margin: 
  </style></head>
  <body><h3>Remote ESP32 Output Control</h3> ❷
  <p><a href=\"/16/on\"><button class=\"button\">O
  <a href=\"/16/off\"><button class=\"button\">Off
  </body></html>
```

This code uses CSS to select the color, size, font size, and rounded edges of the web interface's buttons ❶, with the heading displayed in a larger type ❷. It then adds the two buttons to the web page. Each button has a hyperlink to its respective operation attached: the first link is `/16/on` ❸, and the second is `/16/off` ❹. When the user clicks either of the buttons, the client request will consist of the IP address of the ESP32, followed by */16/on*—for example, *http://192.168.20.28/16/on*.

If you hover your mouse over the button on the web page, the link should appear at the bottom of the screen. For example, Figure 19-10 shows what happens when the mouse hovers over the On button. (The URL in the browser tab at the top is for the last page I visited, the result of clicking the Off button.)



Figure 19-10: Example hyperlink displayed in the browser

When installing a project on a more permanent basis, the end user can also bookmark the full URL for each function. For example, if the IP address returned by the project is 192.168.20.28, the bookmark to turn on would be *http://192.168.20.28/16/on*.

HTML AND CSS RESOURCES

If you're not familiar with HTML or CSS and would like to learn more, you can visit the following sites for more information:

- HTML basics: *https://www.w3schools.com/html/default.asp*
- Creating and using buttons: *https://www.w3schools.com/css/css3_buttons.asp*

- Determining color codes in hexadecimal: *https://www.color-hex.com*

You can also use *https://htmledit.squarefree.com* as a tool to preview your own HTML without having to publish it or upload to the ESP32.

Be warned that anyone who has the URL address for this type of project can operate the controls. For increased security, you might create a secure website or portal that password-protects access to the control buttons.

Project #56: Remote-Controlling Four GPIO Pins

This project builds on the previous project by allowing you to remotely control four GPIO pins, displaying their status on the web page hosted by the ESP32. Users can then quickly review the status of the devices under control, which is ideal for truly remote-controlled projects.

You'll use the same demonstration circuit described in "The GPIO Pins" on page 326 (this is why it includes four LEDs). Once you've set up that circuit, enter and upload the Project #56 sketch to your ESP32 dev board. A moment after that, open the

Serial Monitor in the Arduino IDE to watch as the ESP32 connects to the Wi-Fi network.

Once the ESP32 connects, the IP address of the project should appear in the Serial Monitor. Enter that IP address into a web browser and the interface for this project should appear, as shown in Figure 19-11.

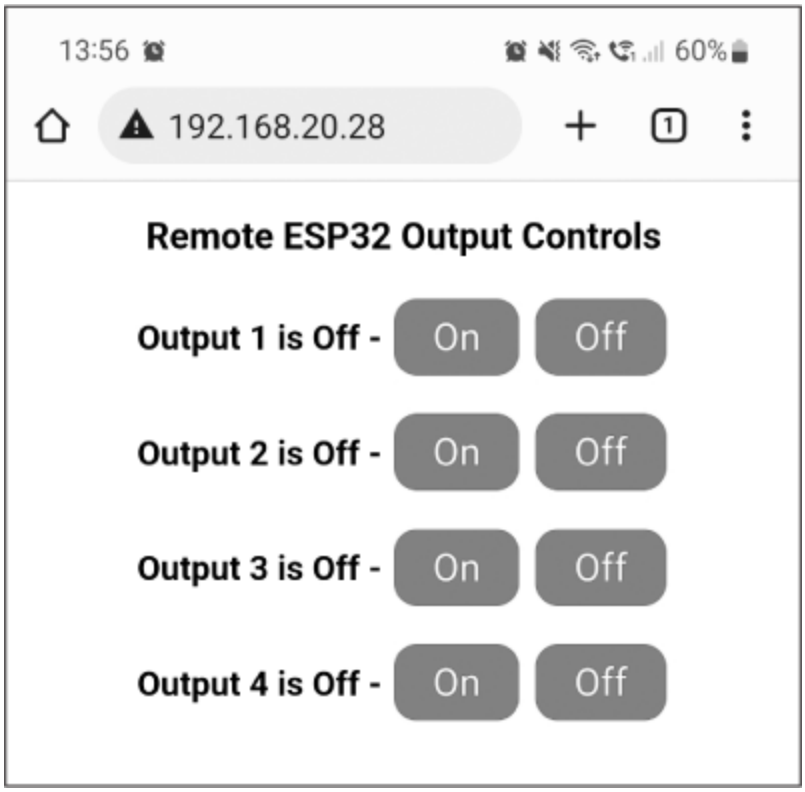


Figure 19-11: The user interface for Project #56

You should be able to control all four LEDs by clicking their respective buttons. After each button click, the web page should

also indicate the state of the linked LED. For example, Figure 19-12 shows that outputs 2 and 4 are on.

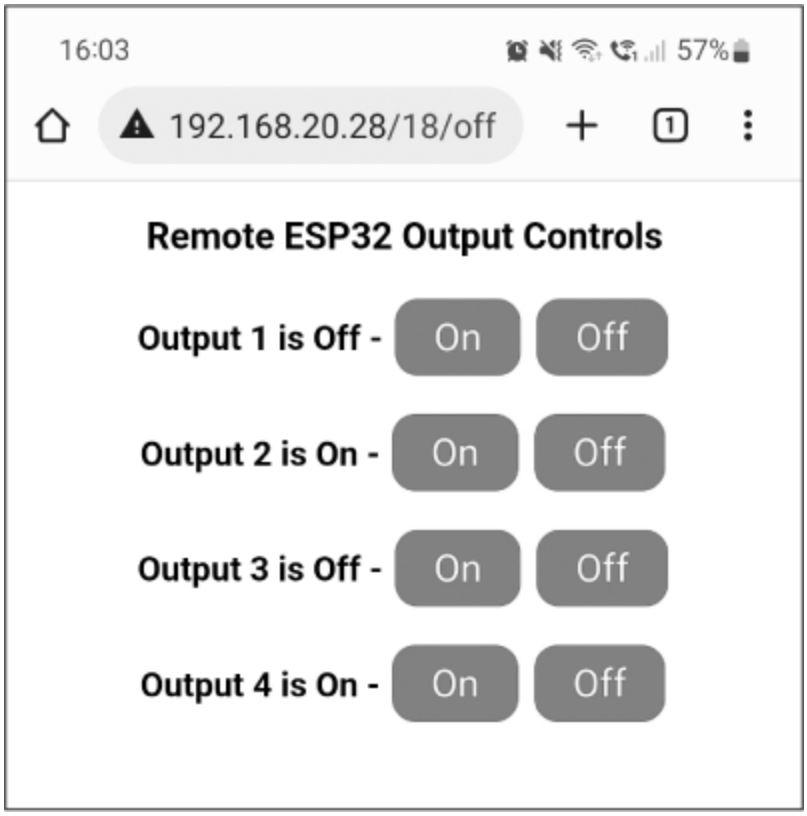


Figure 19-12: Outputs 2 and 4 are on, while 1 and 3 are off.

You can imagine how much more useful this is to have four sets of buttons, enabling control of more relays, outputs, devices controlled by MOSFETs, and so on.

Let's take a closer look to see how this works:

```
// Project #56 - Digital output controls via web
```

```
#include <WiFi.h>
WiFiServer server(80); // Set web server port to

// Store your Wi-Fi network credentials:
char* ssid = "name"; // Insert name of Wi-Fi acces
char* password = "password"; // Insert password

String request; // Stores HTTP request from clie
String outputPin1State = "Off"; ❶
String outputPin2State = "Off";
String outputPin3State = "Off";
String outputPin4State = "Off";

int outputPin1 = 16; // Using GPIO pins 16~19 on
int outputPin2 = 17;
int outputPin3 = 18;
int outputPin4 = 19;

unsigned long currentTime = millis();
unsigned long previousTime = 0;
const long timeoutTime = 2000; // Allowed client

void setup()
{
    Serial.begin(115200);
    // Set up digital pins to control:
    pinMode(outputPin1, OUTPUT); ❸
    pinMode(outputPin2, OUTPUT);
```

```
    pinMode(outputPin3, OUTPUT);
    pinMode(outputPin4, OUTPUT);
    digitalWrite(outputPin1, LOW);
    digitalWrite(outputPin2, LOW);
    digitalWrite(outputPin3, LOW);
    digitalWrite(outputPin4, LOW);
    // Connect to Wi-Fi network:
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    // Show indicator while waiting for connecti
    {
        delay(500);
        Serial.print("~");
    }
    Serial.println(); // Display local IP addres
    Serial.print("Connected, IP address: ");
    Serial.println(WiFi.localIP());
    // Start web server:
    server.begin();
}

void loop()
{
    // Listen for incoming clients from web brow
    WiFiClient client = server.available();
    if (client) // If a request received
```

```
    {
        currentTime = millis();
        previousTime = currentTime;
        Serial.println("New client connected");
        String currentLine = " ";
        while (client.connected() && currentTime-
        {// Stay connected for timeoutTime
            currentTime = millis();
            if (client.available())
            {
                char c = client.read(); // Displ
                Serial.write(c);
                request += c;
                if (c == '\n')
                {
                    // Client request has finish
                    if (currentLine.length() == 
                    {
                        // Send HTTP response ba
                        client.println("HTTP/1.1
                        client.println("Content-
                        client.println("Connecti
                        client.println();
                        // Now do things based o
                        if (request.indexOf("GET
                        {
                            // If request was IP
                            Serial.println("outp
```

```
                outputPin1State = "O
                digitalWrite(outputP
            } else if (request.index
            {
                // If request was IP
                Serial.println("outp
                outputPin1State = "O
                digitalWrite(outputP
            } else if (request.index
            {
            // If request was IP add
                Serial.println("outp
                outputPin2State = "O
                digitalWrite(outputP
            } else if (request.index
            // If request was IP add
                {
                Serial.println("outp
                outputPin2State = "O
                digitalWrite(outputP
                    }
        else if (request.indexOf("GE
        {
            // If request was IP add
            Serial.println("output 3
            outputPin3State = "On";
            digitalWrite(outputPin3,
        } else if (request.indexOf("
```

```
                    // If request was IP address
                    {
                        Serial.println("output 3
                        outputPin3State = "Off";
                        digitalWrite(outputPin3,
                    }
                    else if (request.indexOf("GE
                    {
                        // If request was IP add
                        Serial.println("output 4
                        outputPin4State = "On";
                        digitalWrite(outputPin4,
                    } else if (request.indexOf("
                    // If request was IP address
                    {
                        outputPin4State = "Off";
                        Serial.println("output 4
                        digitalWrite(outputPin4,
                    }

                    // Build web page for displa
                    client.println("<!DOCTYPE ht
                    client.print("<head><meta na
                    client.println("device-width
                    client.println("<link rel=\"
                    // Build buttons using CSS:
                    client.print("<style>html {f
                    client.print("margin: 0px au
```

```
                    client.print(".button {backg
                    client.print("white; padding
                    client.print("text-decoratio
                    client.println("cursor: poin

                    client.println("</style></he
                    // Now HTML for layout:
                    client.println("<body><h3>Re
                    client.println("<p><b>Output
                    client.println(outputPin1Sta
                    client.print(" - </b><a href=
                    client.println("class=\"butt
                    client.print("<a href=\"/16/
                    client.println("class=\"butt
                    client.println("<p><b>Output
                    client.println(outputPin2Sta
                    client.print(" - </b><a href=
                    client.println("class=\"butt
                    client.print("<a href=\"/17/
                    client.println("class=\"butt
                    client.println("<p><b>Output
                    client.println(outputPin3Sta
                    client.print(" - </b><a href=
                    client.println("class=\"butt
                    client.print ("<a href=\"/18
                    client.println("class=\"butt
                    client.println("<p><b>Output
                    client.println(outputPin4Sta
```

```
                    client.print(" - </b><a href=
                    client.println("class=\"butt
                    client.print ("<a href=\"/19
                    client.println(" class=\"but
                    client.println("</body></htm
                    // End the response to clien
                    client.println();
                    break;
                } else
                {// If you got a newline, then c
                    currentLine = " ";
                }
              } else if (c != '\r')
                {
                    currentLine += c;
                }
            }
        }
        request = " ";
        // Close connection to client:
        client.stop();
        Serial.println("Disconnected.");
    }
}
```

The framework for this project is the same as that of the previous project, except that it controls four GPIOs and eight

buttons while keeping track of the GPIO status. The sketch uses four string variables to store the status of the four GPIO pins in text form, On or Off, declaring them with a default value of Off ❶. To save confusion, the sketch also declares variables to give each GPIO pin a name instead of a number in the code ❷. It then sets the pins as outputs and to LOW ❸.

As there are now eight buttons and eight matching actions (on and off for each of the four GPIOs), there are eight client requests to look out for ❹. After each action has taken place, the status of the matching GPIO is updated in its matching state variable ❺. This occurs for all eight client request types.

Once again, the code uses many `client.println()` functions to create the HTML for the user interface. Functions like the one at ❻ in the project code generate the HTML required to display the status of each GPIO pin. The sketch also attaches a hyperlink to each button (using `<a href=`) such as `/16/on` to turn GPIO 16 on, and so on.

Here's the full HTML code contained in the sketch:

```
<!DOCTYPE html><html>
<head><meta name=\"viewport\" content=\"width=de
<link rel=\"icon\" href=\"data:,\">
```

```
<style>html {font-family: Helvetica; display: in
margin: 0px auto; text-align: center;}
".button {background-color: #ff0000; border: non
"text-decoration: none; font-size: 20px; margin:
</style></head>
<body><h3>Remote ESP32 Output Controls</h3>
<p><b>Output 1 is "
// Insert outputPin1State:
" - </b><a href=\"/16/on\"><button class=\"butto
<a href=\"/16/off\"><button class=\"button\">Off
<p><b>Output 2 is "
// Insert outputPin2State:
" - </b><a href=\"/17/on\"><button class=\"butto
<a href=\"/17/off\"><button class=\"button\">Off
<p><b>Output 3 is "
// Insert outputPin3State:
" - </b><a href=\"/18/on\"><button class=\"butto
"<a href=\"/18/off\"><button class=\"button\">Of
<p><b>Output 4 is ");
// Insert outputPin4State:
" - </b><a href=\"/19/on\"><button class=\"butto
<a href=\"/19/off\"><button class=\"button\">Off
</body></html>
```

I encourage you to experiment with the buttons—changing their size, shape, the URLs attached to them, and so on—to help you master the process of building an interface.

Keep this project's circuit assembled. You'll use it in the next section to learn about using pulse-width modulation on the ESP32 dev board.

Pulse-Width Modulation

Just like with the usual Arduino and compatible boards, the ESP32 dev board offers pulse-width modulation (PWM) with all the GPIO pins on the ESP32 dev board. PWM allows you to control the perceived brightness of an LED, instead of simply switching it on and off. The LED's brightness is determined by the duty cycle, or the length of time the GPIO pin is on (so the LED is lit) versus the length of time it is off (the LED is unlit).

Duty cycle is expressed as the percentage of time the pin is on in each *cycle*, the fixed period of time in which the pin can be on or off. The greater the duty cycle, the greater the perceived brightness of the LED connected to the pin. Furthermore, the higher the frequency of the PWM signal—that is, the faster the signal is turned on and off—the smoother the visual effect. If you're controlling a motor, a higher PWM frequency makes the rotational speed a closer approximation of the actual speed required.

Figure 19-13 shows four possible PWM duty cycles. The filled-in gray areas represent the amount of time that the LED is on; as you can see, this increases with the duty cycle.

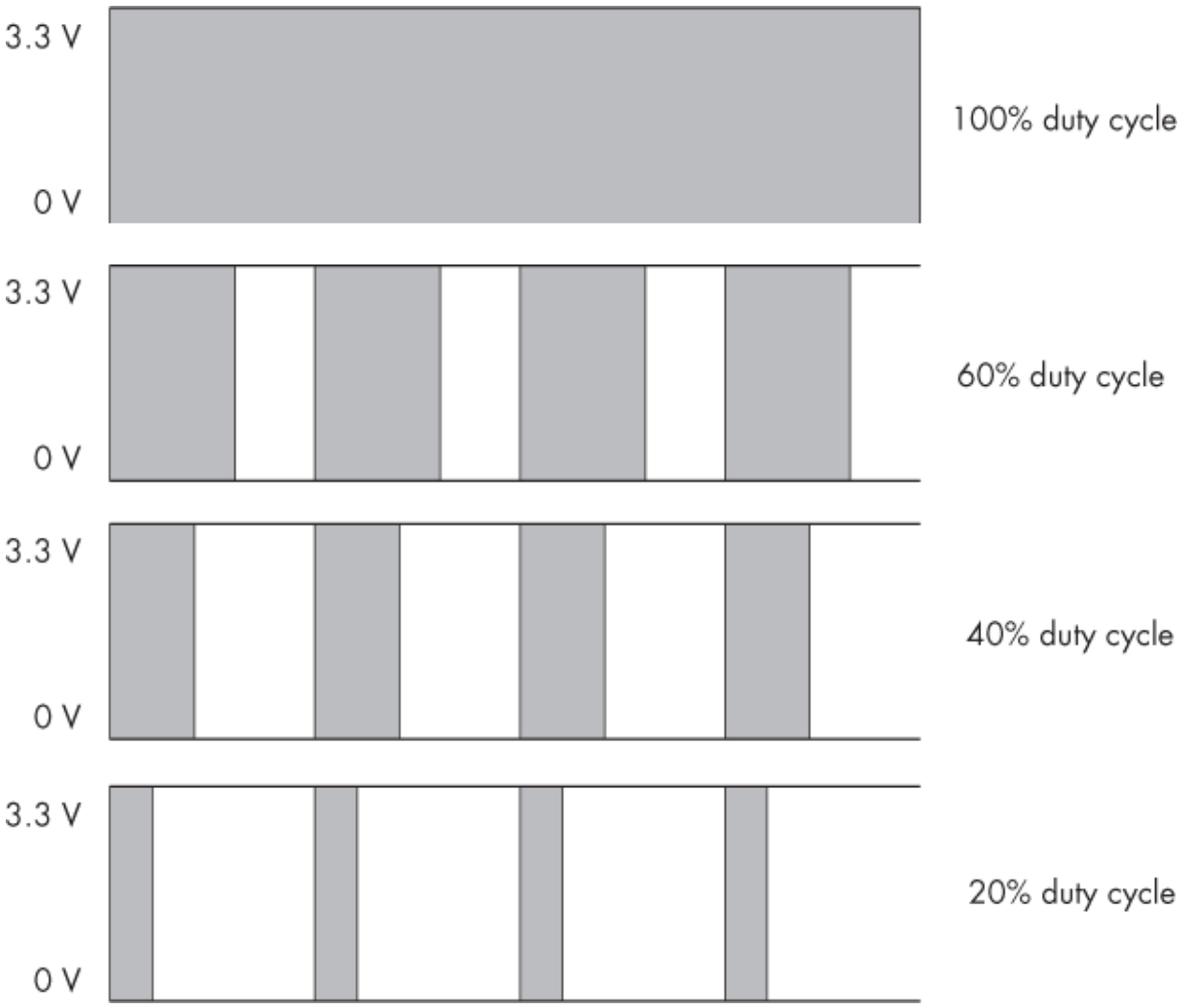


Figure 19-13: Various PWM duty cycles

The ESP32 module has 16 PWM channels (numbered 0 to 15). You can direct each channel to a GPIO pin. To do so, you must

add two lines to `void setup()` for each pin you want to use. The first line is

```
ledcSetup(channel, frequency, resolution);
```

where *`channel`* is the PWM channel to use (0 through 15), *`frequency`* is the PWM frequency, and *`resolution`* is the detail for the duty cycle. For *`frequency`*, I suggest 5,000 for 5 kHz. For *`resolution`*, use 8 for 8-bit resolution, giving you a value of between 0 and 255 for the duty cycle (the same way you'd use `analogWrite()` when using an Arduino or compatible board).

The second line required is

```
ledcAttachPin(GPIO, channel);
```

where *`GPIO`* is the number of the GPIO pin to use, and *`channel`* is the PWM channel to use, as defined in the previous line of code.

Finally, to activate PWM on the GPIO, use

```
ledcWrite(channel, dutyCycle);
```

where *channel* is the PWM channel to control, and *dutyCycle* is, of course, the duty cycle.

To stop using PWM on a particular GPIO pin and instead use it as a digital input or output, use

```
ledcDetachPin(GPIO);
```

where *GPIO* is the number of the GPIO pin (not the PWM channel).

You can test PWM and experiment with these parameters with the following sketch, which uses the same four-LED circuit you've used throughout this chapter. A moment after you enter and upload Listing 19-2, all four LEDs should use PWM to exhibit a "breathing" effect by using increasing and then decreasing values for the duty cycle.

```
void setup()
{
  ❶ ledcSetup(0, 5000, 8); // LED channel, freque
    ledcSetup(1, 5000, 8);
    ledcSetup(2, 5000, 8);
    ledcSetup(3, 5000, 8);

  ❷ ledcAttachPin(16, 0); // GPIO, LED channel
```

```
    ledcAttachPin(17, 1);
    ledcAttachPin(18, 2);
    ledcAttachPin(19, 3);
}

void loop()
{
    int dutyCycle;
    for (dutyCycle = 0; dutyCycle <= 255; dutyCy
    {
      ❸ ledcWrite(0, dutyCycle);
        ledcWrite(1, dutyCycle);
        ledcWrite(2, dutyCycle);
        ledcWrite(3, dutyCycle);
        delay(10);
    }
    for (dutyCycle = 255; dutyCycle > -0; dutyCy
    {
      ❹ ledcWrite(0, dutyCycle);
        ledcWrite(1, dutyCycle);
        ledcWrite(2, dutyCycle);
        ledcWrite(3, dutyCycle);
        delay(10);
    }
}
```

Listing 19-2: Testing PWM outputs

This sketch demonstrates the three relevant lines for using PWM, setting up the required PWM channels ❶, and then attaching those channels to GPIO pins ❷. This allows the duty cycles to be set for each of the pins at an increasing ❸ and decreasing ❹ level.

You'll put this technique to use in the next project.

Project #57: Building a Hosted Web Page for a User Interface

This project synthesizes everything you've learned in this chapter: using an ESP32 dev board to control a GPIO via Wi-Fi, using PWM for various output levels, and building a hosted web page for a user interface. You can use this as a framework to build projects that require PWM control, such as LED lighting effects, controlling DC motors via MOSFETs, or experimenting with sound effects from a distance.

This project uses the same circuit as in previous projects, with only one LED connected to GPIO16 being controlled. Once the circuit is ready and the ESP32 dev board has been connected to the PC, upload the Project #57 sketch. A moment after that, determine the project's IP address as usual with the Serial Monitor and then enter that address into your web browser.

The browser should display the project interface, as shown in Figure 19-14.

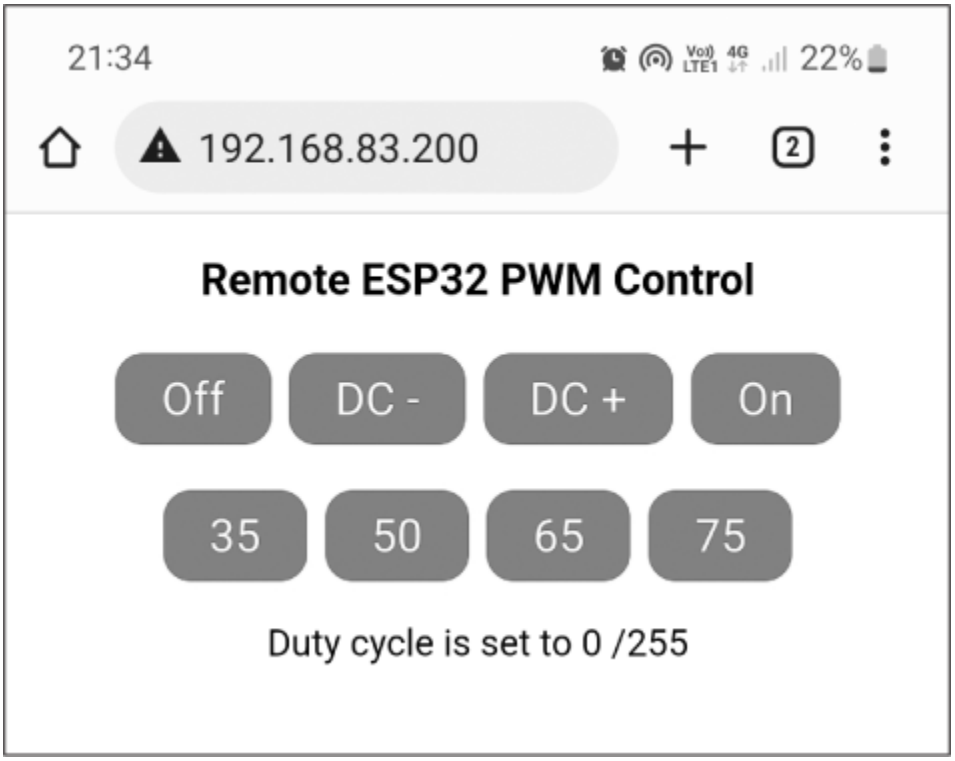


Figure 19-14: The user interface for Project #57

You should be able to turn the LED on and off with the appropriate buttons as usual. This time, though, you can also decrease or increase the duty cycle one step at a time with the DC– and DC+ buttons. You can click the second row of four additional buttons to give the duty cycle a preset value, and the line of text beneath them shows the current duty cycle value. For example, Figure 19-14 shows the LED as off (as the duty

cycle is 0), while in Figure 19-15, the LED is on with a duty cycle of 72/255.

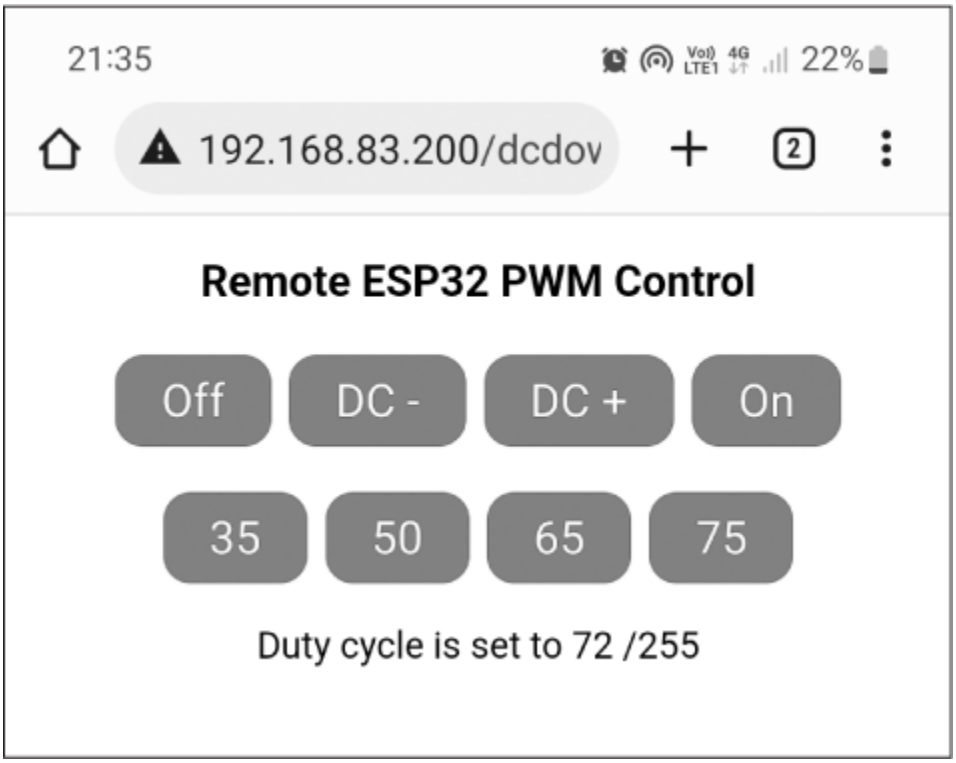


Figure 19-15: The user interface with the duty cycle set to 72/255

Let's see how this works:

```
// Project #57 - ESP32 Remote Control PWM Output

#include <WiFi.h>
WiFiServer server(80); // Set webserver port to

// Store your Wi-Fi network credentials:
char* ssid = "name"; // Insert name of Wi-Fi acces
```

```
char* password = "password"; // Insert password

int outputPin1 = 16;
int dutyCycle = 0; ❶
String request; // Stores HTTP request from clie

unsigned long currentTime = millis();
unsigned long previousTime = 0;
const long timeoutTime = 2000; // Allowed client

void pinOff()
{
    ledcDetachPin(16); // Disconnect from PWM
    digitalWrite(16, LOW); // Pin off
}

void pinOn()
{
    ledcDetachPin(16); // Disconnect from PWM
    digitalWrite(16, HIGH); // Usual turn pin on
}

void pinPWM(int dc)
{
    // Set up digital pin for PWM:
    ledcSetup(0, 5000, 8); // LED channel, frequ
    ledcAttachPin(16, 0);  // Attach PWM – GPIO,
    ledcWrite(0, dc);      // Set duty cycle
```

```
}

void setup()
{
    Serial.begin(115200);
    pinMode(16, OUTPUT);
    pinOff(); ❷

    // Connect to Wi-Fi network:
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    // Show indicator while waiting for connecti
    {
        delay(500);
        Serial.print("~");
    }
    Serial.println(); // Display local IP addres
    Serial.print("Connected, IP address: ");
    Serial.println(WiFi.localIP());
    // Start web server:
    server.begin();
}

void loop()
{
    // Listen for incoming clients from web brow
```

```
    WiFiClient client = server.available();
    if (client) // If a request received
    {
        currentTime = millis();
        previousTime = currentTime;
        Serial.println("New client connected");
        String currentLine = " ";
        while (client.connected() && currentTime-
        {// Stay connected for timeoutTime
            currentTime = millis();
            if (client.available())
            {
                char c = client.read(); // Displ
                Serial.write(c);
                request += c;
                if (c == '\n') // Client request
                {
                    if (currentLine.length() == 
                    {
                        // Send HTTP response ba
                        client.println("HTTP/1.1
                        client.println("Content-
                        client.println("Connecti
                        client.println();

                        // Now do things based o
                        if (request.indexOf("GET
                        {
```

```
            // If request was IP
            pinOff(); ❸
            dutyCycle = 0;
        } else if (request.index(
        // If request was reduce
        {
            if (dutyCycle >= 2) 
            {
                pinPWM(dutyCycle
                dutyCycle--;
            } else if (dutyCycle
            {
                pinOff();
                dutyCycle = 0;
            }

        } else if (request.index(
        {
            // If request was in
            if (dutyCycle > 253)
            {
                pinOn();
                dutyCycle = 255;
            }
            else if (dutyCycle <=
            {
                pinPWM(dutyCycle
                dutyCycle++;
```

```
    }

} else if (request.index
// If request was IP add
{
    pinOn();
    dutyCycle = 255;
}
else if (request.indexOf
{
    // If request was IP
    pinPWM(35);
    dutyCycle = 35;
} else if (request.index
// If request was IP add
{
    pinPWM(50);
    dutyCycle = 50;
}
else if (request.indexOf
{
    // If request was IP
    pinPWM(65);
    dutyCycle = 65;
} else if (request.index
// If request was IP add
{
    pinPWM(75);
```

```
    dutyCycle = 75;
}

// Build web page for di
client.println("<!DOCTYP
client.print("<head><met
client.println(", initia
client.println("<link re
// Build buttons using C
client.print("<style>htm
client.println("margin: 
client.print(".button {b
client.println("white; p
client.print("text-decor
client.println("pointer;
client.println("</style>
// Now HTML for layout:
client.println("<body><h
client.println("<p><a hr
client.print("<a href=\"
client.println("DC -</bu
client.println(" <a href
client.println(" <a href
client.println("<p><a hr
client.println(" <a href
client.println(" <a href
client.println(" <a href
client.println(" Duty cy
```

```
                        client.println(dutyCycle
                        client.println("/255</p>
                        client.println("</body></
                        // End the response to c
                        client.println();
                        break;
                    } else {// If you got a newl
                    currentLine = " ";
                    }
                } else if (c != '\r')
                {
                    currentLine += c;
                }
            }
        }
        request = " ";
        // Close connection to client:
        client.stop();
        Serial.println("Disconnected.");
    }
}
```

The operation of this project is similar to that of the previous projects, but with more buttons and a different type of status display.

When the user changes the duty cycle of the PWM output, that value is stored in an integer variable ❶. The custom `pinOff()` function is called when the user clicks the Off button. Since you can't set PWM to 0 duty cycle to turn off the GPIO output, this function detaches the PWM channel from the GPIO pin and then uses `digitalWrite()` to turn off the GPIO pin. The `pinOn()` function does the opposite; since you can't set the PWM to full-on, this function detaches the PWM channel and then turns the GPIO pin on using `digitalWrite()`.

The sketch uses the custom `PWM(int dc)` function to activate PWM, attach a PWM channel to the GPIO pin, and set the required duty cycle, passed in via the parameter `dc`. When you first start the project or reset it, the sketch turns the GPIO pin off ❷.

Next, the sketch reviews client requests. If the user presses the Off button to turn off the GPIO pin off, the code calls the `pinOff()` function ❸ and sets the `dutyCycle` variable to 0 for display purposes on the interface. If the user presses the DC button ❹, the duty cycle decreases by 1. If this would cause the duty cycle to fall below 1, the GPIO is switched off instead using `pinOff()`. Likewise, if the user presses the DC+ button ❺, the duty cycle increases by 1. If this would push the cycle beyond

254, the sketch instead turns on GPIO using `pinOn()`. Pressing the On button ❻ calls the `pinOn()` function and sets the `dutyCycle` variable to 255 for display purposes on the interface.

The four final buttons ❼ set the duty cycle to preset levels. They're just examples to give you an idea of what is possible. Each request simply sets the PWM value via `pinPWM()` and then updates the `dutyCycle` variable accordingly.

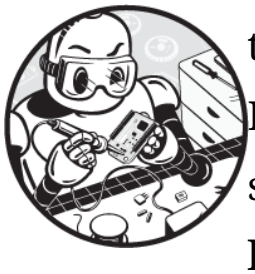
Moving On

This chapter demonstrated how to remotely control your ESP32 dev board over the internet via web pages designed to your specifications. Rather than controlling operations, you can also simply display data generated via the ESP32 on a web page for remote viewing, such as data from a sensor.

You'll continue to use your ESP32 in the following chapters. In the next, you'll learn to control it by interacting with social media.

20

REMOTE CONTROL VIA TELEGRAM



Controlling your Arduino from a distance greater than your local area network often requires paid IoT services, such as Blynk or Microsoft Azure, or some complex coding and time. For a simpler and less expensive way to remotely control your Arduino and retrieve data, you can use the Telegram instant-messaging service and an ESP32-based Arduino-compatible board.

Telegram is a free, globally accessible cross-platform encrypted messaging service that allows users to chat with each other one on one or in groups on a relatively secret basis. Thanks to Telegram's automation feature, you can get an Arduino to

control a Telegram user account to send and receive data over the service. This allows you to build remote-controlled devices and remotely request data from an internet-connected Arduino using Telegram.

In this chapter, I'll introduce you to the Telegram app and web interface. You'll interface Arduino sketches with the Telegram library. You'll learn to:

- Create a simple remote to control digital output pins
- Build a project that remotely retrieves data generated by the BMP180 sensor board
- Configure an automated data transmitter that you can monitor from the internet

Configuring Your Telegram Account

You can communicate over Telegram using either an app on your device or a web browser on almost any device connected to the internet. However, you'll need to create a Telegram account using a device such as a smartphone or tablet before you can use the service on your computer. If you don't already have an account, visit the website *https://telegram.org*, download and install the app for your device, and set up a user account.

To communicate with your Arduino-compatible ESP32 board over Telegram, you'll need to create a Telegram "bot," your own automated Telegram user that receives and sends messages to and from your ESP32. To do so, open Telegram, log into your account on your device, and search for the user *BotFather*, as shown in Figure 20-1 (Android).

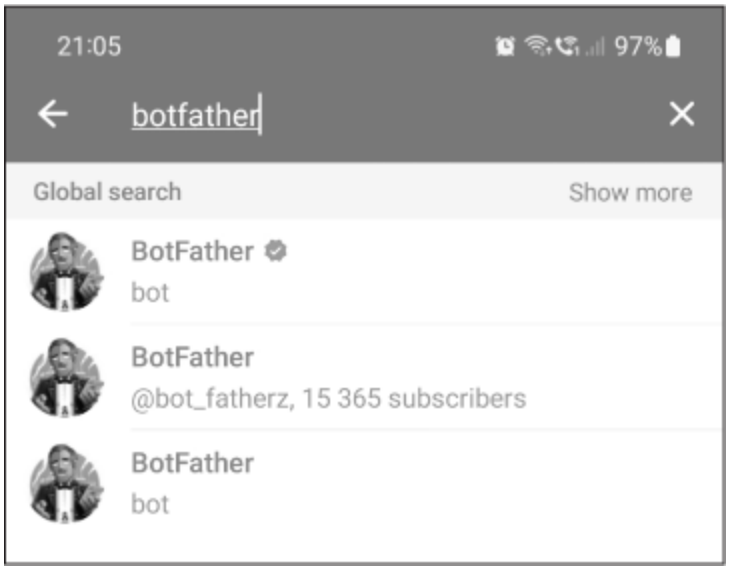


Figure 20-1: Searching for the Telegram user BotFather *on Android*

Select the BotFather account with the blue tick, and you should be presented with the start screen.

Tap **START** to proceed. Telegram should then show you a list of options. Click, touch, or just send /**newbot** in the messenger to be prompted for a bot account name. Once you enter your preferred bot name, you'll be prompted for a username for

your bot. Enter your preferred username. You'll then be presented with your *token*, a unique identifier for your bot that's required for your Arduino sketches, as shown in Figure 20-2.

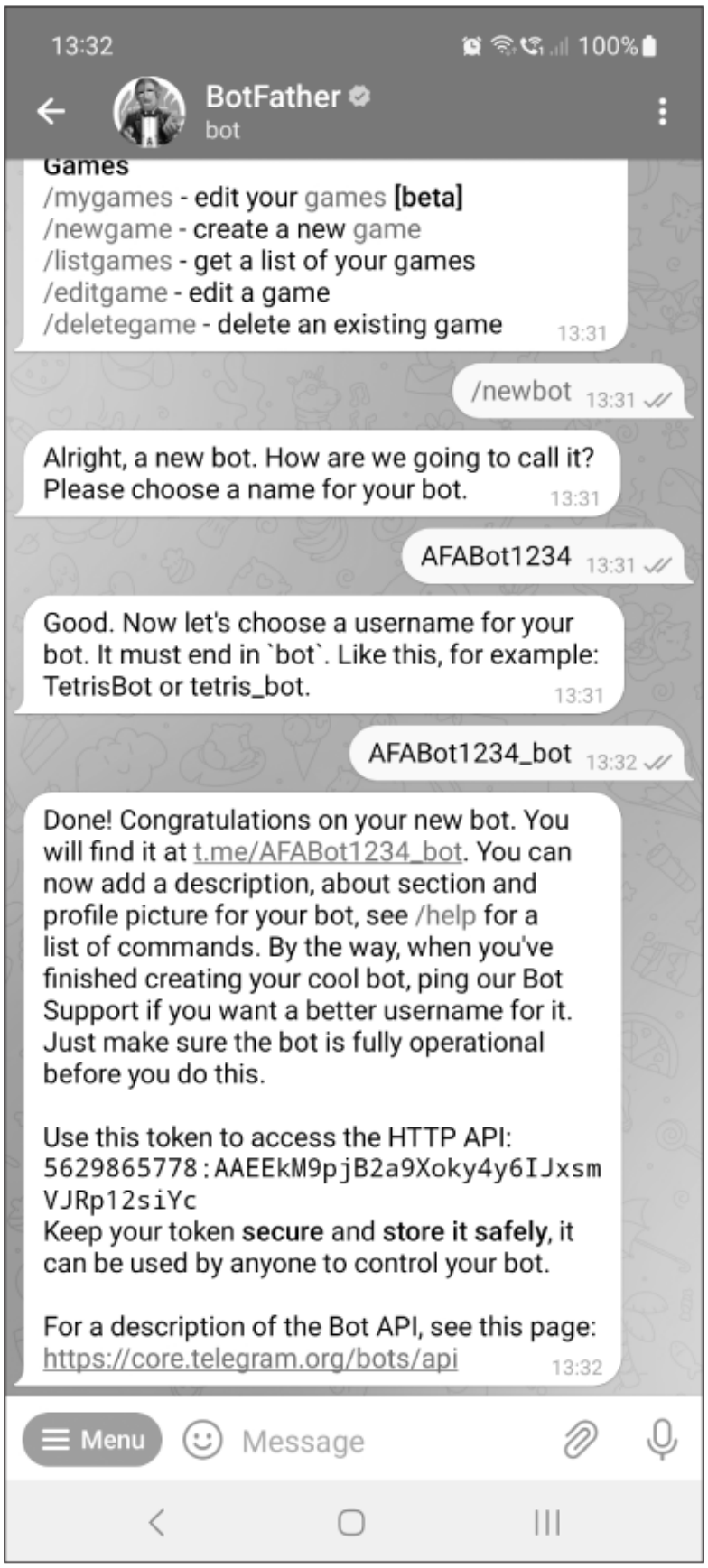

13:32
100%
BotFather
bot
Games
/mygames - edit your games [beta]
/newgame - create a new game
/listgames - get a list of your games
/editgame - edit a game
/deletegame - delete an existing game
13:31
/newbot 13:31
Alright, a new bot. How are we going to call it? Please choose a name for your bot. 13:31
AFABot1234 13:31
Good. Now let's choose a username for your bot. It must end in `bot`. Like this, for example: TetrisBot or tetris_bot. 13:31
AFABot1234_bot 13:32
Done! Congratulations on your new bot. You will find it at t.me/AFABot1234_bot. You can now add a description, about section and profile picture for your bot, see /help for a list of commands. By the way, when you've finished creating your cool bot, ping our Bot Support if you want a better username for it. Just make sure the bot is fully operational before you do this.
Use this token to access the HTTP API:
5629865778:AAEEkM9pjB2a9Xoky4y6IJxsmVJRp12siYc
Keep your token secure and store it safely, it can be used by anyone to control your bot.
For a description of the Bot API, see this page:
https://core.telegram.org/bots/api
13:32
Menu
Message

Figure 20-2: Bot account creation in Telegram

Write the token down for later reference. You might copy and paste it into a file if you're using Telegram on a computer or into a notes app on your mobile device.

Next, you need to determine your Telegram chat ID, a unique number you'll use to authenticate your Arduino's messages to Telegram. Search for the Telegram user *IDBot* and select the result with the image of a fingerprint for the avatar, as shown in Figure 20-3.

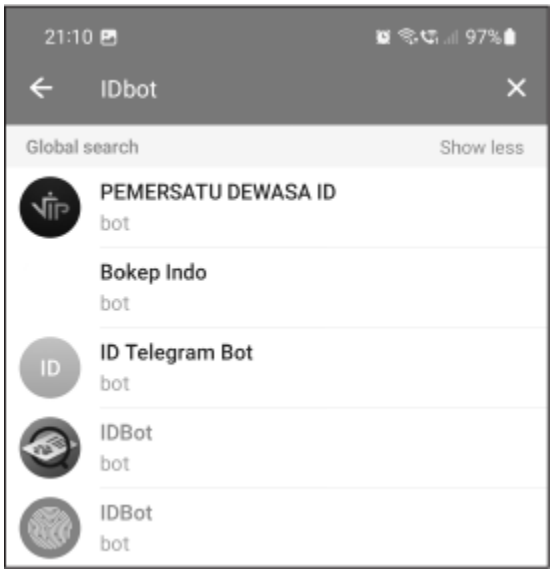


Figure 20-3: Searching for the IDBot *Telegram account*

Once you have opened its account, send IDBot the message /**start** to initialize communication and then /**getid**, as shown in

Figure 20-4. When IDBot replies with your chat ID, write that number down as well, as you'll need it along with the token.

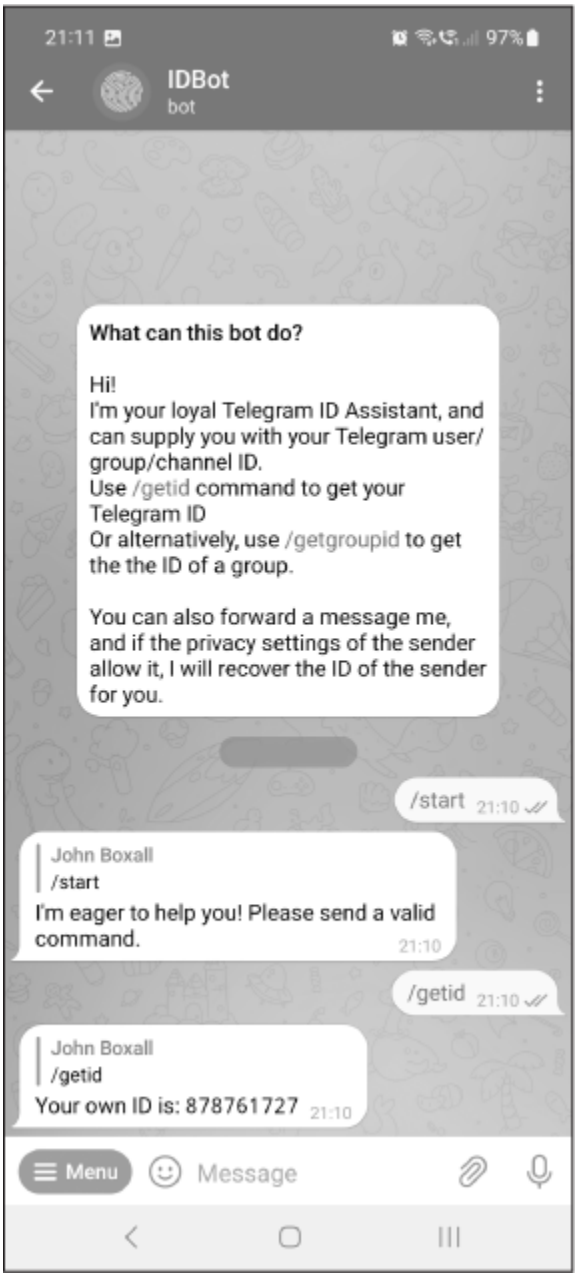


Figure 20-4: Searching for the account's chat ID

Now that you have your Telegram account and bot set up, it's time to configure your Arduino IDE.

Configuring the Arduino IDE

If you haven't already done so, follow the instructions in "Configuring the Arduino IDE for the ESP32" and "Testing the ESP32" in Chapter 19. Once you've set up your ESP32, you need to install two libraries. Search for **UniversalTelegramBot** in the Library Manager, and then click **Install**.

Once UniversalTelegramBot is installed, search for **ArduinoJson** in the Library Manager, and then click **Install** at the bottom of the library description (unless the library was already installed along with UniversalTelegramBot).

You're now ready to create your first project remote-controlled by Telegram.

Project #58: Remote-Controlling Four LEDs

This project demonstrates a fast and simple method of controlling four LEDs via four digital output pins on your Arduino-compatible board via Telegram. With your existing knowledge of electronics and Arduino, you should be able to

easily extrapolate from this basic framework how to control other devices that can be triggered with a digital output pin.

You'll need the following hardware:

- An ESP32 dev board and matching USB cable
- A solderless breadboard
- Assorted jumper wires
- Four LEDs
- One 560 Ω, 0.25 W, 1 percent resistor

Assemble the circuit as shown in Figure 20-5.

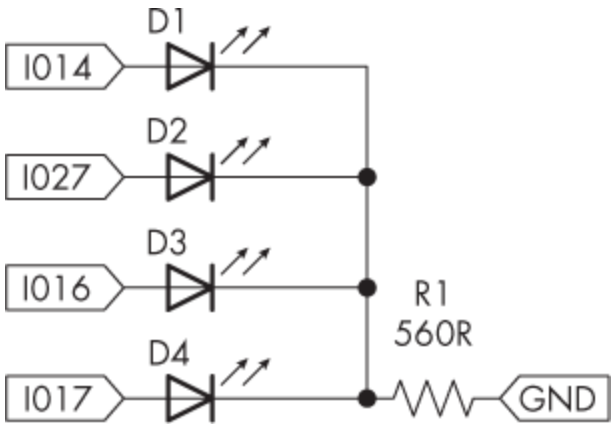


Figure 20-5: The schematic for Project #58

Next, enter and upload the Project #58 sketch, adding your Wi-Fi network credentials in the first few lines as appropriate, as originally explained in Project #55 in Chapter 19. Once the sketch has uploaded, the ESP32 should attempt to connect to

your Wi-Fi network and communicate with the Telegram bot you created earlier. After a few moments, open Telegram on your device or web browser and search for your bot.

Select your bot. You should be presented with the bot's chat history page. You can control your ESP32 board from this chat history. When a user sends the /**start** command, any Telegram bot should send it back to the ESP32, which is programmed to reply to that command with a basic set of instructions. Always include such instructions in your own projects so the user isn't left at a loss on how to control the system.

After Telegram displays the start message, you can send the commands included in the instruction set and watch the LEDs respond to your commands. Figure 20-6 shows an example of starting the bot and controlling the LEDs.

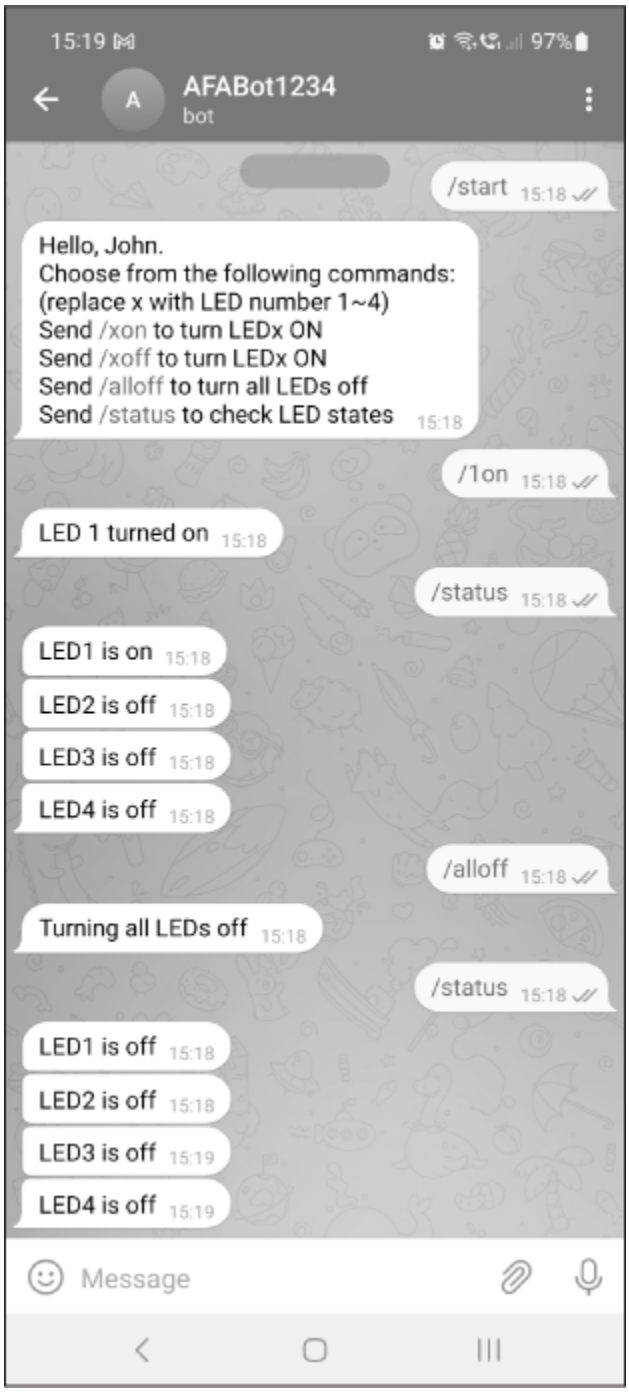


Figure 20-6: Sending various commands to the ESP32 board via Telegram

You should receive a response to all the commands being sent; again, it's good practice to write code that replies to the user to

make sure actions have taken place. The sketch also acknowledges responses in the Arduino IDE's Serial Monitor, as shown in Figure 20-7.



Figure 20-7: Example Serial Monitor output for Project #58

Let's see how this works:

```
// Project #58 - ESP32 GPIO remote control over T

#include <WiFi.h>
#include <WiFiClientSecure.h>
#include <UniversalTelegramBot.h>
#include <ArduinoJson.h>

// Enter your Wi-Fi network SSID and password:
const char* ssid = "wifiname";
const char* password = "password";

// Enter your Telegram bot token and chat ID:
```

```
#define botToken "token"
#define chatID "chatID"

❶ #define LED1 17
#define LED2 16
#define LED3 27
#define LED4 14

WiFiClientSecure client;
UniversalTelegramBot bot(botToken, client);

❷ int bot_delay = 1000;
unsigned long lastRun=0;

void processMessages(int numNewMessages)
{
    Serial.println("Handling New Message");
    Serial.println(String(numNewMessages));

    for (int i = 0; i < numNewMessages; i++)
    {
        // Chat ID of the requester:
     ❸ String chat_id = String(bot.messages[i].c
        if (chat_id != chatID)
        {
            bot.sendMessage(chat_id, "Unauthoriz
            continue;
        }
```

```
    // Print the received message:
    String user_text = bot.messages[i].text;
    Serial.println(user_text);

❹ String your_name = bot.messages[i].from_n

❺ if (user_text == "/start")
    {
        String startMessage = "Hello, " + you
        startMessage += "Choose from the foll
        startMessage += "(replace x with LED
        startMessage += "Send /xon to turn LE
        startMessage += "Send /xoff to turn L
        startMessage += "Send /alloff to turn
        startMessage += "Send /status to chec
      ❻ bot.sendMessage(chat_id, startMessage
    }

❼ if (user_text == "/1on")
    {
        bot.sendMessage(chat_id, "LED 1 turne
        digitalWrite(LED1, HIGH);
    }

    if (user_text == "/1off")
    {
        bot.sendMessage(chat_id, "LED 1 turne
```

```
            digitalWrite(LED1, LOW);
        }

        if (user_text == "/2on")
        {
            bot.sendMessage(chat_id, "LED 2 turn
            digitalWrite(LED2, HIGH);
        }

        if (user_text == "/2off")
        {
            bot.sendMessage(chat_id, "LED 2 turn
            digitalWrite(LED2, LOW);
        }

        if (user_text == "/3on")
        {
            bot.sendMessage(chat_id, "LED 3 turn
            digitalWrite(LED3, HIGH);
        }

        if (user_text == "/3off")
        {
            bot.sendMessage(chat_id, "LED 3 turn
            digitalWrite(LED3, LOW);
        }

        if (user_text == "/4on")
```

```
        {
            bot.sendMessage(chat_id, "LED 4 turn
            digitalWrite(LED4, HIGH);
        }

        if (user_text == "/4off")
        {
            bot.sendMessage(chat_id, "LED 4 turn
            digitalWrite(LED4, LOW);
        }

        if (user_text == "/alloff")
        {
            bot.sendMessage(chat_id, "Turning al
            digitalWrite(LED1, LOW);
            digitalWrite(LED2, LOW);
            digitalWrite(LED3, LOW);
            digitalWrite(LED4, LOW);
        }

        if (user_text == "/status")
        {
            if (digitalRead(LED1))
            {
                bot.sendMessage(chat_id, "LED1 i
            } else
            {
                bot.sendMessage(chat_id, "LED1 i
```

```
            }
            if (digitalRead(LED2))
            {
                bot.sendMessage(chat_id, "LED2 i
            } else
            {
                bot.sendMessage(chat_id, "LED2 i
            }
            if (digitalRead(LED3))
            {
                bot.sendMessage(chat_id, "LED3 i
            } else
            {
                bot.sendMessage(chat_id, "LED3 i
            }
            if (digitalRead(LED4))
            {
                bot.sendMessage(chat_id, "LED4 i
            } else
            {
                bot.sendMessage(chat_id, "LED4 i
            }
        }
    }
}

void setup()
{
```

```
    Serial.begin(115200);

    pinMode(LED1, OUTPUT);
    pinMode(LED2, OUTPUT);
    pinMode(LED3, OUTPUT);
    pinMode(LED4, OUTPUT);

    digitalWrite(LED1, LOW);
    digitalWrite(LED2, LOW);
    digitalWrite(LED3, LOW);
    digitalWrite(LED4, LOW);

    // Connect to Wi-Fi
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);

    // Add root certificate for api.telegram.org
    client.setCACert(TELEGRAM_CERTIFICATE_ROOT);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.println("Connecting to Wi-Fi…");
    }
    Serial.println(WiFi.localIP()); // Display I
}

void loop()
{
```

```
      if (millis() > lastRun + bot_delay)
      {
          int numNewMessages = bot.getUpdates(bot.
          while (numNewMessages)
          {
              Serial.println("Received message");
              processMessages(numNewMessages);
              numNewMessages = bot.getUpdates(bot.
          }
          lastRun = millis();
      }
  }
```

The sketch first includes the required libraries, followed by fields to enter your Wi-Fi network's ID and password and then fields to enter your Telegram token and chat ID. It defines the pin numbers for the LEDs so you can easily refer to them in the rest of the sketch ❶ and then starts instances of the secure Wi-Fi connection client and the Telegram bot interface. Next, the sketch sets the length of the period it will wait before checking if new Telegram messages are available to 1,000 milliseconds ❷.

The sketch uses the custom function `processMessages()` to retrieve the messages received by the Telegram bot and then takes actions based on the contents of the message. The entire

process is repeated for the amount passed through the parameter `numNewMessages`. The sketch first performs a security check by comparing your Telegram alphanumeric chat ID against the chat ID being sent with the message from Telegram ❸. This prevents hackers from attempting to control your Telegram bot. If the two don't match, the sketch jumps out of the loop with a `continue` function and ignores the message(s) received. If they do match, the sketch proceeds to send the message received to the Serial Monitor for debugging purposes and then retrieves the Telegram username attached to your Telegram account ❹. This is used to add a friendly level of personalization.

To decide how to act on the message received, the sketch compares the received message against various `if` functions, the first of which is `/start` ❺. When activated, this part of the sketch combines lines of text, including the Telegram username, into one string variable, which is then sent to be displayed in the Telegram chat using `bot.sendMessage()` ❻. The sketch is set up to accept various other chat messages from a user, all beginning with a forward slash (`/`). For example, the message to turn on LED number 1 (`/1on`) is detected at ❼. When a message is received from the user, a `bot.sendMessage()` function sends an acknowledgment back to the Telegram chat and then turns on

the LED. The sketch receives and acts upon the rest of the possible messages in a similar manner.

In `void setup()`, the sketch configures the digital pins for the LEDs and sets them to `LOW` by default, initializes the Wi-Fi library, and sets up a secure connection to Telegram. Finally, it connects the ESP32 board to your Wi-Fi and sends the IP address to the Serial Monitor.

In `void loop()`, the sketch checks Telegram for any new chat messages every second, using the comparison in the `if` function. Every time a message is received, the value of `lastRun` is updated to the value returned by `millis()`. If the value of `millis()` is greater than `lastRun` plus `bot_delay` (the amount of time to wait between checking for messages), it's time to check for messages again. The number of new messages is then retrieved from Telegram and processed in turn with the `processMessages()` function described earlier.

This sketch provides a simple framework for acting upon messages received from Telegram, along with a simple remote control for digital outputs. The next project builds on this framework, teaching you to retrieve data from the remote ESP32 board using Telegram.

Project #59: Retrieving Remote Data

This project enables you to retrieve, on demand, the data generated by the BMP180 sensor board first used in Chapter 10, including temperature, air pressure, or height above sea level. You can also use this project as a general framework for remotely monitoring parameters over Telegram.

You'll need the following hardware:

- An ESP32 dev board and matching USB cable
- A solderless breadboard
- Assorted jumper wires
- A BMP180 sensor board

Assemble the circuit as shown in Figure 20-8. The BMP180 can work on 3.3 V as well as 5 V, so a level converter board isn't necessary in this case.

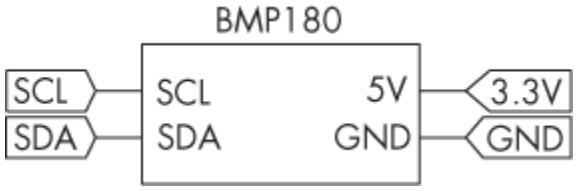


Figure 20-8: The schematic for Project #59

Once you have built the circuit and uploaded the sketch, open Telegram and enter the chat history for your Telegram bot, as

with the previous project. Enter /**start** in Telegram. You should receive a message listing the commands you can use to retrieve sensor data, as shown in Figure 20-9. For variety, I've used the Telegram Windows application, rather than the Android app, to take screenshots for this project.

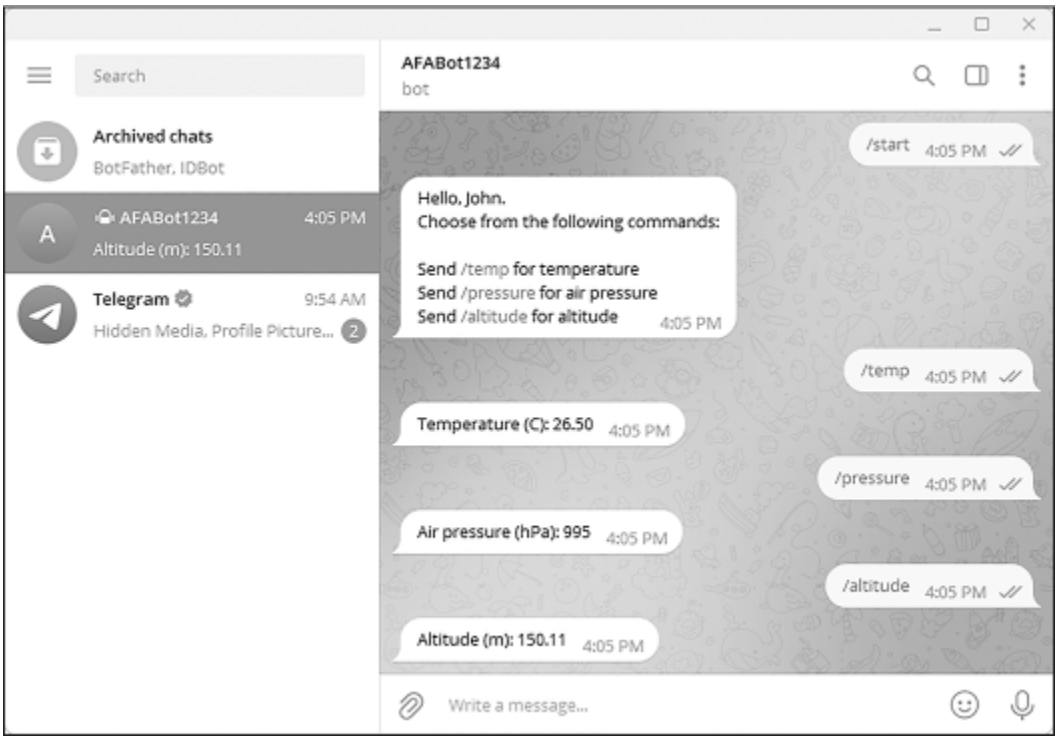


Figure 20-9: Example output from Project #59

Let's see how this works:

```
// Project #59 - ESP32 remote data retrieval via

#include <WiFi.h>
#include <WiFiClientSecure.h>
```

```
  #include <UniversalTelegramBot.h>
  #include <ArduinoJson.h>
❶ #include <Adafruit_BMP085.h>

  Adafruit_BMP085 bmp;

  // Enter your Wi-Fi network SSID and password:
  const char* ssid = "wifiname";
  const char* password = "password";

  // Enter your Telegram bot token and chatID:
  #define botToken "token"
  #define chatID "chatID"

  WiFiClientSecure client;
  UniversalTelegramBot bot(botToken, client);

  // Checks for new messages every 1 second:
  int bot_delay = 1000;
  unsigned long lastRun=0;

  void processMessages(int numNewMessages)
  {
      String welcome; // Used for assembling messa
      Serial.println("Handling New Message");
      Serial.println(String(numNewMessages));

      for (int i = 0; i < numNewMessages; i++)
```

```
    {
        // Chat ID of the requester:
        String chat_id = String(bot.messages[i].
        if (chat_id != chatID)
        {
            bot.sendMessage(chat_id, "Unauthoriz
            continue;
        }

        // Print the received message:
        String user_text = bot.messages[i].text;
        Serial.println(user_text);

        String your_name = bot.messages[i].from_

    ❷ if (user_text == "/start")
        {
            welcome = "Hello, " + your_name + ".
            welcome += "Choose from the followin
            welcome += "Send /temp for temperatu
            welcome += "Send /pressure for air p
            welcome += "Send /altitude for altit
            bot.sendMessage(chat_id, welcome, "
        }

    ❸ if (user_text == "/temp")
        {
            // get BMP180 temperature
```

```
            welcome = "Temperature (C): ";
            welcome += String(bmp.readTemperatur
            bot.sendMessage(chat_id, welcome, "
        }

    ❹ if (user_text == "/pressure")
        {
            // Get BMP180 air pressure, convert
            welcome = "Air pressure (hPa): ";
            welcome += String(bmp.readSealevelPr
            bot.sendMessage(chat_id, welcome, "
        }

    ❺ if (user_text == "/altitude")
        {
            // Get BMP180 altitude:
            welcome = "Altitude (m): ";
            welcome += String(bmp.readAltitude()
            bot.sendMessage(chat_id, welcome, "
        }
    }
}

void setup()
{
  ❻ bmp.begin();
    Serial.begin(115200);
```

```
    // Connect to Wi-Fi:
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);

    client.setCACert(TELEGRAM_CERTIFICATE_ROOT);
    // Add root certificate for api.telegram.org
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.println("Connecting to Wi-Fi…");
    }
    // Print ESP32 Local IP Address:
    Serial.println(WiFi.localIP());
}

void loop()
{
    if (millis() > lastRun + bot_delay)
    {
        int numNewMessages = bot.getUpdates(bot.
        while (numNewMessages)
        {
            Serial.println("Received message");
            processMessages(numNewMessages);
            numNewMessages = bot.getUpdates(bot.
        }
        lastRun = millis();
```

```
        }
    }
```

This sketch's framework for using Telegram is the same as in Project #58. The only differences are the changes required for new hardware used—in this case the BMP180 sensor—and the communication messages for use with Telegram. The sketch includes the required library and initialization for the BMP180 sensor ❶ ❻. It modifies the prior Telegram start message to suit this project's purposes ❷ and programs the responses to temperature ❸, air pressure ❹, and altitude message requests ❺. You can easily modify the sketch for this project to substitute your own required actions to control ESP32 Arduino-compatible boards remotely via Telegram.

Keep the hardware for this project connected. You'll use it in the next project, in which you'll learn to automate data transmission over Telegram.

Project #60: Automating Data Transmission

This project automatically sends data generated by the BMP180 sensor board on a recurring basis to your Telegram account. Rather than using messaging to request data, you can simply

open your Telegram chat to view the latest project updates, watching changes in data in (nearly) real time.

The required hardware is the same as that for Project #59. Once you've set it up, enter and upload the Project #60 sketch, adding your Wi-Fi network credentials in the `#define` fields as usual. After a few moments, open the Telegram bot chat window. You should see the temperature from the BMP180 displayed around every five seconds, as shown in Figure 20-10.

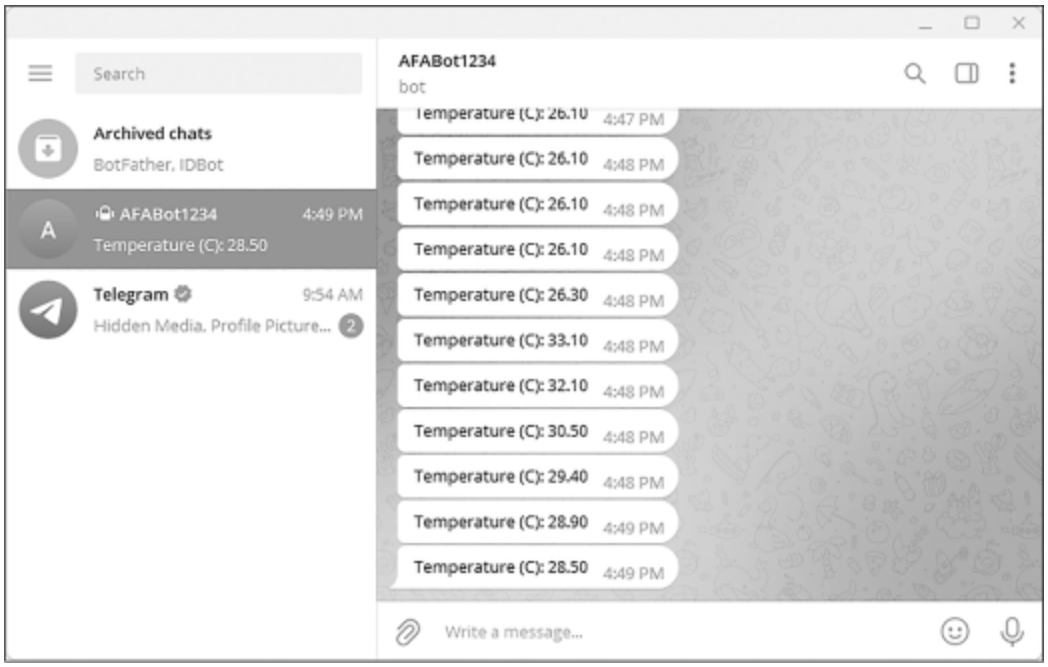


Figure 20-10: Example output from Project #60

Let's see how this works:

```
// Project #60 - Automating data transmission ov

#include <WiFi.h>
#include <WiFiClientSecure.h>
#include <UniversalTelegramBot.h>
#include <ArduinoJson.h>
#include <Adafruit_BMP085.h>

Adafruit_BMP085 bmp;

// Enter your Wi-Fi network SSID and password:
const char* ssid = "wifiname";
const char* password = "password";

// Enter your Telegram bot token and chatID:
#define botToken "token"
#define chatID "chat ID"

WiFiClientSecure client;
UniversalTelegramBot bot(botToken, client);

void setup()
{
    bmp.begin();
    Serial.begin(115200);

    // Connect to Wi-Fi:
    WiFi.mode(WIFI_STA);
```

```
    WiFi.begin(ssid, password);

    client.setCACert(TELEGRAM_CERTIFICATE_ROOT);
    // Add root certificate for api.telegram.org
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.println("Connecting to Wi-Fi…");
    }
    // Print ESP32 Local IP Address:
    Serial.println(WiFi.localIP());
}

void loop()
{
    String messageOut; // Used for assembling me
 ❶ messageOut = "Temperature (C): ";
 ❷ messageOut += String(bmp.readTemperature());
    bot.sendMessage(chatID, messageOut, " ");
    delay(5000);
}
```

As this project is a one-way form of messaging, the required code is much shorter than that of the previous projects. The setup for the BMP180, Wi-Fi and Telegram is still required, but the custom functions to deal with receiving and acting upon messages from the Telegram chat aren’t required. The sketch

simply sends one message on a regular basis, using `void loop()`. It declares the string variable `messageOut` to hold the text to send out, then composes the message with explanation text showing that the data is temperature ❶ and the temperature value from the BMP180 ❷. It then sends the message to the Telegram chat. After a delay of five seconds, the process repeats.

Moving On

In this chapter, you learned to harness the Telegram messaging application for remote project control or data retrieval, without any extra costs for a cloud service or special smartphone app. You could harness these projects to control your own devices with a digital output pin, or you could retrieve data from any sensor that can be connected to your Arduino or compatible boards.

The next chapter introduces you to retrieving the current time from internet time servers.

21

RETRIEVE THE CURRENT TIME FROM AN INTERNET TIME SERVER



When building projects that rely on keeping track of the current time (such as for logging data, creating planned events, or just making your own clocks), it's common to use a real-time clock IC. For increased time and date accuracy, however, you can use an ESP32 Arduino-compatible board to retrieve the current time and date from an internet time server. Using the ESP32 also saves you the cost of an RTC IC or module and a separate Arduino or compatible board when building a new project and allows you to set your required time zone in the code, rather than in a separate interface.

In this chapter, you'll learn about the Network Time Protocol (NTP), install and test the NTP client library for Arduino, and extract the time and date components for your own projects. You'll learn to:

- Build a compact clock with an OLED display
- Build a dual-time clock that displays time at home and at another location
- Build a giant, super-bright digital clock

The Network Time Protocol

Originally standardized in the mid-1980s, NTP is a method of synchronizing connected computers to the Coordinated Universal Time (formally known as Greenwich Mean Time, or GMT). The NTP allows for an accuracy of 1 millisecond or better. In general use, the margin of error is around 100 milliseconds, but a difference of one-10th of a second is easily accurate enough for timing-related Arduino projects.

You can retrieve the time from the NTP server either in the usual 24-hour time and standard date format or in the format of *epoch time*. This is the number of seconds since January 1, 1970, from which you can make your own calculations to

determine the current date and time. Some people use epoch time to calculate their own types of time, such as metric time.

There are many NTP servers available around the world, each with its own address. Generally, the one closest to your location should have the fastest connection to your project, but you might want to use a different server depending on your internet routing. Check *https://www.pool.ntp.org/zone/@* to find the server address most appropriate for you and then use that to replace my default NTP server in the code for this chapter's projects.

You will also need to know your *time zone offset*: the number of hours before or after GMT, or in more contemporary parlance, the Coordinated Universal Time. This is based on the values listed at *https://en.wikipedia.org/wiki/List_of_UTC_offsets*. For example, if you live in London, set your offset to 0, as your time zone falls in the value of UTC (or GMT) 0. If you live in Los Angeles, California, the time zone is UTC −8. This means you'd use a value of −28,800, as LA is 8 hours behind UTC, multiplied by 3,600 to convert to seconds. Finally, the update interval value of 60,000 is the update interval in milliseconds. This is set to a default that equates to one minute, as you don't need to update the time more than once a second.

To prepare to build the projects in this chapter, download and install the NTP Client library for Arduino directly from my GitHub page. Visit *https://github.com/taranais/NTPClient* and click **Code** in the center of the screen. A small menu should appear, as shown in Figure 21-1. Click **Download ZIP** to download the library ZIP file to your PC. If the Arduino IDE ever prompts you to update this library, don't—use the first version you already installed.

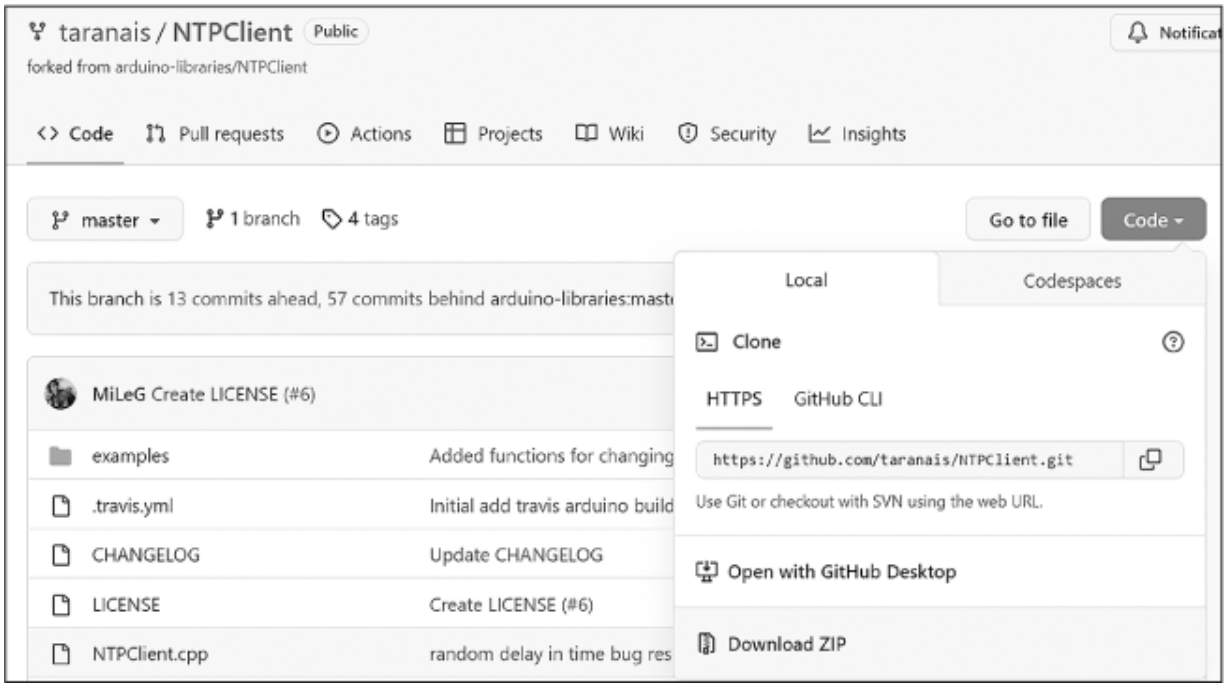


Figure 21-1: Downloading the NTP Client library

Open the Arduino IDE and install the library by selecting **Sketch ▸ Include Library ▸ Add .ZIP Library.** Once the library installs, restart the Arduino IDE to continue.

You're ready to start using internet time in your projects.

Project #61: Retrieving the Time and Date

This project serves as a framework for retrieving the time and date from an NTP server and then extracting the elements of date and time (hours, minutes, and so on) so you can use them in your own projects.

The project uses the Serial Monitor to display the time and date, so for hardware, you'll need only the ESP32 dev board and matching USB cable as used in previous chapters. If you haven't already, turn to Chapter 19 and follow "Configuring the Arduino IDE for the ESP32" through "Testing the ESP32" to install the required board profiles for the Arduino IDE.

Enter and upload the Project #61 sketch, adding your Wi-Fi network credentials in the SSID and password fields near the top of the sketch (as originally explained for Project #55 in Chapter 19). Once the sketch has uploaded, the ESP32 should attempt to connect to your Wi-Fi network and then retrieve and display the time and date information in the Serial Monitor, as shown in Figure 21-2.



Figure 21-2: Example output from Project #61

Let's take a closer look at how this works:

```
  // Project #61 - Retrieving Internet time

❶ #include <NTPClient.h>
  #include <WiFi.h>
  #include <WiFiUdp.h>
  WiFiUDP ntpUDP;

❷ // Enter your Wi-Fi network SSID and password:
  const char* ssid = "SSID";
  const char* password = "password";

  // ntpUDP, time server pool, offset in seconds,
  NTPClient timeClient(ntpUDP, "0.north-america.pool.ntp.org"

  void setup()
  {
      Serial.begin(115200);
      WiFi.mode(WIFI_STA);
```

```
    WiFi.begin(ssid, password);
    Serial.print("Connecting to Wi-Fi ");
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
    timeClient.begin();
}

void loop()
{
    String dateString;
    String date, month, year;

 ❸ timeClient.update();
    Serial.println(timeClient.getEpochTime());
    Serial.println(timeClient.getFormattedDate()
    Serial.println(timeClient.getFormattedTime()

    // Retrieve and display day of week
 ❹ Serial.print("Today is ");
    switch (timeClient.getDay())
    {
        case 0: Serial.println("Sunday"); break;
        case 1: Serial.println("Monday"); break;
        case 2: Serial.println("Tuesday"); break
        case 3: Serial.println("Wednesday"); bre
```

```
        case 4: Serial.println("Thursday"); brea
        case 5: Serial.println("Friday"); break;
        case 6: Serial.println("Saturday"); brea
        case 7: Serial.println("Sunday"); break;
    }

    // Retrieve hours, minutes, and seconds sepa
    if (timeClient.getHours() < 10)
    {
        Serial.print("0");
    }
    Serial.print(timeClient.getHours());
        Serial.print("-");
  if (timeClient.getMinutes() < 10)
  {
          Serial.print("0");
    }
      Serial.print(timeClient.getMinutes());
        Serial.print("-");
    if (timeClient.getSeconds() < 10)
    {
        Serial.print("0");
    }
    Serial.println(timeClient.getSeconds());

❺ // Extract data from formatted date result ar
    dateString = timeClient.getFormattedDate();
    date = dateString.substring(8, 10);
```

```
    Serial.print(date);
    Serial.print("/");
    month = dateString.substring(5, 7);
    Serial.print(month);
    Serial.print("/");
    year = dateString.substring(0, 4);
    Serial.println(year);
    Serial.println();
    delay(1000);
}
```

First, the sketch includes the required libraries for Wi-Fi and the NTP client and creates an instance of the timer server `ntp` ❶. Next, it stores the name and password of the Wi-Fi network for use by the Wi-Fi library ❷.

The `NTPClient timeClient()` function accepts the NTP server address to use, the required time zone offset, and the update interval. As mentioned in the previous section, try the default time server address supplied in the code or find one closer to your location. For the time zone offset, replace the value in the code with the offset appropriate for your location.

In `void loop()`, various string variables are declared for storing time and date information, followed by a force update of the

time client ❸ to get the latest time and date from the server. The next three functions demonstrate how to retrieve the time and date information in different formats and display their values on the Serial Monitor. The first is `timeClient.getEpochTime()`, which retrieves the epoch time. Next, `timeClient.getFormattedDate()` shows the entire time and date in the previously set time zone with the date in *yyyy-mm-dd* format, followed by the letter *T* (for "time") and then the current time in 24-hour time. Finally, `timeClient.getFormattedTime()` shows the current time in *hh*:*mm*:*ss* format.

Next, the sketch demonstrates how to extract individual pieces of time and date information. The `timeClient.getDay()` function returns an integer between 0 and 6 that represents the day of the week, from Sunday through Saturday, respectively. This is used in a `switch…case` function ❹ to determine the current day and display the name in the Serial Monitor. The sketch displays the individual values for hours, minutes, and seconds with the functions `timeClient.getHours()`, `timeClient.getMinutes()`, and `timeClient.getSeconds()`, which all return their respective values as integers. The code also includes a test for values less than 10 for minutes and seconds to add a 0 if necessary for the proper

time format. This is so that you don't end up with a time display of 9:5:00 for five minutes past 9 AM, for example.

Getting the day, month, and year is a little different. The sketch first retrieves the entire time and date using `timeClient.getFormattedDate()`❺ and then stores the values for the date, month, and year in string variables using `.substring` functions. For example, the year falls within the first four characters of the string `dateString`, so the function `dateString.substring(0,4)` will return those first four characters. Finally, there's a delay of 1 second before the process repeats.

If your project loses its connection to your Wi-Fi network, the time will stay the same and not update until the ESP32 can reconnect to the NTP server, at which point the time and date will automatically update.

The following projects use this sketch as a framework for retrieving the time and date and displaying the results on different devices.

Project #62: Displaying the Time and Date on an OLED

This project demonstrates how to display the time and date from an NTP server on the inexpensive OLED display you first

used in Chapter 10. You could modify this project to control other types of displays, such as LCDs or even Nixie tubes, if so inclined.

You'll need the following hardware:

- An ESP32 Arduino-compatible board and USB cable
- A 128 × 32 pixel, 0.91-inch OLED display
- A solderless breadboard
- Various jumper wires

If this is your first time using the OLED display, please turn to Chapter 10 and follow the instructions in "Graphic OLED Displays" on page 149 to the point where you have tested the OLED and then assemble the circuit as shown in Figure 21-3.

NOTE

Although your OLED might be marked as a 5 V device, the specified model also works on 3.3 V without a level converter.

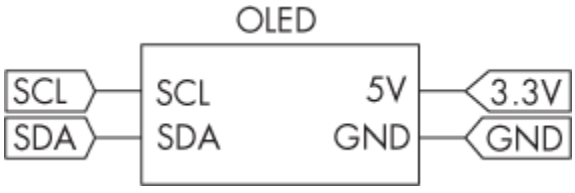


Figure 21-3: The schematic for Project #62

Enter and upload the Project #62 sketch, updating your Wi-Fi network details as appropriate. The OLED should display a message telling you the ESP32 is attempting to connect to Wi-Fi, as shown in Figure 21-4.



Figure 21-4: The project connecting to Wi-Fi

Once the project has connected to the NTP server, the OLED should display the current time and date, along with the day of the week, as shown in Figure 21-5.



Figure 21-5: Example output from Project #62

Let’s take a closer look at how this works:

```
// Project #62 - OLED NTP clock

#include <NTPClient.h>
#include <WiFi.h>
#include <WiFiUdp.h>
#include <U8g2lib.h>
#include <Wire.h>
U8G2_SSD1306_128X32_UNIVISION_F_HW_I2C u8g2(U8G2_

// Enter your Wi-Fi network SSID and password
const char* ssid = "SSID";
const char* password = "Password";
WiFiUDP ntpUDP;

❶ // ntpUDP, time server pool, offset in seconds,
NTPClient timeClient(ntpUDP, "0.us.pool.ntp.org", 36000,

void setup()
{
    Wire.begin();
    u8g2.begin();
    u8g2.setFont(u8g2_font_9x18_tr);
    Serial.begin(115200);
  ❷ WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to Wi-Fi ");
```

```
    while (WiFi.status() != WL_CONNECTED)
    {
        u8g2.clearBuffer();
        u8g2.drawStr(0, 16, "Connecting");
        u8g2.drawStr(0, 31, "to Wi-Fi"…");
        u8g2.sendBuffer();
    }
    timeClient.begin();
}

void loop()
{
    int hours, minutes, seconds;
    int date, month, year, dayOfWeek;
    String hh, mm, ss, dateString;
    String OLED1, OLED2;

    timeClient.update();

    // Assemble time:
 ❸ hours = timeClient.getHours();
    minutes = timeClient.getMinutes();
    seconds = timeClient.getSeconds();

    if (hours < 10) {hh = hh + "0";}
    if (minutes < 10) {mm = mm + "0";}
    if (seconds < 10) {ss = ss + "0";}
    hh = hh + String(hours);
```

```
  mm = mm + String(minutes);
  ss = ss + String(seconds);
❹ OLED1 = hh + ":" + mm + ":" + ss;

  // Assemble date:
❺ dateString = timeClient.getFormattedDate();
  switch (timeClient.getDay())
  {
      case 0: OLED2 = OLED2 + "Sun "; break;
      case 1: OLED2 = OLED2 + "Mon "; break;
      case 2: OLED2 = OLED2 + "Tue "; break;
      case 3: OLED2 = OLED2 + "Wed "; break;
      case 4: OLED2 = OLED2 + "Thu "; break;
      case 5: OLED2 = OLED2 + "Fri "; break;
      case 6: OLED2 = OLED2 + "Sat "; break;
      case 7: OLED2 = OLED2 + "Sun "; break;
      }

❻ OLED2 = OLED2 + dateString.substring(8, 10);
  OLED2 = OLED2 + "/";
  OLED2 = OLED2 + dateString.substring(5, 7);
  OLED2 = OLED2 + "/";
  OLED2 = OLED2 + dateString.substring(0, 4);

  // Show time and date on OLED:
❼ u8g2.clearBuffer();
  u8g2.drawStr(0, 16, OLED1.c_str());
  u8g2.drawStr(0, 31, OLED2.c_str());
```

```
    u8g2.sendBuffer();
    delay(1000);
}
```

The sketch includes and initializes the libraries required for Wi-Fi, the NTP client, the I²C bus, and the OLED. Set the fields for your Wi-Fi network details, as usual.

The sketch then initiates an instance of the NTP client and sets up the pool server, time zone offset, and update intervals ❶. In `void setup()`, it starts the I²C bus, OLED display library, and Serial Monitor for debugging and then initializes the Wi-Fi. The next block of code causes the "Connecting …" message to appear on the OLED during the connection process ❷. The sketch then starts the NTP client library.

The code in `void loop()` aims to assemble the time information and display it on the top line of the OLED and then assemble the date information and display it on the bottom line. This starts by retrieving the time ❸ and then adding a 0 to the string `OLED1`, which will contain the time data for hours, minutes, and seconds if the individual piece of data is below 10. The hour, minute, and second values are then added into their respective

string variables, which are finally concatenated into one string for display ❹.

After retrieving the date string ❺, the code retrieves the day of week and places it inside the string `OLED2`, which it uses to display the entire date via the `switch…case` function. Next, it retrieves the day, month, and year and adds that data to the main date string, including with separators for a neat display ❻.

Finally, the sketch sends the two lines of data, `OLED1` and `OLED2`, to the OLED for display ❼. As demonstrated in the code, you must add the suffix `.c_str` to the variable name to tell the OLED library to convert the variable from a string to usable data.

Keep the hardware for this project assembled. In the next project, you'll use it to make a dual-zone clock that takes advantage of the offset function in the NTP library.

Project #63: Displaying Two Time Zones on an OLED

This project shows you how to display the time in two zones simultaneously on the OLED display used in the previous project. This might be useful if you're a HAM radio operator or if you often communicate with friends, family, or colleagues in different time zones.

The required hardware and assembly are the same as Project #62. Once the hardware is ready, enter and upload the Project #63 sketch. Don't forget to add your Wi-Fi network credentials into this sketch. After the ESP32 connects to the Wi-Fi network, the OLED should display the "home zone" time for the east coast of Australia (where I live) and display the "away zone" time for San Francisco in the United States (the home of No Starch Press), as shown in Figure 21-6.



Figure 21-6: Example display for Project #63

To learn to change the time zones, let's take a closer look at the sketch:

```
// Project #63 - OLED dual-zone NTP clock

#include <NTPClient.h>
```

```
#include <WiFi.h>
#include <WiFiUdp.h>
#include <U8g2lib.h>

#include <Wire.h>
U8G2_SSD1306_128X32_UNIVISION_F_HW_I2C u8g2(U8G2_

// Enter your Wi-Fi network SSID and password:
const char* ssid = "SSID";
const char* password = "Password";
WiFiUDP ntpUDP;

// ntpUDP, time server pool, offset in seconds, u
// Starts with home offset (e.g. 36000 for UTC +
NTPClient timeClient(ntpUDP, "0.us.pool.ntp.org", 36000,

void setup()
{
    Wire.begin();
    u8g2.begin();
    u8g2.setFont(u8g2_font_9x18_tr);

    Serial.begin(115200);
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to Wi-Fi ");
    while (WiFi.status() != WL_CONNECTED)
    {
```

```
        u8g2.clearBuffer();
        u8g2.drawStr(0, 16, "Connecting");
        u8g2.drawStr(0, 31, "to Wi-Fi…");
        u8g2.sendBuffer();
    }
    timeClient.begin();
}

void loop()
{
    int hours, minutes, seconds;
    String hh, mm, ss, dateString;
    String OLED1, OLED2;
    timeClient.update();

    // Assemble home time (BNE, UTC + 10):
 ❶ timeClient.setTimeOffset(36000);
    hours = timeClient.getHours();
    minutes = timeClient.getMinutes();
    seconds = timeClient.getSeconds();

    if (hours < 10) {hh = hh + "0";}
    if (minutes < 10) {mm = mm + "0";}
    if (seconds < 10) {ss = ss + "0";}
    hh = hh + String(hours);
    mm = mm + String(minutes);
    ss = ss + String(seconds);
 ❷ OLED1 = "Home " + hh + ":" + mm + ":" + ss;
```

```
  ❸ hh = " "; mm = " "; ss = " ";

    // Assemble away time (SFO, UTC - 8):
    timeClient.setTimeOffset(-28800);
    hours = timeClient.getHours();
    minutes = timeClient.getMinutes();
    seconds = timeClient.getSeconds();

    if (hours < 10) {hh = hh + "0";}
    if (minutes < 10) {mm = mm + "0";}
    if (seconds < 10) {ss = ss + "0";}
    hh = hh + String(hours);
    mm = mm + String(minutes);
    ss = ss + String(seconds);
  ❹ OLED2 = "SFO " + hh + ":" + mm + ":" + ss;

    // Show time and date on OLED:
  ❺ u8g2.clearBuffer();
    u8g2.drawStr(0, 16, OLED1.c_str());
    u8g2.drawStr(0, 31, OLED2.c_str());
    u8g2.sendBuffer();
    delay(1000);
}
```

This sketch is similar to the one for Project #62, in that it also assembles two lines of data to display on the OLED. To create a dual-zone clock, the code begins by setting the time zone offset

for the first zone ❶. The example home time zone in the sketch is for Brisbane, which is UTC +10, so the offset value for the `timeClient.setTimeOffset()` function is 36,000 (3,600 multiplied by +10 hours), but you can change this to match whichever time zone you like. Following this, the sketch retrieves the time data and assembles it into the string variable `OLED1` ❷, preceding it with the label `Home`. The sketch then blanks out the string variables used to temporarily hold retrieved time data for reuse ❸.

The same process repeats for a second time zone. I've used the example of San Francisco, which is UTC –8, meaning that the offset value for the `timeClient.setTimeOffset()` function is –28,800 (3,600 multiplied by –8 hours). The sketch retrieves and assembles the time date into the string variable `OLED2` ❹, starting with a label for the away zone `SFO`. Finally, it sends the two strings of data to the OLED for display ❺, followed by a short delay before the entire process repeats.

For a challenge, you might try modifying this project to scroll through various time zones at the press of a button.

Project #64: Building a Giant Digital Clock

For this chapter's final project, let's have some fun and build a giant digital clock using the seven-segment LED modules from Project #27 in Chapter 8. Once assembled, you could use this project to keep time in your workshop as an example of your prowess.

If you don't already have the seven-segment LED modules set up, revisit Project #27 and build the four required display boards. (For a challenge, you might make six boards so you can display seconds in addition to minutes and hours.) Apart from the display boards, wiring, and 9 V DC 1A power supply covered in that project, you will only need the ESP32 board used in this chapter and the usual jumper wires and USB cable.

NOTE

The TPIC6B595 shift register ICs used in the LED display boards are both 3.3 V and 5 V tolerant, so you don't need to connect a level shifter between them and the ESP32 board.

Once you have connected the four display boards, upload the Project #64 sketch. Using the connections diagrammed in Figure 21-7, connect the ESP32 to the first display board (the one on the left of the figure).

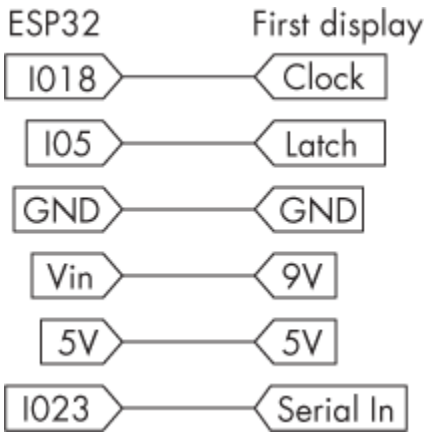


Figure 21-7: The connection layout between the ESP32 and first display board

Connect the power to the first display board. The ESP32 should connect to the Wi-Fi network and retrieve the time. Stand back to see all four digits light up with the current time, as shown in Figure 21-8.

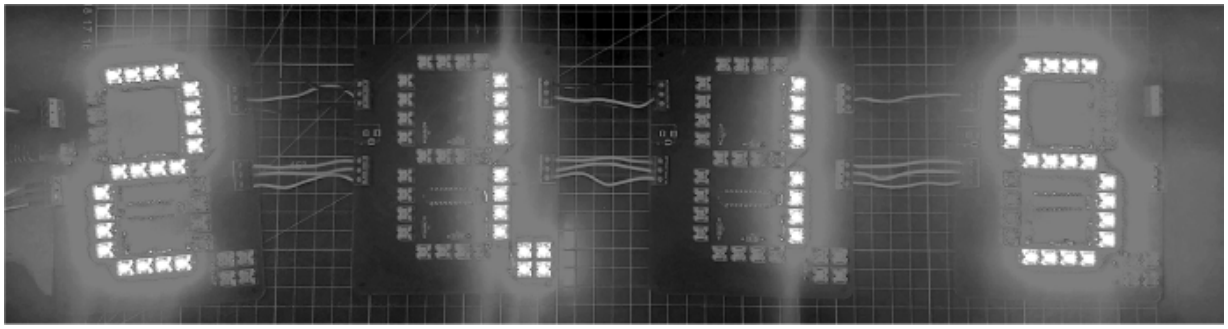


Figure 21-8: Project #64 in action

Let's see how this works in the sketch:

```
// Project #64 - Giant LED NTP Clock

#include <NTPClient.h>
```

```
#include <WiFi.h>
#include <WiFiUdp.h>

// Enter your Wi-Fi network SSID and password:
const char* ssid = "SSID";
const char* password = "Password";
WiFiUDP ntpUDP;

// ntpUDP, time server pool, offset in seconds, u
// Starts with home offset (e.g. 36000 for UTC +
NTPClient timeClient(ntpUDP, "0.us.pool.ntp.org", 36000,
❶ #define LATCH 5  // Latch RCK pin
#define CLOCK 18 // Clock SRCK pin
#define DATA 23  // Data SERIN pin

int digits[] = { B00111111,   // 0
                 B00000110,   // 1
                 B01011011,   // 2
                 B01001111,   // 3
                 B01100110,   // 4
                 B01101101,   // 5
                 B01111101,   // 6
                 B00000111,   // 7
                 B01111111,   // 8
                 B01100111}; // 9

void sendNumbers(int numbers[], int dp)
{
```

```
    digitalWrite(LATCH, LOW);
    for (int i = 0; i < 4; i++)
    {
        int dig_idx = numbers[i];
        if (dp == i)
        {
          // Display the digit:
          shiftOut(DATA, CLOCK, MSBFIRST, digits
        } else
        {
          shiftOut(DATA, CLOCK, MSBFIRST, digits
        }
    }
    digitalWrite(LATCH, HIGH);
}

void setup()
{
    pinMode(LATCH, OUTPUT);
    pinMode(CLOCK, OUTPUT);
    pinMode(DATA, OUTPUT);
    Serial.begin(115200);
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to Wi-Fi…");
    while (WiFi.status() != WL_CONNECTED)
    {
        Serial.print(".");
```

```
        delay(500);
    }
  ❷ timeClient.begin();
}

void loop()
{
    int numbers[4];
    int hours, minutes;
  ❸ timeClient.update();

    hours = timeClient.getHours();
    minutes = timeClient.getMinutes();

  ❹ numbers[3] = hours / 10;
    numbers[2] = hours % 10;
    numbers[1] = minutes / 10;
    numbers[0] = minutes % 10;

  ❺ sendNumbers(numbers, 2);
    delay(1000);
}
```

The operations in this sketch should be familiar to you at this point. The code includes the required libraries and Wi-Fi network details. It then creates an instance of the NTP client with the pool server, time zone offset, and update interval

details. For easy reference, the sketch defines the pin numbers used for the outputs to the shift registers on the display boards ❶.

The array `digits[]` holds the 10 bytes of data used to define how a digit is displayed on the LED boards, where each bit represents one of the seven segments used to comprise a digit. The custom `sendNumbers()` function accepts an array of four integers to display on the four LED boards, as well as another integer that, if used, turns on the decimal point on the matching LED board.

In `void setup()`, the code initializes the required digital pins, sets up the Serial Monitor for debugging, and starts the Wi-Fi connection and the NTP client ❷. The code in `void loop()` updates the NTP client for the latest time information ❸ and then retrieves and stores the hours and minutes in variables. The time must be broken down into single digits before it's sent to the LED boards, so the use of modulo and division ❹ on the hours and minutes values returns the first and second digits, which are then stored in the array `numbers[]`. Finally, the sketch sends those digits to the display using `sendNumbers()`, with the decimal point placed at the second digit as separator between hours and minutes ❺.

For a final challenge, you can try altering the code in any of the projects in this chapter to use 12-hour time rather than 24-hour time. To do so, you must subtract 12 from any hour value that is 13 or more before displaying that hour. You could, of course, also add a test for AM and PM.

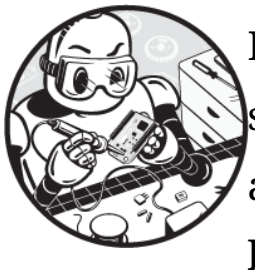
Moving On

In this chapter, you learned how to harness accurate time and date information from the internet and show that information using various display types. Armed with these skills, you might try building additional clocks with various displays as practice for other projects that require the time and date.

The next chapter covers another way to use the ESP32, demonstrating how to capture data from the dev board into the Google Sheets app for analysis.

22

CAPTURE AND LOG DATA TO GOOGLE SHEETS



You can use internet-connected boards like the ESP32 to capture data and store it in online spreadsheets for real-time monitoring and later analysis. In this chapter, you'll use this method to build a single project that sends time and temperature data to a Google Sheets spreadsheet tool within Google Workspace, a collection of cloud-computing collaboration tools that are accessible from most web-enabled devices.

You can use this framework in future projects to record any other type of data you can gather with Arduino-compatible projects. For example, you might want to share weather data at

a remote location with other colleagues or monitor sensor values in a laboratory experiment from your cell phone while on holiday elsewhere.

Project #65: Logging Time and Temperature Data

This project demonstrates how to send the time and date from an NTP server, along with the temperature and air pressure from a BMP180 sensor board, to Google Workspace's Google Sheets tool via the ESP32 board.

You'll need the following hardware:

- An ESP32 dev board and matching USB cable
- A solderless breadboard
- Assorted jumper wires
- A BMP180 temperature and pressure sensor board

If you haven't already, turn to Chapter 19 and complete the instructions in "Configuring the Arduino IDE for the ESP32" through "Testing the ESP32" to get your ESP32 working. Also make sure you've completed "BMP180 Sensors" in Chapter 10 and read Chapter 21 to familiarize yourself with NTP servers.

To build this project, you'll start by configuring a Google Workspace account to receive data from the ESP32 and then set

up the ESP32 board and sensor for data transmission.

Preparing the Google Sheets Document

Google Sheets is a cloud-based spreadsheet included as part of the Google Workspace cloud-based product. If you don't already have one, set up a Google Workspace account at *https://workspace.google.com*. A free trial is available if you want to experiment without commitment, but at time of writing, the price for a monthly subscription is less than US$10 a month.

Once you've logged into your account, visit *https://docs.google.com/spreadsheets* to view the Google Sheets start page, as shown in Figure 22-1.

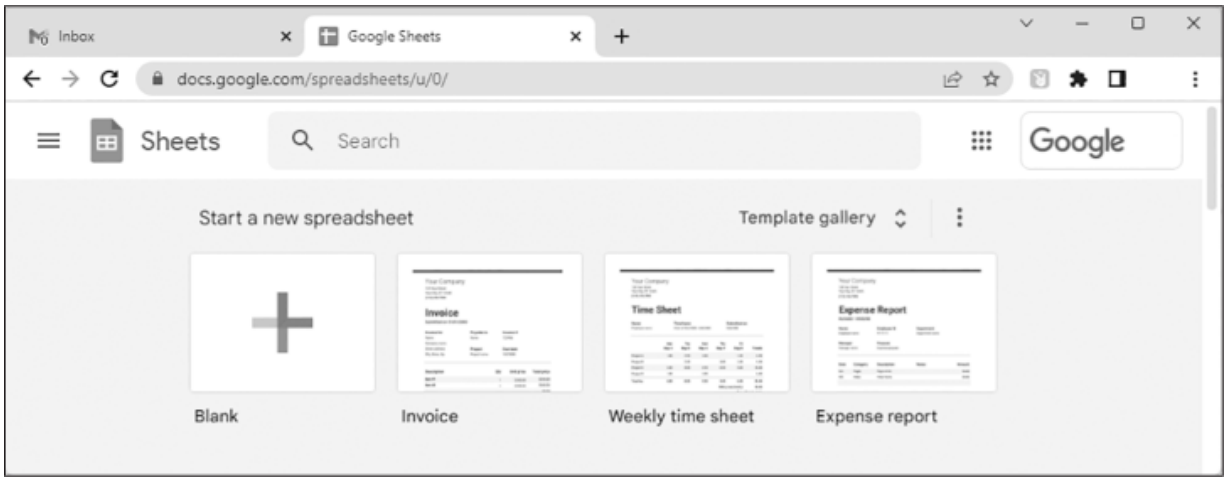


Figure 22-1: The Google Sheets start page

Create a new, blank spreadsheet by clicking the **Blank** icon under Start a new spreadsheet. A new tab for that spreadsheet will appear, as shown in Figure 22-2.

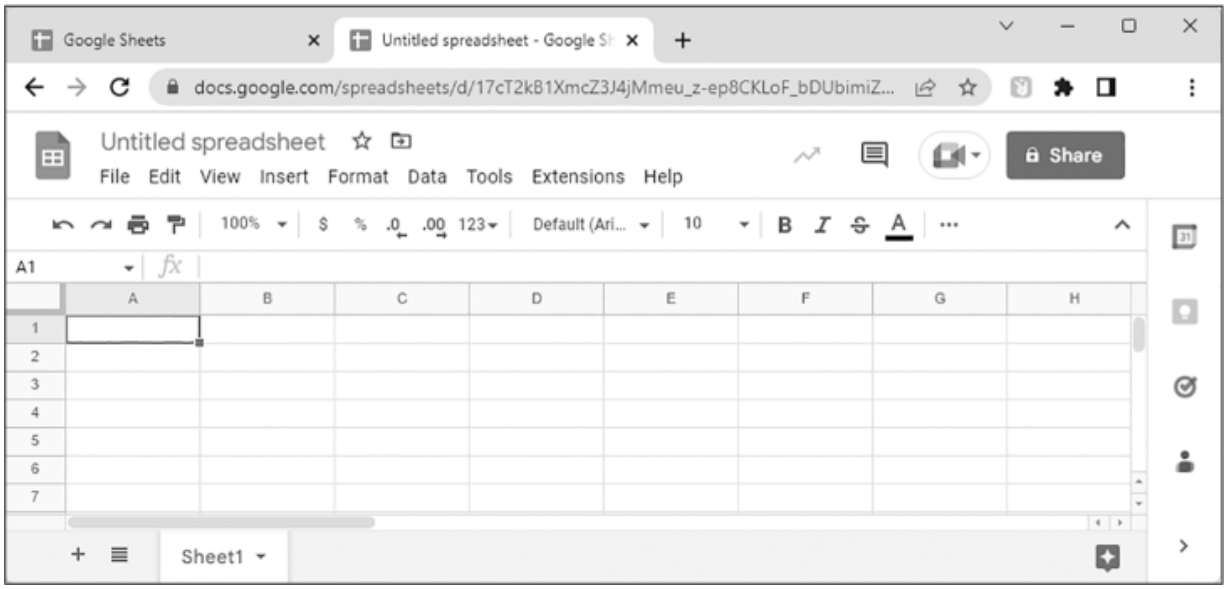


Figure 22-2: A blank spreadsheet

Click **Untitled spreadsheet** and enter an appropriate name for a spreadsheet that tracks temperature and air pressure data for your location—for example, **OfficeWeather**—and then press ENTER. Next, rename the page label at the bottom of the page from Sheet1 to **office**.

Now give the columns in row 1 the following headings, starting from cell A1, as shown in Figure 22-3: **date, month, year, hour, minute, second, temperature, airpressure**. These will match the data sent from the ESP32 board, as you'll see shortly.

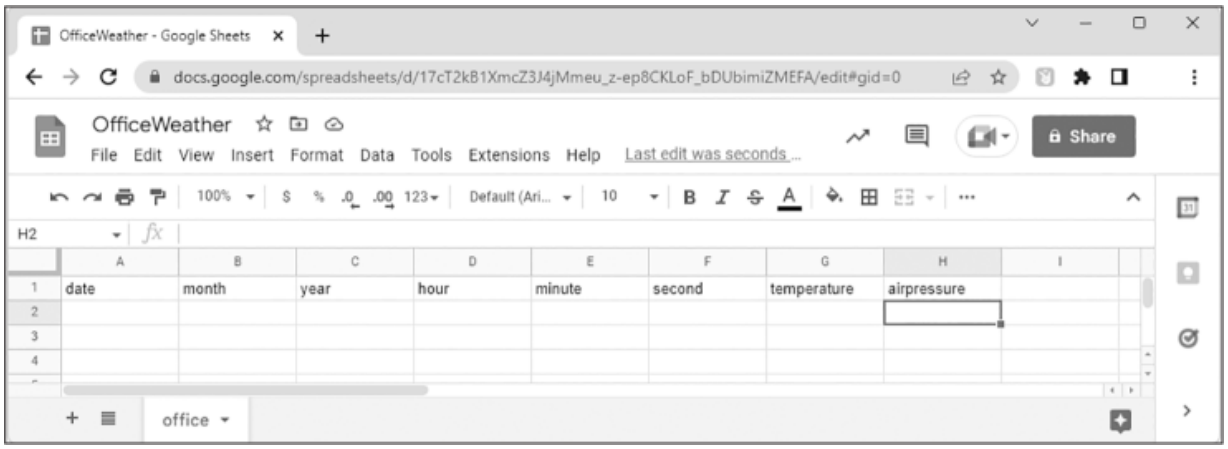


Figure 22-3: The spreadsheet setup, with names and data headings

Note that column headings in Google Sheets, must always be lowercase and not contain any spaces or symbols.

Wait for Google to automatically save the sheet, then retrieve and save the sheet ID. This is the long group of characters between */d* and */edit* in the sheet's URL, as shown in Figure 22-4. Copy the sheet ID and paste it into a text document so you can easily retrieve it later.

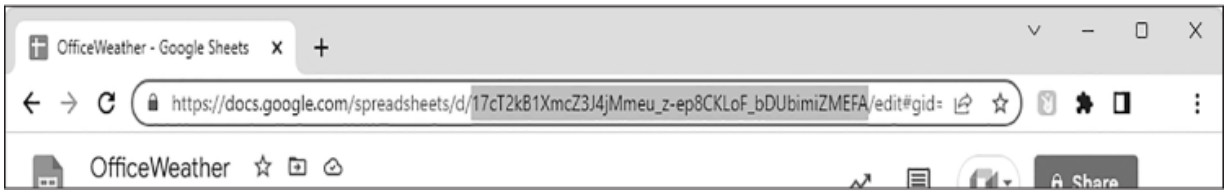


Figure 22-4: Example of a sheet ID highlighted in a URL

Your sheet is now ready to receive data. For now, leave it open in the web browser.

The Google Apps Script

Your ESP32 hardware will send data to your Google sheet by making an HTTP call with the data attached to the end of the HTTP address. To enable that process, you must upload a small piece of JavaScript code to the Google servers that decodes HTTP calls from the ESP32 board, separating the data and placing it in the correct column in the sheet.

Open the Google Apps script tool by selecting **Extensions** ▸ **Apps Script** in Google Sheets.

A new tab for the Apps Script editor will open in the web browser, as shown in Figure 22-5.

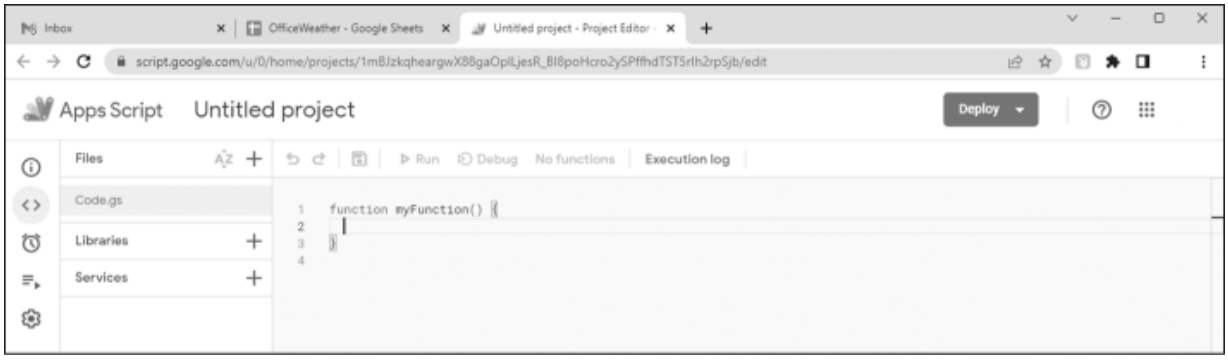


Figure 22-5: Opening a new tab

Click **Untitled project** and enter a name for the script, such as **OfficeWeatherScript**. Copy the following code and paste it over the text in line 1 of the editor:

```
var sheet_id = "sheet_ID";
var sheet_name = "office";
function doGet(e)
{
    var ss = SpreadsheetApp.openById(sheet_id);
    var sheet = ss.getSheetByName(sheet_name);
    var date = Number(e.parameter.date);
    var month = Number(e.parameter.month);
    var year = Number(e.parameter.year);
    var hour = Number(e.parameter.hour);
    var minute = Number(e.parameter.minute);
    var second = Number(e.parameter.second);
    var temperature = Number(e.parameter.tempera
    var airpressure = Number(e.parameter.airpres
    sheet.appendRow([date,month,year,hour,minute
}
```

Insert the sheet ID you retrieved earlier between the quotes in the first line of the code. The sheet label name, `office`, appears between the quotes in the next line. Each column of data to store is represented by the `var` declarations in the order they appear across row 1 of the spreadsheet (`date` to `airpressure`).

Finally, the `sheet.appendrow()` function writes the received data to a new row in the spreadsheet. The order in which the variables appear in the function determines the order in which

the data is written to the sheet. Each set of data received will be saved to the next empty row, until you hit the maximum of 40,000 rows in a Google sheet. From that point, you cannot log any more data.

Your editor page should now resemble Figure 22-6. The script editor will highlight any syntax errors, so check for errors before moving on.

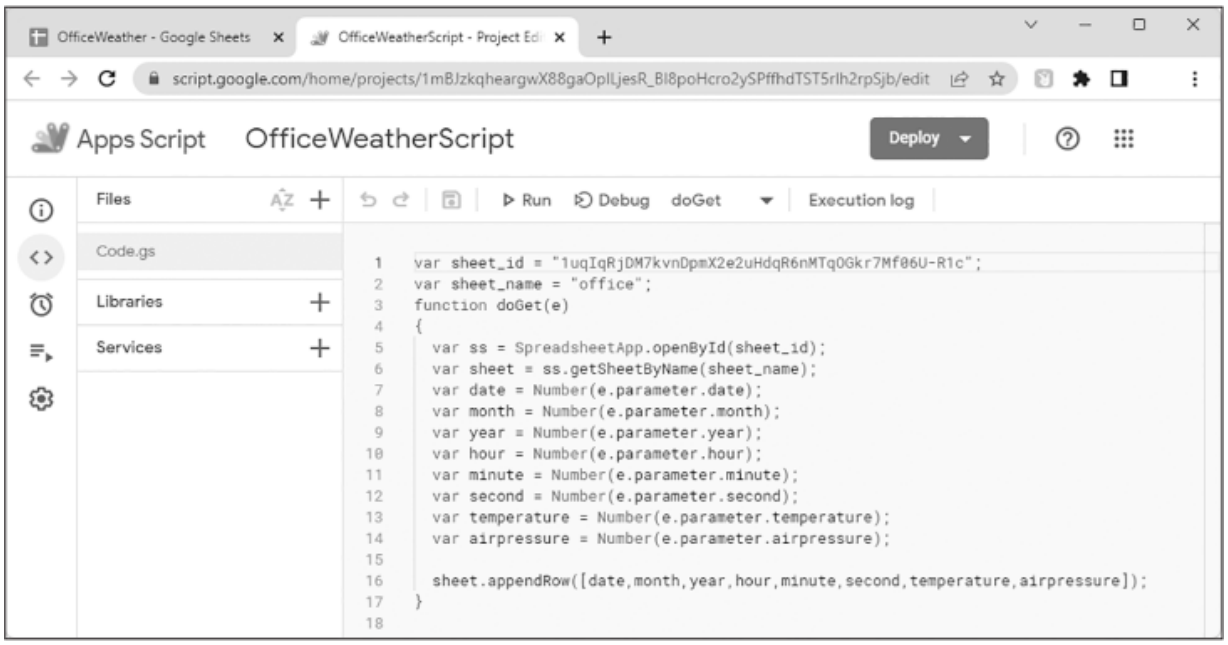


Figure 22-6: The completed app script

To deploy the script, select **Deploy** ▸ **New deployment**. When the New deployment window appears, click the gear icon to select the deployment type. In the drop-down menu that

appears, click **Web app** to open the configuration dialog, which has three fields.

You will be presented with the Configuration dialog, which has three fields. In the first, New Description, enter **OfficeWeather** (or whatever you chose to name your sheet). Select **Me** for the Web app field, and select **Anyone** for the Who has access field, as shown in Figure 22-7. Click **Deploy**.

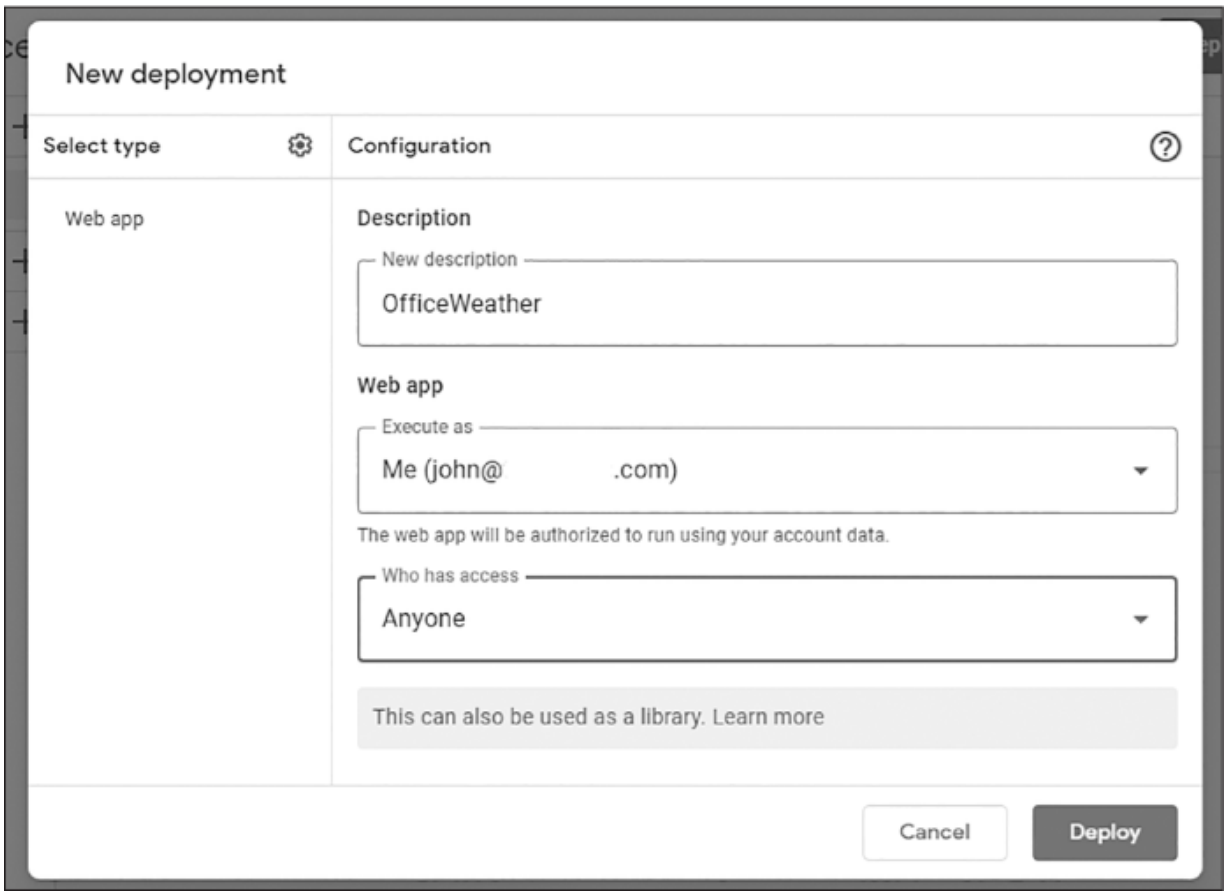


Figure 22-7: The deployment configuration page

A pop-up should prompt you to authorize access to the account. Click **Authorize access**.

If prompted to log into your Google account again, do so. You will be presented with a new window showing your spreadsheet's deployment ID and web app URL.

Use the Copy links in the window to copy each of the items and save them in the document containing your sheet ID. You can now close the Apps Script tab in the web browser.

Before setting up this project's hardware, test that your script is working. Using a text editor, enter the web app URL on a new line and then add the following text to the end of the URL:

```
?date=12&month=1&year=2023&hour=12&minute=37&sec
```

The text appended to the URL describes the data being sent to the spreadsheet in the format *column=data*, with each column and data pair separated by a question mark. The data in this line of code is intended just to test the script—you'll program the ESP32 dev board to send its own data shortly.

Copy the entire URL into the address field of a web browser and press ENTER. Two things should happen. First, the web browser

should return a message telling you that the script completed but did not return anything, as shown in Figure 22-8.



Figure 22-8: A message returned by the Google Apps script

The message means the script worked but didn't send any data back to the web browser. That's okay—the data went to the spreadsheet instead. Check your sheet and you should see the data on a new row, as shown in Figure 22-9. If you like, you can go back and edit the data at the end of the entered URL, resubmit the URL, and see another row of data entered into the spreadsheet.

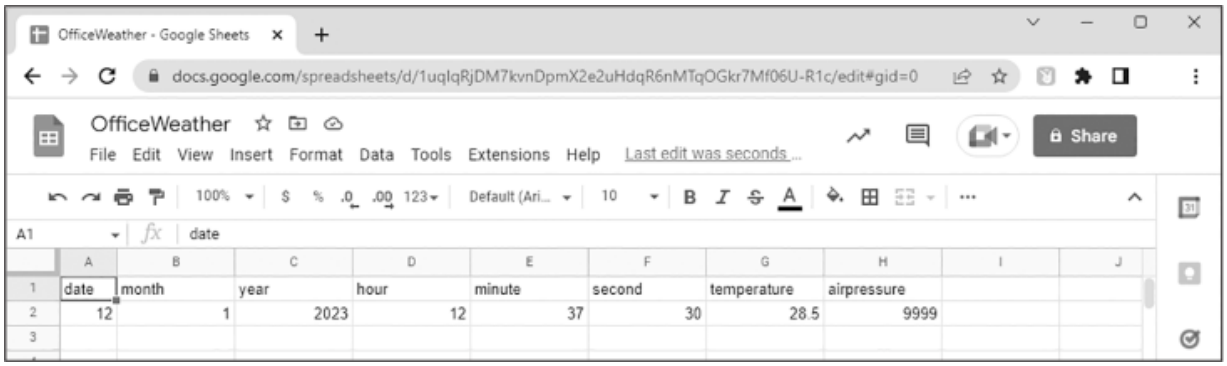


Figure 22-9: The spreadsheet after a successful HTML call

This process is the basis for data logging from the ESP32: the sketch will arrange the data captured from the sensor along with the time and date into one URL and then perform an HTTP call that the Google Apps script you created earlier will process to enter data into the spreadsheet. If this test didn't work for you, check that you've correctly followed all instructions provided in the project up to this point.

Preparing the Hardware

Assemble your hardware as shown in Figure 22-10. Note that although your BMP180 board may be marked as a 5 V device, it will work on 3.3 V and the ESP32 board without a level converter.

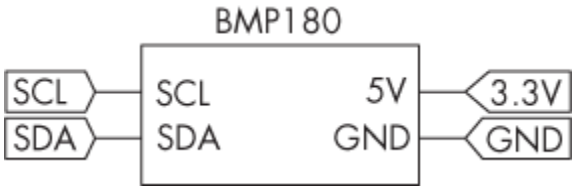
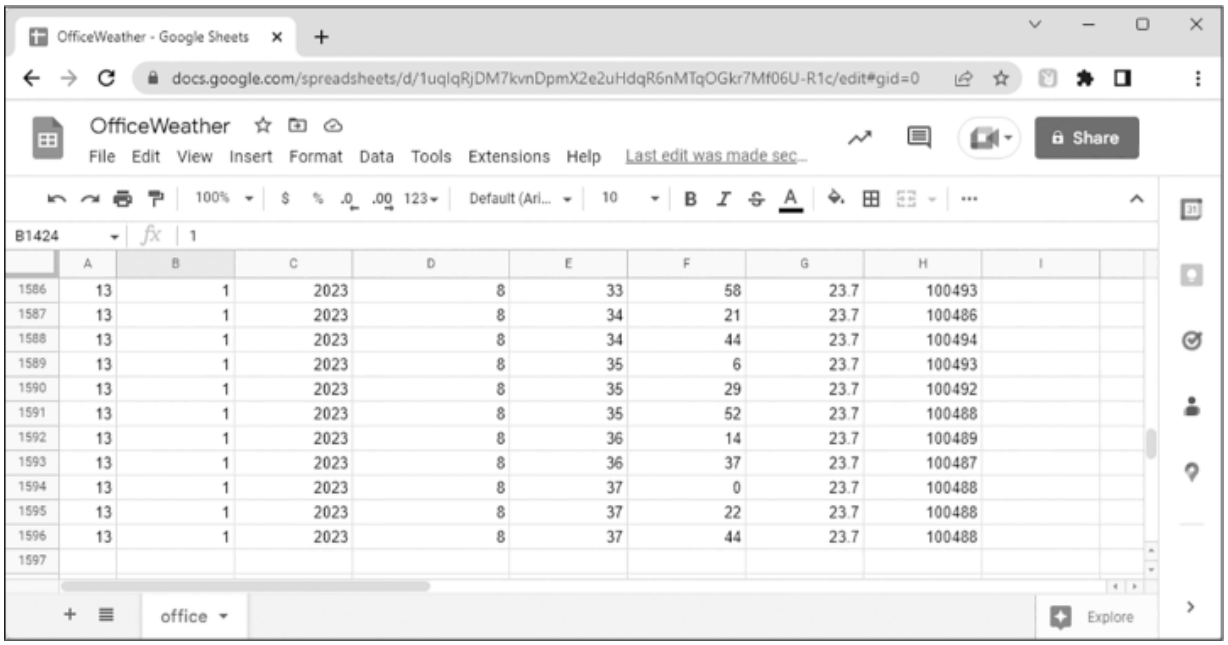


Figure 22-10: The schematic for Project #65

Enter and upload the sketch, including your Wi-Fi network name (SSID) and password in the fields toward the beginning of the program, as originally explained in Project #55 in Chapter 19. Don't forget to change the offset for your time zone, as described in "Network Time Protocol" in Chapter 21, and insert your web app URL into the appropriate field. The ESP32 should connect to the Wi-Fi network and then retrieve the time and date from an NTP server and the temperature and air pressure from the BMP180. It should combine the data into a string comprised of the web app URL and the data and then complete a HTTP call to the Google server.

A moment later, you should see the data appear in the spreadsheet. Leave the sheet open to see it update nearly in real time, as shown in Figure 22-11.



	A	B	C	D	E	F	G	H
1586	13	1	2023	8	33	58	23.7	100493
1587	13	1	2023	8	34	21	23.7	100486
1588	13	1	2023	8	34	44	23.7	100494
1589	13	1	2023	8	35	6	23.7	100493
1590	13	1	2023	8	35	29	23.7	100492
1591	13	1	2023	8	35	52	23.7	100488
1592	13	1	2023	8	36	14	23.7	100489
1593	13	1	2023	8	36	37	23.7	100487
1594	13	1	2023	8	37	0	23.7	100488
1595	13	1	2023	8	37	22	23.7	100488
1596	13	1	2023	8	37	44	23.7	100488
1597								

Figure 22-11: Example output from Project #65

Let's see how this works:

```
// Project #65 - Logging time and temperature da

#include <WiFi.h>
#include <HTTPClient.h>
#include <NTPClient.h>
#include <WiFiUdp.h>
#include <Adafruit_BMP085.h>

WiFiUDP ntpUDP;
Adafruit_BMP085 bmp;

const char* ssid = "SSID";
const char* password = "password";
```

```
  // ntpUDP, time server pool, offset in seconds, u
  NTPClient timeClient(ntpUDP, "0.au.pool.ntp.org"

❶ String webAppURL = "webapp_URL";

  unsigned long lastUpdate = 0;
❷ unsigned long updatePeriod = 20000; // 20 second

  void setup()
  {
    ❸ bmp.begin();
      Serial.begin(115200);
      WiFi.mode(WIFI_STA);
      WiFi.begin(ssid, password);
      Serial.print("Connecting to Wi-Fi ");
      while (WiFi.status() != WL_CONNECTED)
      {
          Serial.print(".");
          delay(500);
      }
      timeClient.begin();
  }

  void loop()
  {
      String dateString, finalAppURL;
      String date, month, year;
```

```
  String hh, mm, ss;
  String temperature, pressure;

❹ if ((millis() - lastUpdate) > updatePeriod)
  {
      if (WiFi.status() == WL_CONNECTED)
      {
          HTTPClient http;
          timeClient.update();

          // Extract date from NTP server:
          dateString = timeClient.getFormattedD
          date = dateString.substring(8, 10);
          month = dateString.substring(5, 7);
          year = dateString.substring(0, 4);

          // Extract time from NTP server:
          hh = String(timeClient.getHours());
          mm = String(timeClient.getMinutes())
          ss = String(timeClient.getSeconds())

          // Add time and date to web app URL:
        ❺ finalAppURL = webAppURL + "?date=" +
          finalAppURL += month + "&year=" + yea
          finalAppURL += "&minute=" + mm + "&s

        ❻ // Get weather data from BMP180:
          temperature = String(bmp.readTempera
```

```
            pressure = String(bmp.readSealevelPr

            // Add weather data to web app URL:
            finalAppURL = finalAppURL + "&tempera
            finalAppURL += temperature + "&airpr

          ❼ http.begin(finalAppURL.c_str());

          ❽ int httpCode = http.GET(); // Get res
            if (httpCode > 0)
            {
                Serial.print("HTTP Response code
                Serial.println(httpCode);
                String httpMessage = http.getStr
                Serial.println(httpMessage);
            } else
            {
                Serial.print("HTTP error: ");
                Serial.println(httpCode);
            }
        http.end(
        } else
        {
            Serial.println("Wi-Fi problem.");
        }
  ❾ lastUpdate = millis();
    }
}
```

The sketch includes and initializes all required libraries, followed by the fields for your Wi-Fi network's name and password and then the initialization for the network time server client ❶.

The sketch then sets the period between sending different data readings to the spreadsheet ❷, in milliseconds. This is the minimum amount of time between updates, but the actual update frequency is determined by the speed of Google's server and your internet connection. For example, consider the values shown in Figure 22-12—even though the demonstration was set to 20 seconds, the actual times recorded were 22 seconds or more. Values of 10 seconds or less may not be recorded, as this doesn't give the processes enough time to complete the reception and saving of data on the Google server.

In `void setup()`, the usual declarations take place, along with starting the required libraries and the Serial Monitor. The sketch connects to your Wi-Fi network and starts the network time client. The code in `void loop()` declares the required string variables used to store the time and temperature data from the network time server and BMP180 sensor, respectively.

The program includes a calculation to determine whether the required amount of time has elapsed before sending more data ❸. This allows the sketch to proceed when the `millis()` count from the last update time is greater than the set period ❹. Once the period has elapsed, it's time to extract the date and time information from the NTP server and store it in the string variables and then append this data to the web app URL ❺.

Next, the temperature and air pressure data are retrieved from the BMP180 sensor and those values are placed into string variables ❻. These variables are then appended to the giant string `finalAppURL`, which already contains the web app URL along with the date and time.

Now that the web app URL has been fully constructed, an HTTP call sends the data to the Google server ❼. The final lines of code get the HTTP response code ❽ from the Google server and send it to the Serial Monitor. This is useful for debugging, since it confirms whether the process has been successful, but you can remove it once your sketch is operating as required. Finally, the sketch records the elapsed time after the latest update, which is used to calculate the delay before the next update ❾.

Tips for Future Google Sheets Projects

Now that you have a working mechanism to log real-world data to a Google Sheets spreadsheet, here are a few final tips for building this type of project. First, you're limited to viewing your projects on a browser, as the spreadsheet is on the Google server. You can also monitor the project's progress from using the Google Sheets app on your mobile device or tablet, as shown in Figure 22-12.

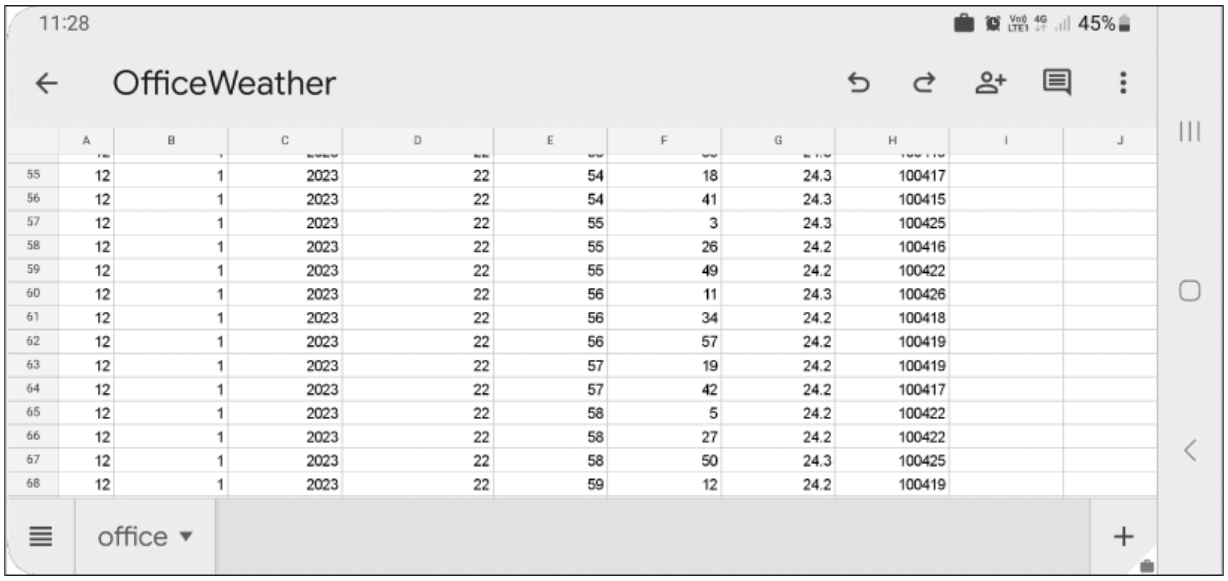


Figure 22-12: Example output from Project #65 displayed using the Google Sheets app on Android

As previously noted, the maximum number of rows in a Google Sheets spreadsheet is 40,000. To monitor two or more ESP32

boards at the same time, give them separate spreadsheets; you can then combine and analyze the data from the desktop once the required data has been captured. If you want to edit an existing project—by changing the variable names or locations in the spreadsheet, for example—don't forget to update the app script and then create a new deployment that uses a new web app URL that you then insert into the sketch.

Moving On

This chapter gave you an inexpensive framework for capturing any type of data that can be gathered by an Arduino-compatible board and sending it straight to a spreadsheet. This allows you to monitor real-time data from almost anywhere in the world and share this data with others in a public or private manner. You can use your own sensors, change the spreadsheet variables, and update times to suit your own needs.

In the next chapter, you'll continue working with internet-connected devices, creating your own web server with the ESP32 board to share information in read-only format for others to view.

23

BUILDING A MINI WEB SERVER



The term *web server* may bring to mind a room full of computer servers connected by dozens of cables. However, any device with an IP address can act as a web server, including your ESP32 board. Thanks to its onboard Wi-Fi connection and software libraries, the board can respond to HTTP requests from web browser clients to display your required data.

In this chapter, you'll create your own mini web server to monitor, in real time, any information gathered or generated by the ESP32 board, allowing anyone with access to the server to review the data captured without any additional software or tools. You'll set up a simple web page using HTML to test and

demonstrate the framework in this chapter and then build a web server to display ESP32 analog and digital input status and then another to display date, time, and temperature information.

Project #66: Creating a Basic Text Web Server

In this project, you'll create a basic text web server. You can use it as a framework to display text or other information built with HTML.

If you haven't already, turn to Chapter 19 and complete the instructions in "Configuring the Arduino IDE for the ESP32" through "Testing the ESP32" to get your ESP32 working. Be sure to follow the instructions to set up port forwarding, if necessary. You should also have completed "BMP180 Sensors" in Chapter 10.

For hardware, you'll need only the ESP32 board and matching USB cable. Once you've connected your ESP32 to your computer, upload the Project #66 sketch, adding your Wi-Fi network and name in the appropriate fields as originally explained in Project #55 in Chapter 19.

Open the Serial Monitor in the Arduino IDE. You should see the network name (SSID) that the ESP32 has connected to, along

with its IP address on your local network. Figure 23-1, shows a connection to IP address 192.168.20.10.

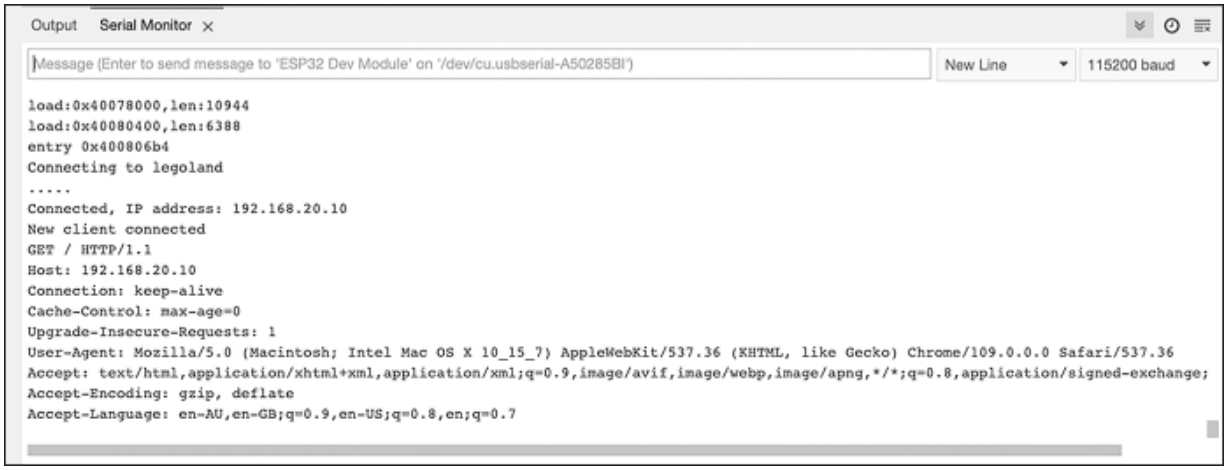


Figure 23-1: The Serial Monitor output for Project #66

Now open a web browser and enter the IP address for your project in the URL field, as shown in Figure 23-2.

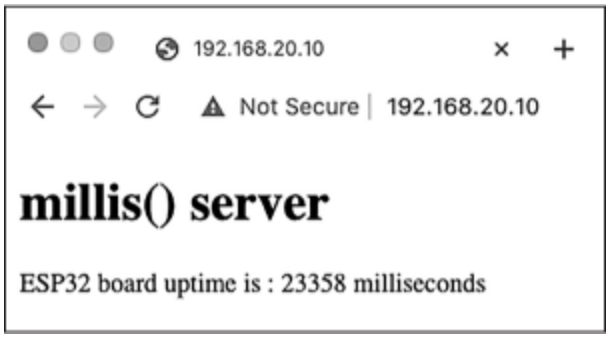


Figure 23-2: A basic web server in action

Let's see how this works:

```
// Project #66 - Basic web page server

#include <WiFi.h>
WiFiServer server(80); // Set web server port to

char* ssid = "name";
char* password = "password";

String request; // Stores HTTP request from clie
unsigned long currentTime = millis();
unsigned long previousTime = 0;
const long timeoutTime = 2000; // Allowed client

void setup()
{
    Serial.begin(115200);
    // Connect to Wi-Fi network:
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    // Show indicator while waiting for connecti
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println(); // Display local IP addres
    Serial.print("Connected, IP address: ");
```

```
    Serial.println(WiFi.localIP());
    // Start web server:
    server.begin();
}

void loop()
{
    String millisValue;
    String textLine = " ";
    // Listen for incoming clients from web brows
    WiFiClient client = server.available(); ❶
    if (client) // If a request received
    {
        currentTime = millis();
        previousTime = currentTime;
        Serial.println("New client connected");
        String currentLine = " ";
        while (client.connected() && (currentTim
        {// Stay connected for timeoutTime
            currentTime = millis();
            if (client.available())
            {
                char c = client.read();
                Serial.write(c);
                request += c;
                if (c == '\n') // Client request
                {
                    if (currentLine.length() == 
```

```
                    {
                        // Send HTTP response ba
                        client.println("HTTP/1.1
                        client.println("Content-
                        client.println("Connecti
                        client.println();

                        // Build web page for di
                        client.println("<!DOCTYP

                        // Now HTML for your pag
                        client.println("<h1>mill
                        millisValue = String(mil
                        textLine += "<p>ESP32 bo
                        textLine += millisValue;
                        textLine += " millisecon

                        client.println(textLine)
                        client.println("</body></
                        // End the response to c
                        client.println();
                        break;
                    } else
                    {// If you received a newlin
                        currentLine = " ";
                    }
                } else if (c != '\r')
                {
```

```
                    currentLine += c;
                }
            }
        }
        request = " ";
        // Close connection to client:
        client.stop(); ❺
        Serial.println("Disconnected.");
    }
}
```

The sketch first includes the Wi-Fi library and initializes an instance of a web server with port number 80. The client computer will use this port number to contact the web server. That is, if you're using a static IP address, you'll enter the IP address followed by `:80` in the web browser's URL field to reach the web server.

The sketch then declares the required variables, including `timeoutTime`, the length of time in milliseconds that a client can stay connected to your web server. If you plan to allow many clients to use the web server, you may want to reduce this number, as the server can deal with only one client request at a time.

In `void setup()`, the code starts the Serial Monitor and then connects to the Wi-Fi network as usual. Once connected, the IP address is displayed in the Serial Monitor and the web server starts. In `void loop()`, the ESP32 waits for an incoming request from an HTTP client (a web browser on a remote device) ❶. When it receives such a request, the sketch records the current value returned by `millis()` to determine how long the client stays connected. As long as the amount of time the client is connected is less than the limit set earlier, your web server receives the client request and sends it to the Serial Monitor for debugging and observational purposes.

When the client request ends, the server sends the following HTTP response in HTML ❷:

```
HTTP/1.1 200 OK
Content-type:text/html
Connection: close
```

To send your web page HTML code back to the client web browser, the sketch uses `client.print()` to send text to the web client and then uses `.println()`to send text followed by a newline character to the client, starting with the page

initialization. You'll need to begin with this page initialization line of HTML every time you form a web page.

Next, the sketch sends the contents of the web page in the form of the following HTML code:

```
<h1>millis() server</h1>
<p>ESP32 board uptime is : millis </p>
```

The sketch replaces *millis* in the HTML with the value of `millis()` by building up a line of text after the header `<h1>` line ❸ into a string and then sending the string with the `client.println(textLine)` function.

The sketch sends the line to end your web page ❹. As with the page initialization code, you'll always use this same line to end a web page in similar projects. Finally, the sketch closes the connection to the client ❺ and the ESP32 awaits a new request.

In the next project, you'll build on the skills you just learned to display data relating to the ESP32's I/O ports on a web page.

Project #67: Creating an ESP32 I/O Port Monitor

This project displays the status of four digital input pins and four analog input pins on your ESP32 board. This gives you the framework for a web server that displays data from devices with analog outputs read by the ESP32's analog inputs, such as light sensors or potentiometers, and data from devices with simple digital outputs, such as motion sensors or door-activated switches. In this case, you'll stimulate these types of sensors with buttons and trimpots.

You will need:

- An ESP32 dev board and matching USB cable
- A solderless breadboard
- Assorted jumper wires
- Four tactile buttons
- Four 10 kΩ, 0.25 W, 1 percent resistors
- Four 10 kΩ breadboard-compatible trimpots

Assemble the circuit as shown in Figure 23-3.

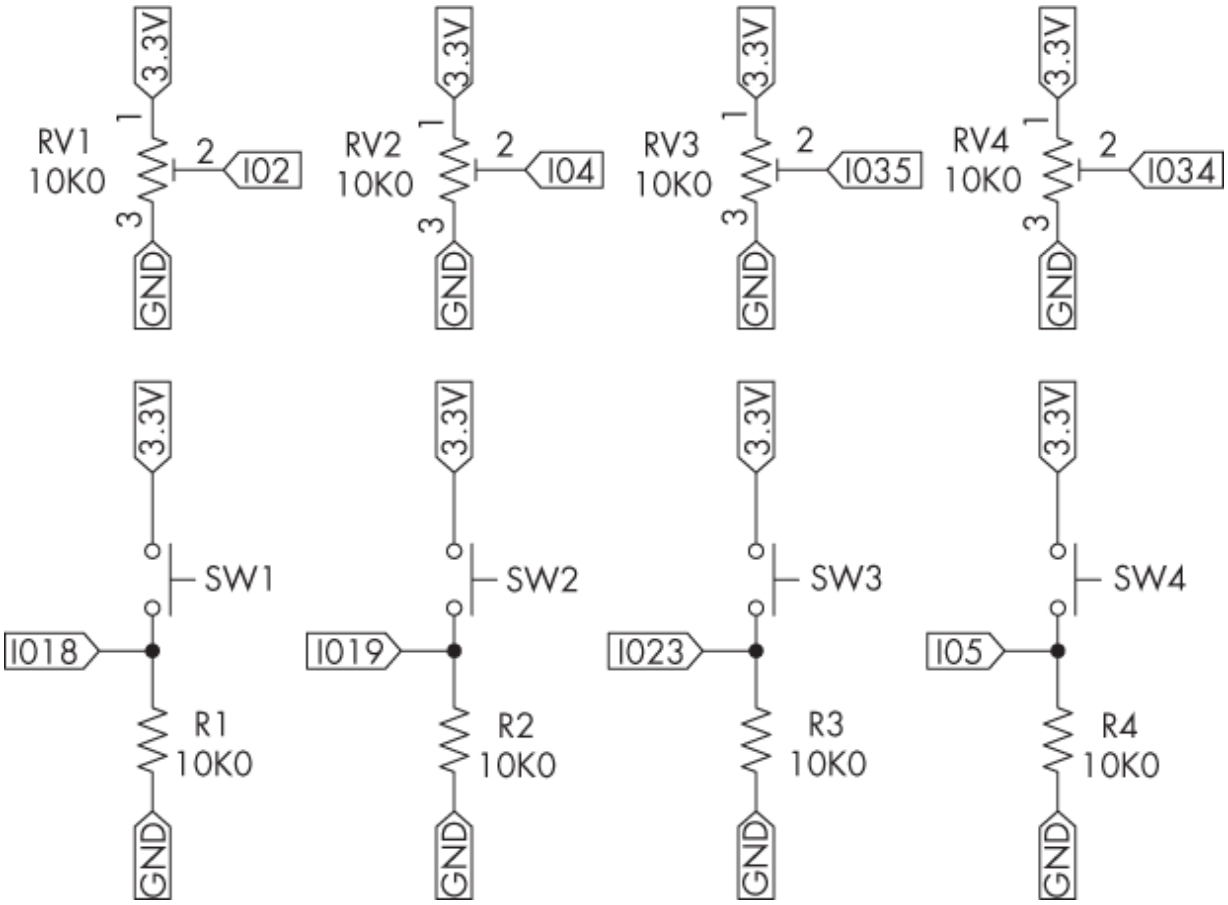


Figure 23-3: The schematic for Project #67

Upload the sketch, adding your Wi-Fi network and name in the fields at the top of the sketch, then open the Serial Monitor in the Arduino IDE. The Serial Monitor should show the network name (SSID) of the Wi-Fi network to which the ESP32 has connected, along with the ESP32's IP address on your local network.

Open this IP address in a web browser using a device on your local network. You should be presented with the status website served up by the ESP32. Adjust the trimpots to random

positions, hold down one or more buttons in the circuit, and then refresh the web browser. The next display should show the values measured by the ADC pins and the status of the four digital inputs, as shown in Figure 23-4.

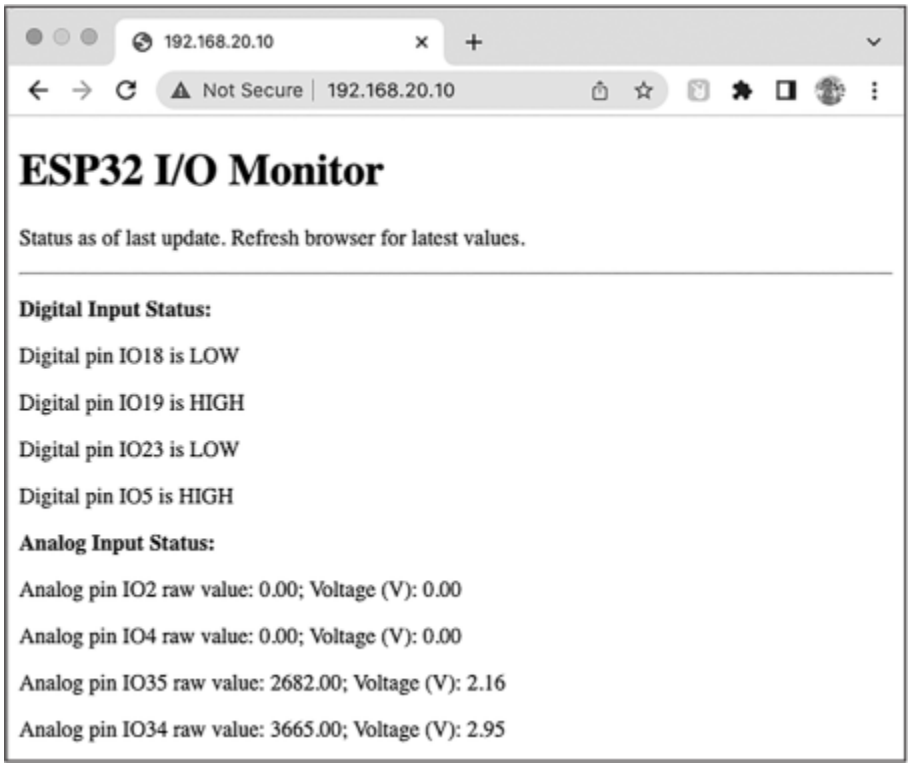


Figure 23-4: Example output from Project #67

Unlike the Arduino Uno and compatible boards, whose ADC has a range of 0 through 1,023 thanks to its 10-bit resolution, the ADC on the ESP32 boards has a range of 0 through 4,095, as it uses 12-bit resolution (2^{12} = 4,096). You can also see the calculated voltage for the ADC inputs in the last item of data for each analog pin's status on the web page.

Let’s see how this works:

```
// Project #67 - ESP32 I/O monitor

#include <WiFi.h>
WiFiServer server(80); // Set web server port to

char* ssid = "SSID";
char* password = "password";

String request; // Stores HTTP request from clie

unsigned long currentTime = millis();
unsigned long previousTime = 0;
const long timeoutTime = 2000; // Allowed client

void setup()
{
    Serial.begin(115200);
    pinMode(18, INPUT);
    pinMode(19, INPUT);
    pinMode(23, INPUT);
    pinMode(5, INPUT);

    // Connect to Wi-Fi network:
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
```

```
    while (WiFi.status() != WL_CONNECTED)
    // Show indicator while waiting for connecti
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println(); // Display local IP addres
    Serial.print("Connected, IP address: ");
    Serial.println(WiFi.localIP());
    // Start web server
    server.begin();
}

void loop()
{
    String textLine = " "; ❶
    int IO18, IO19, IO23, IO5;
    float IO2, IO4, IO35, IO34;
    float IO2A, IO4A, IO35A, IO34A;
    String IO2S, IO4S, IO35S, IO34S;
    // Listen for incoming clients from web brow
    WiFiClient client = server.available();
    if (client) // If a request received
    {
        currentTime = millis();
        previousTime = currentTime;
        Serial.println("New client connected");
        String currentLine = " ";
```

```
        while (client.connected() && (currentTim
        {// Stay connected for timeoutTime:
            currentTime = millis();
            if (client.available())
            {
                char c = client.read();
                Serial.write(c);
                request += c;
                if (c == '\n') // Client request
                {
                    if (currentLine.length() == 
                    {

                        // Send HTTP response ba
                        client.println("HTTP/1.1
                        client.println("Content-
                        client.println("Connecti
                        client.println();

                        // Build web page for di
                        client.println("<!DOCTYP

                        // Now HTML for your pag
                        client.println("<h1>ESP3
                        client.print ("<p> Statu
                        client.println("Refresh 
                        client.println("<hr>");
```

```
IO18 = digitalRead(18); 
IO19 = digitalRead(19);
IO23 = digitalRead(23);
IO5 = digitalRead(5);

IO2 = analogRead(2);
IO4 = analogRead(4);
IO35 = analogRead(35);
IO34 = analogRead(34);

IO2A = ((IO2 / 4096) * 3
IO4A = ((IO4 / 4096) * 3
IO35A = ((IO35 / 4096) *
IO34A = ((IO34 / 4096) *

client.println("<p><b>Di
textLine += "<p>Digital 
if (IO18 == 1)
{
    textLine += "HIGH";
} else
{
  textLine += "LOW</p>";
}
client.println(textLine)
textLine = " ";
textLine += "<p>Digital 
if (IO19 == 1)
```

```
                {
                    textLine += "HIGH";
                } else
                {
                    textLine += "LOW</p>
                }
                client.println(textLine)
                textLine = " ";
                textLine += "<p>Digital
                if (IO23 == 1)
                {
                    textLine += "HIGH";
                } else
                {
                    textLine += "LOW</p>
                }
                client.println(textLine)
                textLine = " ";
                textLine += "<p>Digital
                if (IO5 == 1)
                {
                  textLine += "HIGH";
                } else
                {
                    textLine += "LOW</p>
                }
                client.println(textLine)
                textLine = " ";
```

```
                client.println(" ");
                client.println("<p><b>An
                textLine += "Analog pin
                textLine += String(IO2);
                textLine += "; Voltage (
                client.println(textLine)
                client.println(IO2A);
                client.println("</p>");
                textLine = " ";

                textLine += "Analog pin
                textLine += String(IO4);
                textLine += "; Voltage (
                client.println(textLine)
                client.println(IO4A);
                client.println("</p>");
                textLine = " ";

                textLine += "Analog pin
                textLine += String(IO35)
                textLine += "; Voltage (
                client.println(textLine)
                client.println(IO35A);
                client.println("</p>");
                textLine = " ";

                textLine += "Analog pin
                textLine += String(IO34)
```

```
                        textLine += "; Voltage (V
                        client.println(textLine)
                        client.println(IO34A);
                        client.println("</p>");
                        textLine = " ";
                        client.println("<hr>");

                        client.println("</body></
                        // End the response to cl
                        client.println();
                        break;
                    } else
                    {// If you received a newlin
                        currentLine = " ";
                    }
                } else if (c != '\r')
                {
                    currentLine += c;
                }
            }
        }
        request = " ";
        // Close connection to client:
        client.stop();
        Serial.println("Disconnected.");
    }
}
```

Like the previous project, this sketch serves an HTML web page after receiving a client request. The only differences are the blocks of code required to deal with the I/O values and displaying the results in HTML.

The sketch declares variables ❶ used to store the status of each of the eight inputs—integer variables for the digital inputs, and floating-point variables for the analog inputs. It then uses string variables to store the values from the ADC that it will later combine with HTML for the web page display. The main web page construction starts ❷ with a large header and some text for the user, followed by a horizontal line. The sketch stores the values of the digital inputs (1 for HIGH, 0 for LOW) ❸ and then the analog values of the four ADCs, converting the latter into voltages ❹.

To display the input data on the web page ❺, the sketch tests the status of each digital pin, adding the results as text to the string `textLine`. It displays that text using `client.print()` and repeats the process for the other three digital input pins. Next, it displays the values of each analog input and the matching voltage ❻. It concatenates the required HTML text for each line into the string `textLin`, followed by an HTML `</p>` command to end the paragraph. After the sketch sends the line to end the

web page ❼, the ESP32 closes the connection and awaits a new client request.

Throughout the code, the sketch uses the function `String()` to convert an integer or floating-point value to text to add that value to a larger string variable. This is the key to displaying data generated by the sketch within an HTML web page.

For more practice in displaying sensor data on a web page, you'll display more types of data on the server web page in the next project.

Project #68: Building a Time and Weather Server

This project builds on your existing knowledge to create a web server that displays, on request, the current time, date, temperature, and air pressure at sea level. This is a good example of displaying data from multiple sources on one web page.

You will need:

- An ESP32 dev board and matching USB cable
- A solderless breadboard
- Assorted jumper wires
- A BMP180 temperature and air pressure sensor board

Assemble the circuit as shown in Figure 23-5.

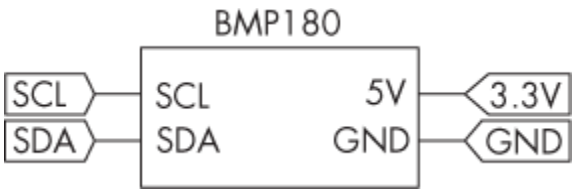


Figure 23-5: The schematic for Project #68

Upload the Project #68 sketch, adding your Wi-Fi network and name in the fields at the top of the code as usual, then open the Serial Monitor in the Arduino IDE. The Serial Monitor should display the network name (SSID) that the ESP32 has connected to, along with its IP address on your local network.

As discussed in “Network Time Protocol” in Chapter 21, you may want to change the network time server IP address and the time zone offset to suit your location.

Open the IP address in a web browser using a device on your local network. You should see the status website served up by the ESP32, as shown in Figure 23-6. Refresh the browser to view the latest information at any given time.

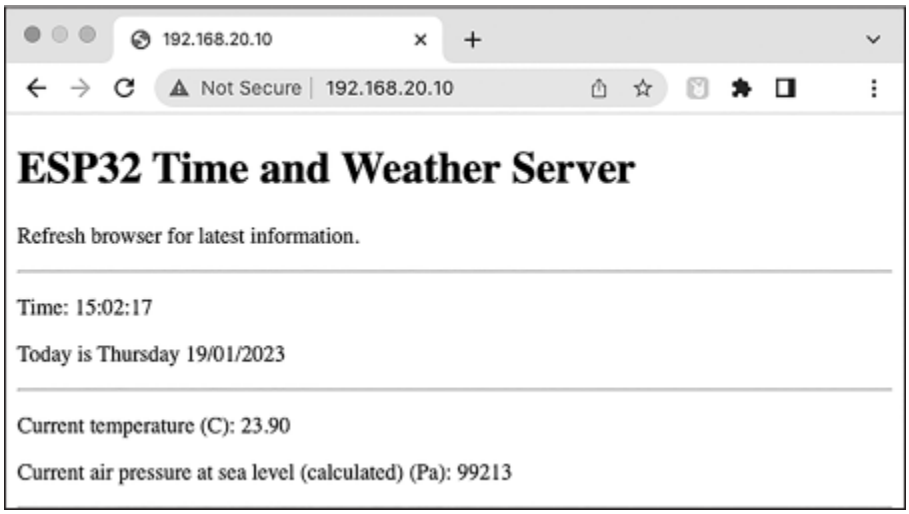


Figure 23-6: Example output from Project #68

Let's see how this works:

```
// Project #68 - ESP32 time and weather server

#include <Adafruit_BMP085.h>
Adafruit_BMP085 bmp;
#include <NTPClient.h>
#include <WiFi.h>
#include <WiFiUdp.h>
WiFiUDP ntpUDP;
WiFiServer server(80); // Set web server port to

char* ssid = "SSID";
char* password = "password";

// ntpUDP, time server pool, offset in seconds,
NTPClient timeClient(ntpUDP, "0.au.pool.ntp.org", 36000
```

```
String request; // Stores HTTP request from clie

unsigned long currentTime = millis();
unsigned long previousTime = 0;
const long timeoutTime = 2000; // Allowed client

void setup()
{
    Serial.begin(115200);
    bmp.begin();
    // Connect to Wi-Fi network
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println(); // Display local IP addres
    Serial.print("Connected, IP address: ");
    Serial.println(WiFi.localIP());
    // Start web server
    server.begin();
}

void loop()
{
```

```
    String dateString;
    String date, month, year;

    // Listen for incoming clients from web brows
    WiFiClient client = server.available();
    if (client) // If a request received
    {
        currentTime = millis();
        previousTime = currentTime;
        Serial.println("New client connected");
        String currentLine = " ";
        while (client.connected() && (currentTim
        {// Stay connected for timeoutTime
            currentTime = millis();
            if (client.available())
            {
                char c = client.read(); // Displa
                Serial.write(c);
                request += c;
                if (c == '\n') // Client request
                {
                    if (currentLine.length() ==
                    {
                        // Send HTTP response ba
                        client.println("HTTP/1.1
                        client.println("Content-
                        client.println("Connecti
                        client.println();
```

```
                    // Build web page for di
                    client.println("<!DOCTYP
                    client.println("<h1>ESP3
                    client.println("<p>Refre
                    client.println("<hr>");
                    client.print("<p>Time: "
                    timeClient.update();
                    client.print(timeClient.
                    client.print("</p>");
                    client.print("<p>Today i
                    switch (timeClient.getDa
                    {
                        case 0: client.print
                        case 1: client.print
                        case 2: client.print
                        case 3: client.print
                        case 4: client.print
                        case 5: client.print
                        case 6: client.print
                        case 7: client.print
                    }
                    // Retrieve and format da
                    dateString = timeClient.
                    date = dateString.substr
                    client.print(date);
                    client.print("/");
                    month = dateString.subst
                    client.print(month);
```

```
                    client.print("/");
                    year = dateString.substr
                    client.print(year);
                    client.print("</p>");
                    client.println("<hr>");
                    client.print("<p>Current
                    client.print(bmp.readTem
                    client.print("</p>");
                    client.print("<p>Current
                    client.print("sea level
                    client.print(bmp.readSea
                    client.print("</p>");
                    client.println("<hr>");
                    client.println("</body></
                    // End the response to cl
                    client.println();
                    break;
                    } else
                        {// If you received a
                            currentLine = "
                    }
                } else if (c != '\r')
                {
                    currentLine += c;
                }
            }
        }
        request = " ";
```

```
        // Close connection to client:
        client.stop();
        Serial.println("Disconnected.");
    }
}
```

The requirements to serve a web page are the same as with the previous project; only the code required for the recorded data and display is different. The sketch includes the required libraries and starts instances of the Wi-Fi server and the Network Time Protocol client.

As in previous projects, the sketch connects to the local-area network Wi-Fi and displays the connection details on the Serial Monitor. When a client request is received, the ESP32 board sends the web server HTML to the client with the first piece of data (the current time) requested from the network time server ❶ and then sends the time to the web page with the `client.print()` function. Next, the sketch retrieves the day of the week ❷, determines the name of the day with a `switch…case` function, and sends all this information to the client.

The sketch stores the full date in the string `dateString` ❸. The date values (day of month, month, and year) are extracted into

individual string variables so the sketch can use `client.print()` to send them as part of the HTML.

The code adds a horizontal line for a neater display ❹ and then retrieves and displays the temperature ❺ and air pressure at sea level ❻. Finally, it completes the web page ❼, and the response to the client request is finished.

Moving On

In this chapter, you learned to create web pages that can display any data you can gather or calculate with devices used alongside an ESP32. From simple weather reports to data from sensors in a laboratory, you can put any of this data on the web for you or others to use.

The next chapter covers a final IoT skill set, demonstrating how to use an ESP32-controlled digital camera for surveillance or fun.

24

THE ESP32 CAMERA BOARD



In the past, to use an Arduino with a digital camera, you'd have to hack into the camera's remote-control system and control it with the Arduino's digital output pins. Today, thanks to the release of the inexpensive ESP32 dev board with a camera (also known as the ESP32 camera board), you can easily control a small digital camera in your projects and save the images for review or streaming over Wi-Fi.

The ESP32 camera board uses the same ESP32 microcontroller with Wi-Fi as found on the ESP32 dev board used in Chapters 19 through 24. However, the camera board is a much more compact unit and includes a connection for a tiny but useful

camera module, which is usually included with purchase of the ESP32 camera board. It may also be the most complex device you've controlled with an Arduino-compatible board in this book so far.

In this chapter, you'll learn to:

- Configure ESP32 camera boards for Arduino
- Build a simple video-streaming device
- Control ESP32 camera boards to capture still images on command

Selecting an ESP32 Camera

This chapter is based on the AI- Thinker ESP32 CAM module, a small board that contains the ESP32 hardware, a microSD memory card socket, a small power regulator, and a connector for a small camera. The package should also include the camera module.

Some ESP32 camera boards, such as PMD Way part 97263257 (shown in Figure 24-1), have no USB interface to save space.



Figure 24-1: An ESP32 camera board

If that's the case for the module you purchase, you'll need a USB-to-serial cable like the PL2303TA cable shown in Figure 24-2 (PMD Way part 727859). You'll use this cable to power the ESP32 camera for uploading sketches and monitoring serial output in the Arduino IDE's Serial Monitor.



Figure 24-2: A USB-to-TTL cable

Alternatively, you can source the ESP32 camera with a matching USB interface board such as PMD Way part 26409000, shown in Figure 24-3.

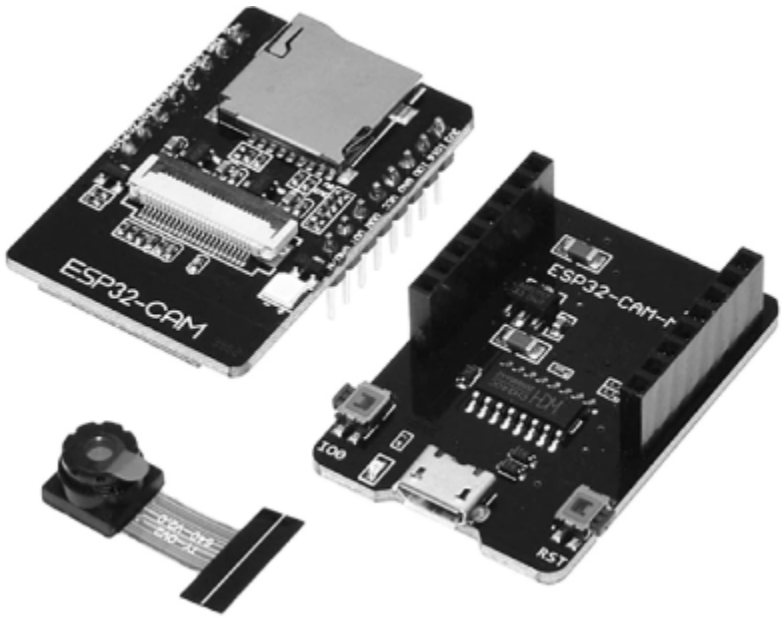


Figure 24-3: An ESP32 camera board with a USB interface board

I chose the ESP32 camera board used as shown in Figure 24-1 not only because it's the most popular model but because it allows you to use an external Wi-Fi antenna to enable a greater operational distance from the Wi-Fi access point. Usually, the module used is shipped configured to use the external antenna. You can still use it without the antenna, but the Wi-Fi range will not be optimal.

After initial experimentation, you may want to substitute your default camera lens with a different lens type for zoom or wide-

angle photography or use a longer cable between the camera and the ESP32 camera board. You can purchase these lenses from PMD Way (see part 26401000, for example) or your ESP32 camera board supplier. However, these extra camera types aren't required for the projects in this chapter.

Setting Up the ESP32 Camera

Once you've purchased your ESP32 camera board, you'll need to set it up to receive sketches and operate properly.

If you haven't already, turn to Chapter 19 and complete the instructions in "Configuring the Arduino IDE for the ESP32" through "Testing the ESP32" to get your ESP32 working. Next, connect the ESP32 camera to your PC. If you have the USB interface board as shown in Figure 24-2, connect via a USB cable as usual. If you're using the alternate USB-to-TTL cable for the USB interface, connect the pins of the ESP32 camera to the cable as shown in Table 24-1.

Table 24-1: Connections Between ESP32 Camera and USB-TTL Cable

ESP32 camera pin	Cable connector

ESP32 camera pin	Cable connector
5V	5V (red)
GND	GND (black)
U0T	RX (white)
U0R	TX (green)

If you're using the USB-to-TTL cable, you must also connect a jumper wire between a GND pin and the IO0 pin before uploading a sketch. After uploading the sketch, you must disconnect and reconnect the USB cable to the PC and then remove the IO0 to GND wire and press the reset button on the board to start the sketch. You'll need to do this every time you upload a sketch.

Once you've connected your ESP32 camera to your PC, open the Arduino IDE and set the board to AI Thinker ESP-32 CAM by selecting **Tools** ▸ **Board** ▸ **esp32**. You may also need to set the USB port to the one used by your USB interface by selecting **Tools** ▸ **Port**.

Open the WiFiScan example sketch included with the Arduino Wi-Fi library. Upload this sketch to your ESP32 camera board, and open the Serial Monitor. You should see the available Wi-Fi networks in your nearby area listed, as shown in Figure 24-4.



Figure 24-4: Example output from running the WiFiScan example sketch

The received signal strength indicator (RSSI) value in Figure 24-4 is a measurement of how well the ESP32 camera board can hear a signal from your Wi-Fi access point. The closer to 0, the better the signal. The CH value is the Wi-Fi channel to your Wi-Fi access point; most access points have 16 channels. Finally, the Encryption type shows which form of encryption protocol is used for the listed Wi-Fi access points. If the WiFiScan example sketch worked correctly, your ESP32 camera board is ready to use.

If your camera didn't arrive preconnected to the board, you'll now need to attach it. Working slowly and carefully, pry open

the black plastic hinged strip across the connector, as shown in Figure 24-5. It may help to use some plastic tweezers instead of your fingers.

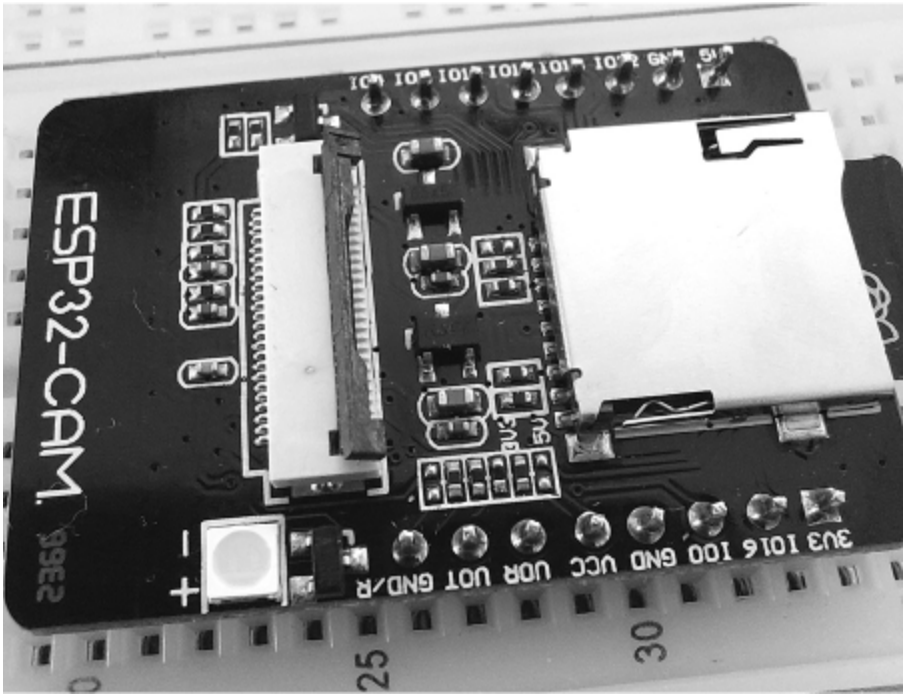


Figure 24-5: Opening the camera cable connector on the ESP32 camera board

Slide the camera's interface cable into the connector on the ESP32 camera board. The camera lens should face upward during this process, with the cable sliding in just a few millimeters. Don't exert too much force; if you run into an obstacle, the cable has gone far enough, as shown in Figure 24-6.

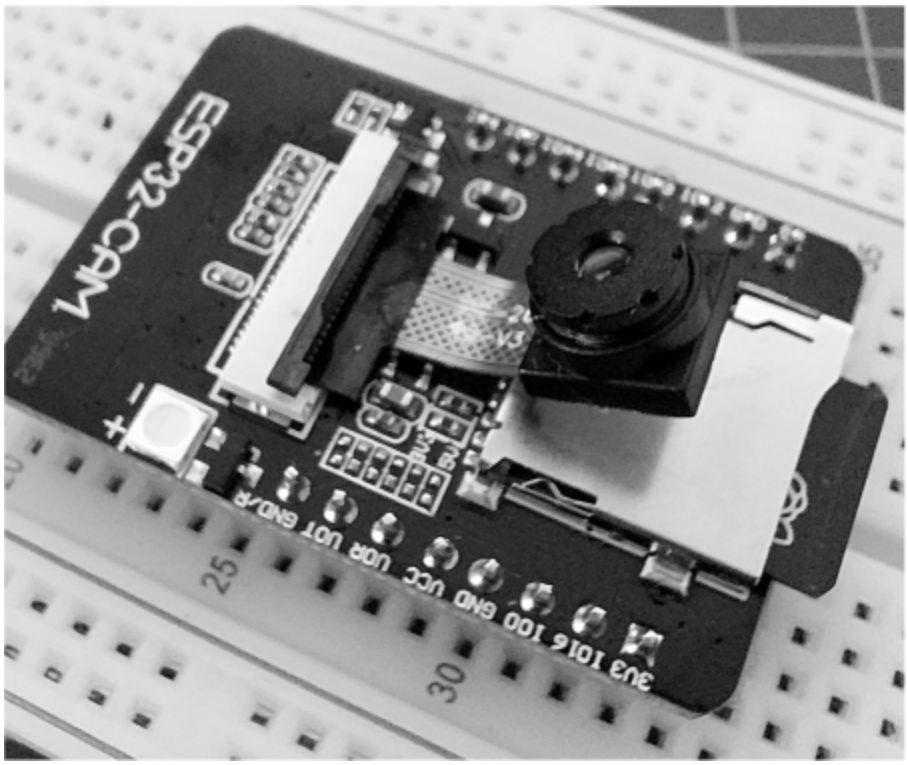


Figure 24-6: Inserting the camera cable connector on the ESP32 camera board

Finally, push the black plastic hinged strip across the connector down until it clicks into place, which also secures the camera connector cable into the ESP32 camera board, as shown in Figure 24-7.

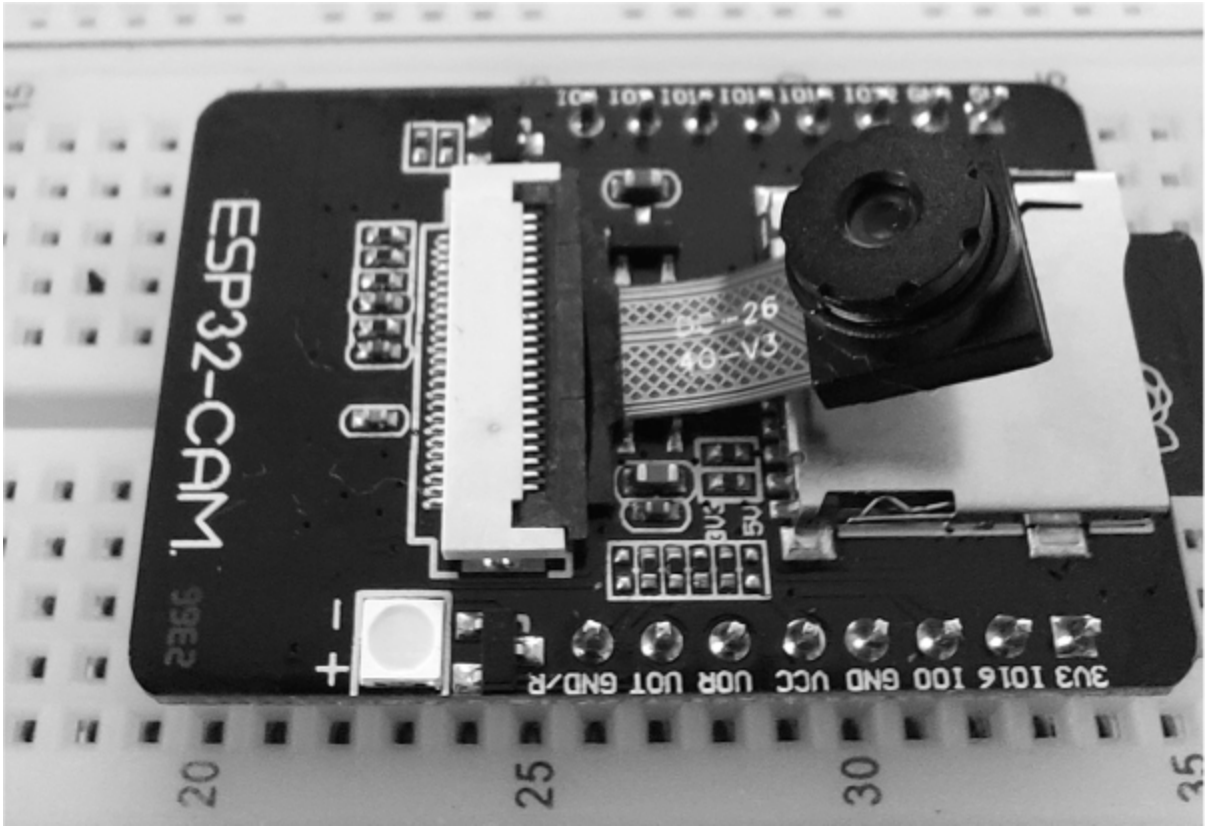


Figure 24-7: The camera installed into the ESP32 camera board

If there is a protective plastic cover over the lens, carefully remove it by pulling the tab on one side of the protector. You're ready to test your camera by using it to stream video across your Wi-Fi network.

Project #69: Streaming Video from a Basic Camera Server

In this project, you'll control the ESP32 camera module remotely over a Wi-Fi local-area network using a web browser

interface, experimenting with various settings on the camera interface to adjust the camera's image or video stream.

For hardware, you'll need just the ESP32 camera module and USB interface as described in the previous section. For the sketch, you'll need to download the project files from the book's website at *https://nostarch.com/arduino-arduinians*, as usual. However, this chapter's projects include extra files beyond the usual *.ino* sketch file. For this project, you only need to open the *project69.ino* file, and the Arduino IDE will automatically open the rest. As shown in Figure 24-8, you should see the included files as tabs under the menu bar.



Figure 24-8: All project file tabs being displayed in Arduino IDE

The additional files (*app_httpd.cpp*, *camera_index.h*, and *camera_pins.h*) include data to define the GPIO pins for the ESP32 camera board and include data to help display the web page for the camera. This project operates a predefined

interface, so you can't modify much of it. You'll only need to concern yourself with the *.ino* sketch file.

In the Project #69 sketch, add your Wi-Fi network and name in the SSID and password fields, as originally explained in Project #55 in Chapter 19. Once you've connected your ESP32 camera to your computer, upload the sketch and then open the Serial Monitor in the Arduino IDE. If your USB interface is via the external cable, remember to remove the wire link from IO0 to GND and then press **RESET** on the camera module.

You should see the IP address that the ESP32 camera is using appear on the Serial Monitor, as shown in Figure 24-9.

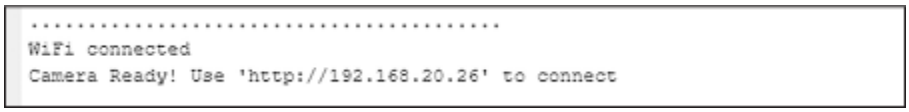


Figure 24-9: Example output from Serial Monitor displaying the camera's IP address

Open a web browser on your PC or mobile device connected to the same Wi-Fi network and visit the IP address shown in the Serial Monitor. The ESP32 camera should serve a web page with various controls and settings. Click **Start Stream** at the bottom of the page. The live video stream from the camera should appear on your screen, as shown in Figure 24-10. (The video in the figure was taken from my kitchen table.)

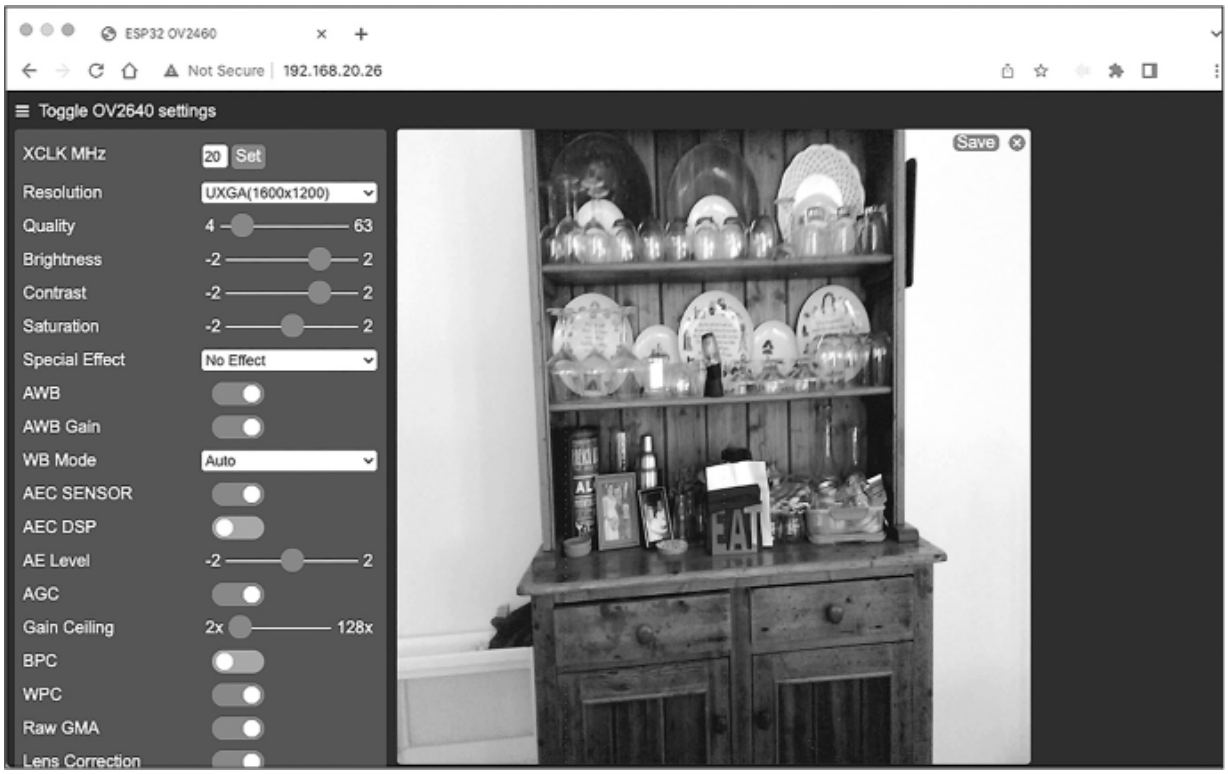


Figure 24-10: Example output from the camera streaming page

Take some time to experiment with the settings, adjusting the brightness, contrast, and so on. When streaming video, the greater the quality of the image, the greater the amount of data being transmitted over the network. If you find the video update speed, or *frame rate*, too slow or jerky, change the resolution using the drop-down menu. You can also switch to streaming in black and white, as this requires much less data transmission than color. If you're curious about the frame rate at the time of operation, you can view it in the Serial Monitor when the camera is streaming, as shown in Figure 24-11.

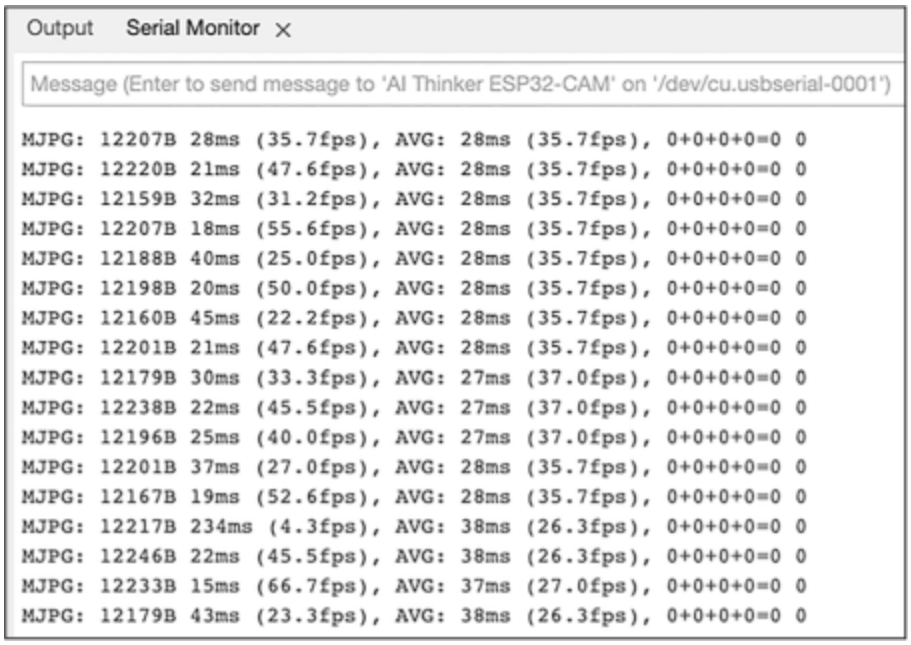


Figure 24-11: Example frame rate output from an ESP32 camera

If you're not happy with the quality or brightness of the video or images, you may need better external lighting or to reposition your camera to allow more ambient light to reach the lens.

Let's take a look at the *.ino* sketch file:

```
// Project #69 - Camera web server

#include "esp_camera.h"
#include <WiFi.h>

#define CAMERA_MODEL_AI_THINKER
#include "camera_pins.h"
```

```
const char* ssid = "SSID";
const char* password = "password";

void startCameraServer();

void setup()
{
  ❶ Serial.begin(115200);
    Serial.setDebugOutput(true);
    Serial.println();

  ❷ camera_config_t config;
    config.ledc_channel = LEDC_CHANNEL_0;
    config.ledc_timer = LEDC_TIMER_0;
    config.pin_d0 = Y2_GPIO_NUM;
    config.pin_d1 = Y3_GPIO_NUM;
    config.pin_d2 = Y4_GPIO_NUM;
    config.pin_d3 = Y5_GPIO_NUM;
    config.pin_d4 = Y6_GPIO_NUM;
    config.pin_d5 = Y7_GPIO_NUM;
    config.pin_d6 = Y8_GPIO_NUM;
    config.pin_d7 = Y9_GPIO_NUM;
    config.pin_xclk = XCLK_GPIO_NUM;
    config.pin_pclk = PCLK_GPIO_NUM;
    config.pin_vsync = VSYNC_GPIO_NUM;
    config.pin_href = HREF_GPIO_NUM;
    config.pin_sscb_sda = SIOD_GPIO_NUM;
```

```
    config.pin_sscb_scl = SIOC_GPIO_NUM;
    config.pin_pwdn = PWDN_GPIO_NUM;
    config.pin_reset = RESET_GPIO_NUM;
    config.xclk_freq_hz = 20000000;
    config.pixel_format = PIXFORMAT_JPEG;
❸   config.frame_size = FRAMESIZE_UXGA;
    config.jpeg_quality = 10;
    config.fb_count = 2;

❹   esp_err_t err = esp_camera_init(&config);
    if (err != ESP_OK)
    {
      ❺ Serial.printf("Camera init failed with er
        return;
    }
❻   s->set_framesize(s, FRAMESIZE_QVGA); // Set c

❼   WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(500);
        Serial.print(".");
    }
    Serial.println(" ");
    Serial.println("WiFi connected");

    startCameraServer();
```

```
  ❽ Serial.print("Camera Ready! Use 'http://");
    Serial.print(WiFi.localIP());
    Serial.println(" ' to connect");
}

void loop() {}
```

The sketch includes the required libraries and then defines the ESP32 camera type, allowing the correct pinout labels to be associated for the Arduino IDE to use. Be sure you've inserted your Wi-Fi network's name and password in the appropriate fields.

In `void setup()`, the code activates the Serial Monitor for debugging ❶. It makes the Arduino GPIO associations for the camera interface pins ❷ and then defines the default image size and quality ❸. If there's a problem initializing the camera, this is articulated ❹. The following `printf()` function is available only when using ESP32-compatible Arduino, not Arduino boards themselves. It allows the display of a variable's value within a string of text. For example, the line at ❺ will display the error code stored in variable `err` at position `%x` in hexadecimal.

The default image frame size is set to Quarter-VGA (320 × 240 pixels, or QVGA) for speed ❻. As the QVGA resolution is smaller, less data is required for each frame, meaning the frame rate can be higher and result in smoother video.

After making the Wi-Fi connection ❼, the sketch displays the ESP32 camera's web page IP address in the Serial Monitor ❽. Have fun experimenting with the camera. There are plenty of ways to use this inexpensive piece of hardware, such as monitoring property entrances, watching children from another room, or viewing a bird's nest. Now is a good time to learn about maximizing the Wi-Fi range with an external antenna.

External Wi-Fi Antenna

The recommended ESP32 camera board in this chapter allows for the use of an external Wi-Fi antenna, which makes a longer distance from the Wi-Fi access point possible. When shopping for an antenna, make sure to find one that includes the cable from the antenna to the ESP32 camera board. The socket on the board is called a *uFL* or *mini ipex connector*. The external antenna and cable should be sold in sets but are usually not

included with the camera, meaning you need to order them separately.

Figure 24-12 shows an example external antenna and cable, PMD Way part 70202023.



Figure 24-12: An external antenna and cable

To check the module antenna configuration, turn the module over so you can see the antenna socket, the tiny round copper ring just below the antenna on the PCB, as shown in Figure 24-13.

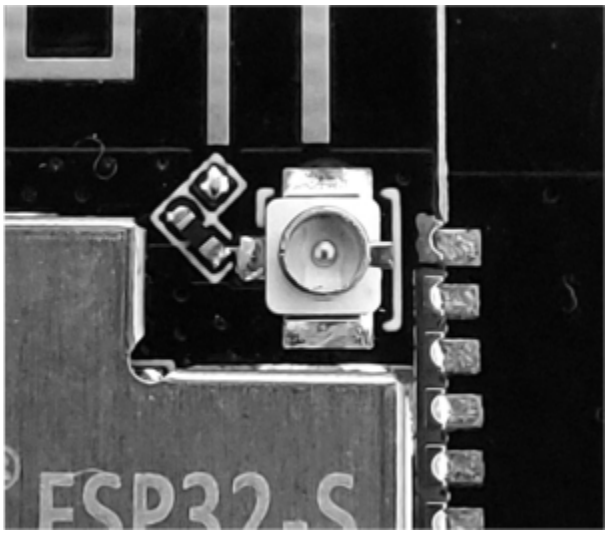


Figure 24-13: An ESP32 camera board antenna socket

On the left of the socket should be three tiny PCB pads—top, bottom, and left. Two of the pads should be bridged by a surface-mount resistor. If the bottom and left pads are bridged in this way, as shown in Figure 24-14, you're ready for an external antenna that can be connected into the socket, as shown in Figure 24-13.

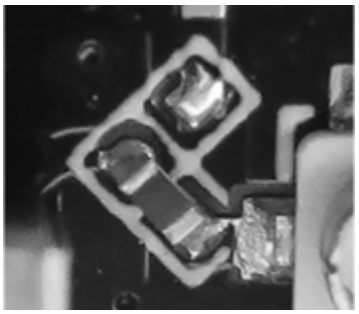


Figure 24-14: ESP32 camera board bridging resistors

However, if your board has the resistor across the top and left pad, you'll need to desolder and resolder it to the correct position. To do this, you'll need a fine-tip soldering iron or a hot-air gun suitable for surface-mount component reflow work, as well as a small amount of solder. If this is a problem for you, check with your board supplier before ordering your ESP32 camera board, if possible.

Using the antenna greatly improves the radio range of the ESP32 camera board. Once you have the external antenna and cable connected, you can easily demonstrate the improvement in radio range by running the WiFiScan demonstration sketch used earlier in the chapter. For example, Figure 24-15 shows the results of the sketch without the antenna and then with the antenna connected.



Figure 24-15: WiFiScan results with and without an external antenna

Without the antenna connected, the output shows an RSSI value of −92; with the antenna connected, however, the RSSI value is −57.

Now that you know how to test Wi-Fi for the ESP32 camera board and set up video streaming, I'll show you how to take photos on command and save them to the microSD card with the time and date stored in the filename of each image for easy reference.

Project #70: Taking Photos and Saving Them to a Memory Card

This project demonstrates how to control the onboard camera to take a photo at your command. You can use the code from this project in your own sketches to create time-lapse cameras, cameras triggered by sensors or switches, or just your own simple portable digital camera.

For hardware, you'll need the ESP32 camera board with matching USB interface and a microSD memory card on which to store the captured images. If you haven't already done so, format the microSD card for use with the camera, setting the filesystem type to FAT32. Any normal format procedure on a PC, Mac, or Linux machine will have the option to set the filesystem type.

Insert the formatted card into your ESP32 camera, then connect the ESP32 camera to your computer and upload the Project #70

sketch. Open the Serial Monitor in the Arduino IDE. If your USB interface is via the external cable, remember to remove the wire link from IO0 to GND and then press **RESET** on the camera module.

The Serial Monitor should display the filename of the images the ESP32 camera has created, as shown in Figure 24-16.

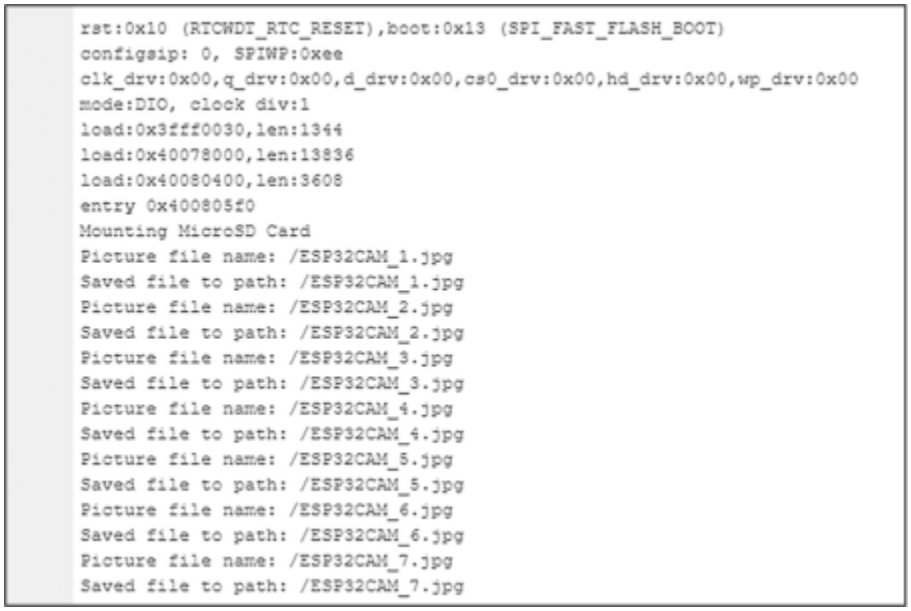


Figure 24-16: Serial Monitor output of image filenames

To stop the camera and review the results, disconnect the USB cable from the computer and insert the microSD card into the computer. Open the file manager for your PC and navigate to the drive representing the microSD card, and you should find the images taken by your ESP32 camera. For example, Figure 24-17 shows a photo of the view from my office.

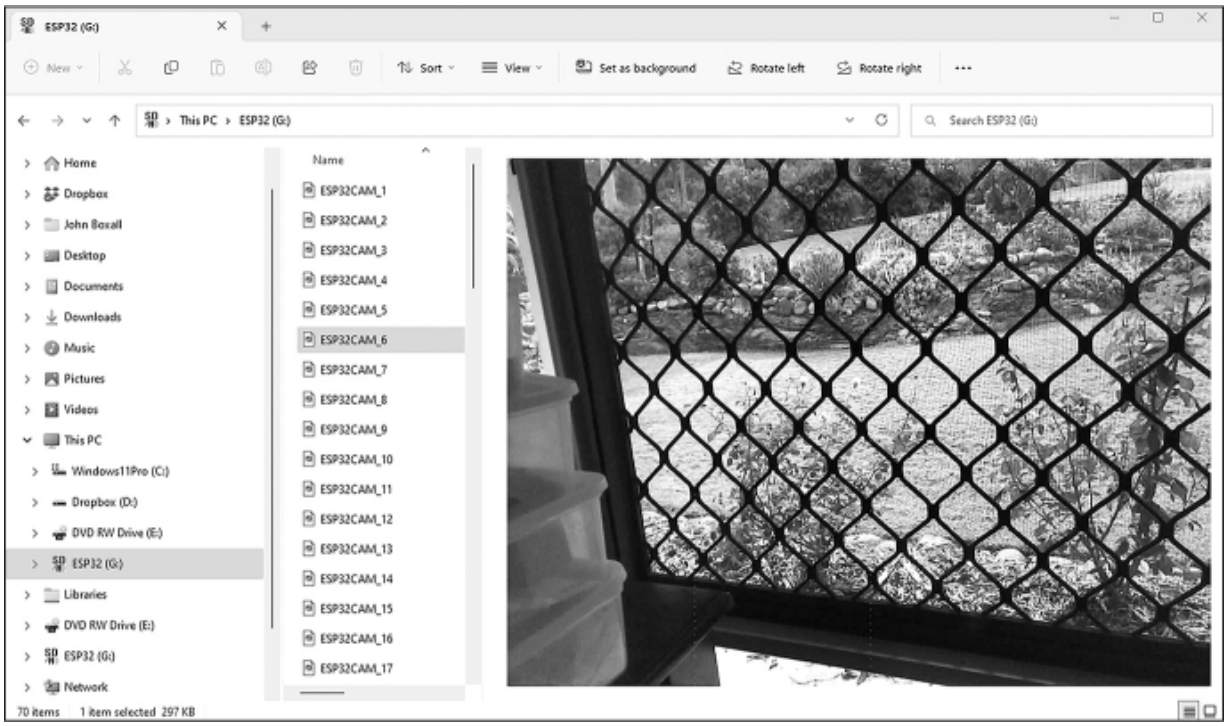


Figure 24-17: Example output from Project #70

Let's see how this works:

```
// Project #70 - Save images to microSD card

#include "esp_camera.h"
#include "soc/soc.h"
#include "soc/rtc_cntl_reg.h"
#include "driver/rtc_io.h"
#include "FS.h"
#include "SD_MMC.h"

// Pin definitions for CAMERA_MODEL_AI_THINKER
#define PWDN_GPIO_NUM 32
```

```
  #define RESET_GPIO_NUM -1
  #define XCLK_GPIO_NUM 0
  #define SIOD_GPIO_NUM 26
  #define SIOC_GPIO_NUM 27
  #define Y9_GPIO_NUM 35
  #define Y8_GPIO_NUM 34
  #define Y7_GPIO_NUM 39
  #define Y6_GPIO_NUM 36
  #define Y5_GPIO_NUM 21
  #define Y4_GPIO_NUM 19
  #define Y3_GPIO_NUM 18
  #define Y2_GPIO_NUM 5
  #define VSYNC_GPIO_NUM 25
  #define HREF_GPIO_NUM 23
  #define PCLK_GPIO_NUM 22

❶ int imageCounter = 0;

  void configESPCamera()
  {
      // Object for camera configuration parameter
      camera_config_t config;

    ❷ config.ledc_channel = LEDC_CHANNEL_0;
      config.ledc_timer = LEDC_TIMER_0;
      config.pin_d0 = Y2_GPIO_NUM;
      config.pin_d1 = Y3_GPIO_NUM;
      config.pin_d2 = Y4_GPIO_NUM;
```

```
    config.pin_d3 = Y5_GPIO_NUM;
    config.pin_d4 = Y6_GPIO_NUM;
    config.pin_d5 = Y7_GPIO_NUM;
    config.pin_d6 = Y8_GPIO_NUM;
    config.pin_d7 = Y9_GPIO_NUM;
    config.pin_xclk = XCLK_GPIO_NUM;
    config.pin_pclk = PCLK_GPIO_NUM;
    config.pin_vsync = VSYNC_GPIO_NUM;
    config.pin_href = HREF_GPIO_NUM;
    config.pin_sscb_sda = SIOD_GPIO_NUM;
    config.pin_sscb_scl = SIOC_GPIO_NUM;
    config.pin_pwdn = PWDN_GPIO_NUM;
    config.pin_reset = RESET_GPIO_NUM;
    config.xclk_freq_hz = 20000000;
    config.pixel_format = PIXFORMAT_JPEG;
    config.frame_size = FRAMESIZE_UXGA;
    config.jpeg_quality = 10;
    config.fb_count = 2;

    esp_err_t err = esp_camera_init(&config);
    if (err != ESP_OK)
    {
        Serial.printf("Camera init failed with e
        return;
    }

    // Set camera quality parameters:
    sensor_t *s = esp_camera_sensor_get();
```

```
❸ // Brightness (-2 to 2)
  s->set_brightness(s, 0);
  // Contrast (-2 to 2)
  s->set_contrast(s, 0);
  // Saturation (-2 to 2)
  s->set_saturation(s, 0);
  // Special effects (0 - none, 1 - negative, 
  // 3 - red tint, 4 - green tint, 5 - blue ti
  s->set_special_effect(s,  0);
  // Auto white balance (0 = disable , 1 = ena
  s->set_whitebal(s, 1);
  // Auto white balance gain (0 = disable , 1 =
  s->set_awb_gain(s, 1);
  // WB MODES (0 - automatic, 1 - sunny, 2 - c
  // 3 - office, 4 - home)
  s->set_wb_mode(s,  0);
  // EXPOSURE CONTROLS (0 = disable , 1 = enab
  s->set_exposure_ctrl(s, 1);
  // AEC2 (0 = disable , 1 = enable)
  s->set_aec2(s, 0);
  // AE LEVELS (-2 to 2)
  s->set_ae_level(s, 0);
  // AEC VALUES (0 to 1200)
  s->set_aec_value(s, 300);
  // GAIN CONTROLS (0 = disable , 1 = enable)
  s->set_gain_ctrl(s, 1);
  // AGC GAIN (0 to 30)
```

```
    s->set_agc_gain(s, 0);
    // GAIN CEILING (0 to 6)
    s->set_gainceiling(s, (gainceiling_t)0);
    // BPC (0 = disable , 1 = enable)
    s->set_bpc(s, 0);
    // WPC (0 = disable , 1 = enable)
    s->set_wpc(s, 1);
    // RAW GMA (0 = disable , 1 = enable)
    s->set_raw_gma(s, 1);
    // LENC (0 = disable , 1 = enable)
    s->set_lenc(s, 1);
    // HORIZ MIRROR (0 = disable , 1 = enable)
    s->set_hmirror(s, 0);
    // VERT FLIP (0 = disable , 1 = enable)
    s->set_vflip(s, 0);
    // DCW (0 = disable , 1 = enable)
    s->set_dcw(s, 1);
    // COLOR  BAR PATTERN (0 = disable , 1 = ena
    s->set_colorbar(s, 0);
}

void initMicroSDCard()
{
    if (!SD_MMC.begin())
    {
        Serial.println("microSD card failed");
        return;
    }
```

```
    uint8_t cardType = SD_MMC.cardType();
    if (cardType == CARD_NONE)
    {
        Serial.println("microSD card not found")
        return;
    }
}

void takeNewPhoto(String path)
{
    // Set up frame buffer:
    camera_fb_t *fb = esp_camera_fb_get();

    if (!fb)
    {
        Serial.println("Camera capture failed");
        return;
    }

    // Save picture to microSD card:
    fs::FS &fs = SD_MMC;
    File file = fs.open(path.c_str(), FILE_WRITE
    if (!file)
    {
        Serial.println("Couldn't open file in wr
    } else
    {
        file.write(fb->buf, fb->len);
```

```
        Serial.printf("File saved as: %s\n", path
    }
    file.close();
    esp_camera_fb_return(fb);
}

void captureImage()
{
    imageCounter += 1;
    // Build image path and filename:
    String path = "/ESP32CAM_" + String(imageCou
    Serial.printf("Picture file name: %s\n", path
    takeNewPhoto(path);
}

void setup()
{
    Serial.begin(115200);
    WRITE_PERI_REG(RTC_CNTL_BROWN_OUT_REG, 0);
    configESPCamera();
    initMicroSDCard();
}

void loop()
{
    captureImage();
    delay(5000);
}
```

The sketch includes all the required libraries and then gives the pin definitions for the AI-Thinker ESP32 camera board. Every time the camera takes a photo, the program tracks the number of photos using the `imageCounter` integer variable ❶, which will become part of the image's filename (as demonstrated in Figure 24-17).

The sketch also defines the pin numbers used for connection to the camera ❷, saves the image type, and defines the image size and quality from `config.pixel_format`. This is set to the *.jpg* file type, an image size of UXGA (1,600 × 1,200 pixels), and a maximum quality of 10 (the higher the number, the lower the image quality). Other camera settings ❸ include white balance control, brightness, contrast, and so on. The comments in the sketch above each of these parameters describe their value ranges, so you can experiment with the values to get the desired result for your camera installation.

The `initMicroSDCard()` function contains the initialization for the microSD card circuitry, while the `takeNewPhoto()`function enables the camera to take and save an image; its parameter is the filename with which to store the image. The crux of the sketch is `captureImage()`, which the program calls to capture images. This function increments the `imageCounter` variable and

inserts it into a string for the image filename beginning with */ESP32CAM_* (which you can change to meet your needs) and adds the *.jpg* file extension to the end. Note that if you reset the ESP32 board before retrieving images from the memory card, they will be overwritten.

In `void setup()`, the sketch initializes the Serial Monitor, followed by the `WRITE_PERI_REG()` function that turns off brown-out detection for the camera. This function allows the camera to keep operating (though at a lower quality) if the supply voltage drops temporarily. The sketch initializes the camera and the microSD card circuitry. Finally, it captures a new image by calling `captureImage()`, followed by an arbitrary five-second delay.

You can build your own digital photography projects around the sketch for this project, which contains everything you need to control the camera—simply adjust the camera settings to your liking and call the `captureImage()` function when you need a photo taken. The process of capturing an image and saving it to the microSD card takes around two seconds, though, so you can't force it to work any faster than that.

Now that you can take your own photos, it's time to learn about the pinouts on your ESP32 camera board so you can interact

with external devices to expand your projects.

The ESP32 Camera Pinouts

As with the ESP32 dev board, the ESP32 camera board has a range of GPIO pins you can use for noncamera operations. Once you understand these pinouts, you can use them in your own projects as needed. Table 24-2 shows the pin labels and their matching Arduino use.

Table 24-2: ESP32 Camera Pinouts

Pin label	Pin use	Note
5V	5 V power input	n/a
GND	GND	n/a
IO12	GPIO pin	n/a
IO13	GPIO pin	n/a
IO15	GPIO pin	n/a

Pin label	Pin use	Note
IO14	GPIO pin	n/a
IO2	GPIO pin	n/a
IO4	GPIO pin	Controls onboard bright LED
3V3	3.3 V power input	Preferable to use 5 V
IO16	GPIO pin	n/a
IO0	Code uploading/GPIO pin	Connect to GND to upload code
GND	GND	n/a
VCC	Power output	Output, not input!
UOR	Serial RX (receive)	n/a

Pin label	Pin use	Note
UOT	Serial TX (transmit)	n/a
GND	GND	n/a

If you’re using the camera, it’s preferable to power the ESP32 camera with 5 V; otherwise, 3.3 V is fine. Running at the lower voltage can cause a slight image degradation. I recommend a 5 V DC with at least 500 mA of current from an external power supply when building your own projects that include more hardware than just the board.

The bright white “flash” LED is connected to GPIO pin IO4 and can be turned on and off like any digital output pin. It’s also connected to the microSD card socket circuit and will turn on when the card is being accessed. This LED generates a lot of heat, so when controlling it for your own projects, don’t run it continuously for longer than one second before allowing time for it to cool off.

There is a small red LED on the other side of the camera module that is connected internally as GPIO pin 33. It's wired inversely so that setting that pin LOW turns the LED on and HIGH turns it off. Overall, the GPIO pins can be used as inputs or outputs. Remember that the logic voltage is 3.3 V, not 5 V.

Moving On

In this chapter, you learned to harness the inexpensive and useful ESP32 camera board for video streaming or taking photos and saving them to a memory card. This adds a final skill set to the six ESP32-based chapters, which have demonstrated various ways of controlling projects remotely and logging and displaying data.

This marks the final chapter of *Arduino for Arduinians*! To continue your Arduino journey, check out the epilogue for some next steps.

AFTERWORD



At this point, having read about (and I hope built) the 70 projects in this book, you should have a much deeper and broader understanding of electronics and the Arduino environment, along with the knowledge and confidence required to create more complex and useful Arduino-based projects.

You've learned to interface with a wide variety of devices outside the Arduino ecosystem via data buses and different forms of wireless communication, including Bluetooth and the Telegram instant-messaging platform. With your newfound skills, you might build devices that control mains outlets via the internet, create web interfaces to monitor sensor data, add MP3-quality audio to your projects to generate user notifications, use watchdog timers to improve project operation reliability, and much more.

I hope this book has inspired you to keep exploring the world of electrical engineering. You won't be alone in doing so. There's a thriving community of Arduino users you can find on the internet in places like the following:

The official Arduino forum *https://forum.arduino.cc*

The hackaday.io website *https://hackaday.io/list/3611-arduino-projects*

Instructables *https://www.instructables.com/circuits/arduino/projects*

Reddit *https://www.reddit.com/r/arduino*

You can also search for the #arduino hashtag on social media for project ideas and inspiration or seek out a local hackerspace or club, either online or in person.

I'm happy to receive feedback about this book via the contact details on the publisher's web page: *https://nostarch.com/contactus*. While you're there, don't forget to check out my other books published by No Starch Press, including *AVR Workshop*, which takes you into the world of 8-bit microcontrollers that power the Arduino ecosystem.

As always, don't just sit there—make something!

APPENDIX

COMPRESSING AND INSTALLING ARDUINO LIBRARIES



Once you've created a new Arduino library, you can store and distribute it by saving the library files as a ZIP file. Future users who obtain the ZIP file can then easily install the library.

In this appendix, you'll learn how to make a ZIP file in Windows, macOS, and Linux as part of the process of collating Arduino library files for installation. You'll then learn how to install an Arduino library received as a ZIP file into the Arduino IDE.

Compressing Files into a ZIP File

In this section, you'll learn how to compress custom Arduino library files into a ZIP file, which not only enables easy distribution but also puts them in the required format for fast

installation into the Arduino IDE. Skip to the appropriate section for your OS.

This section is based on the example Arduino library called *blinko*. However, in your own projects, you'd substitute this with your own library name.

Windows 7 and Newer

To create a ZIP file for an Arduino library with Windows, first place the custom Arduino library files you want to compress, along with the example sketch (stored in its own folder, as are all sketches), in one location. Figure A-1 shows an example of the library files together.

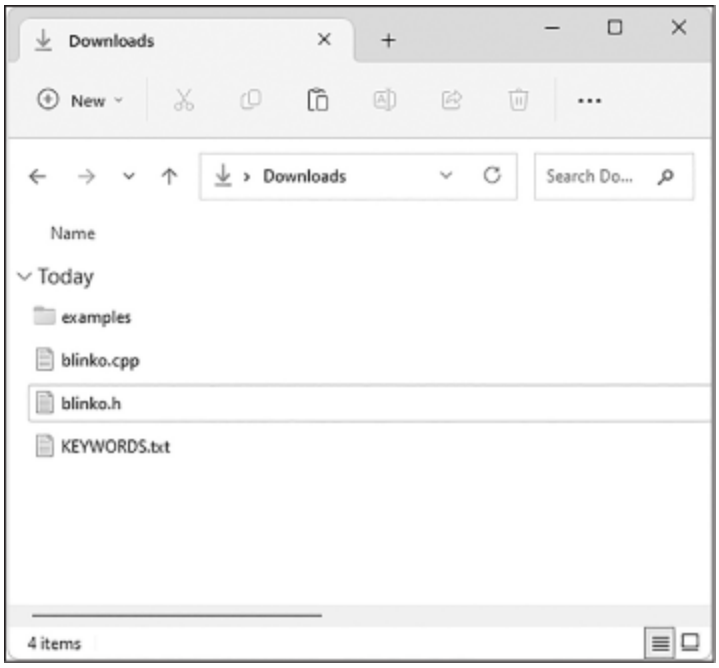


Figure A-1: Arduino library files in the Downloads folder

Select all the files, right-click the highlighted files, and select **Compress to Zip file**, as shown in Figure A-2.

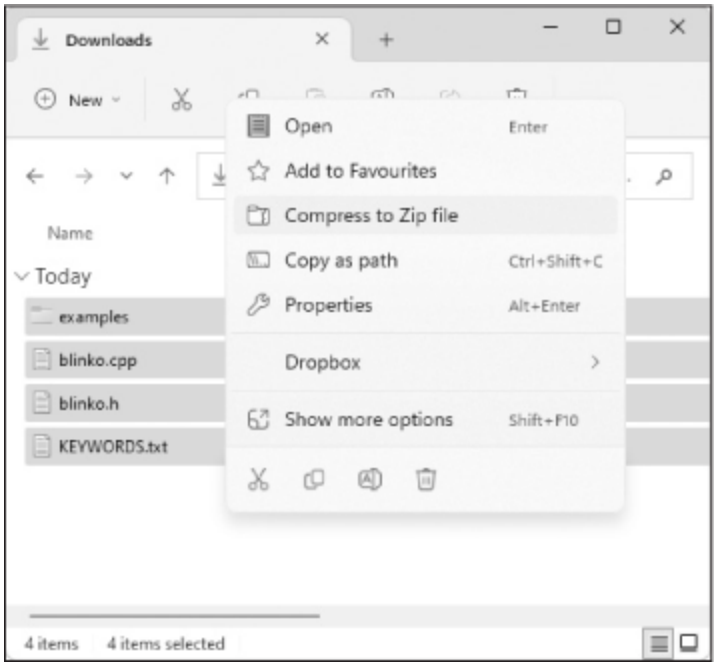


Figure A-2: Compressing the library files

A new file will appear in the folder, with name editing enabled. Change the library name to *blinko.zip*, then press ENTER, as shown in Figure A-3.

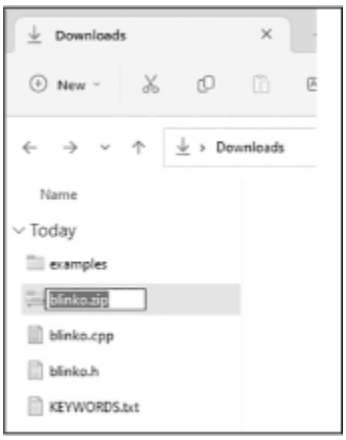


Figure A-3: Changing the name of the library ZIP file

You now have a library ZIP file that you can easily distribute to others or install yourself.

macOS 10 or Newer

To create a ZIP file with macOS, first place the custom Arduino library files you want to compress, along with the example sketch (stored in its own folder, as are all sketches), in one location. Figure A-4 shows an example of the library files together.

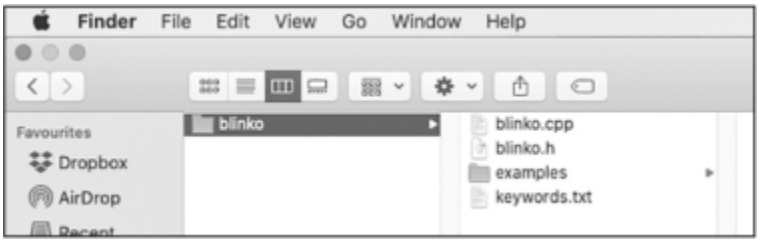


Figure A-4: The Arduino library files in the Downloads folder

Select all the files, right-click the files, and select **Compress 4 Items**, as shown in Figure A-5.

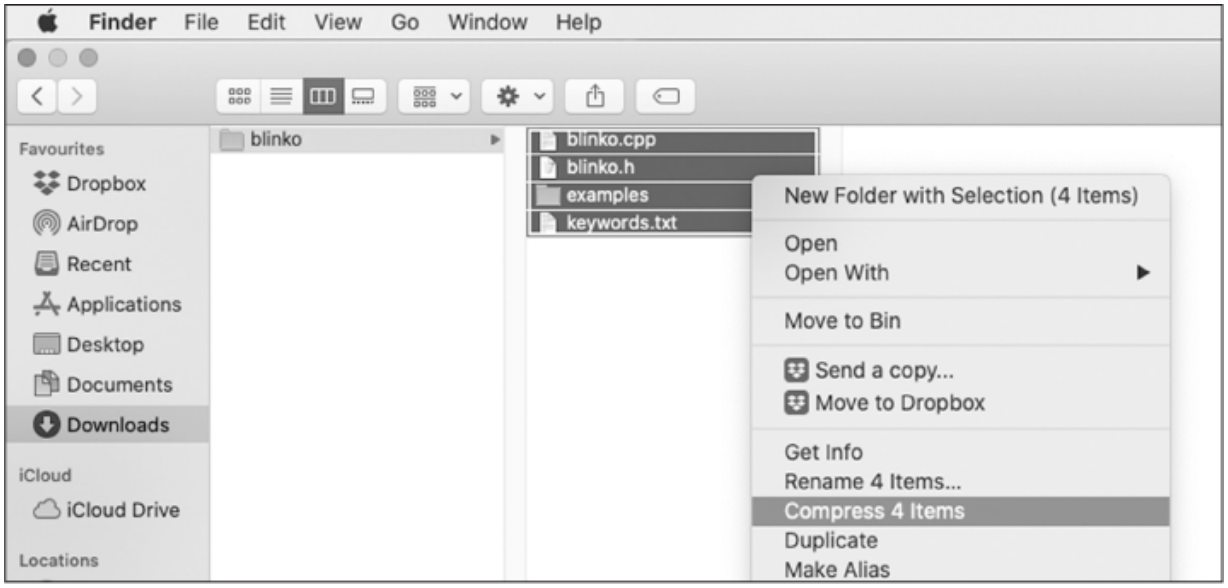


Figure A-5: Compressing the library files

A new file called *Archive.zip* will appear in the folder, as shown in Figure A-6.

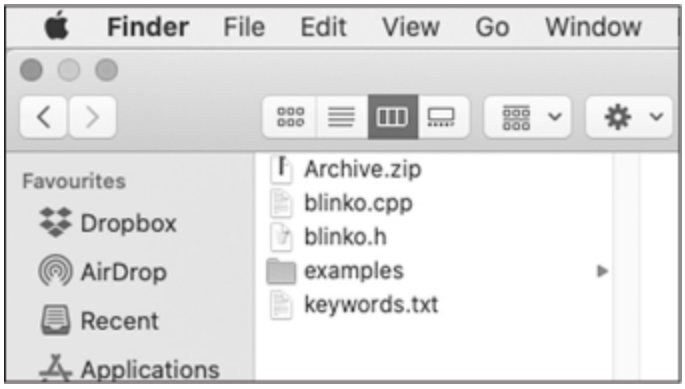


Figure A-6: The Archive.zip *file in the Downloads folder*

Click on the *Archive.zip* folder and change the name to *blinko.zip*, as shown in Figure A-7.

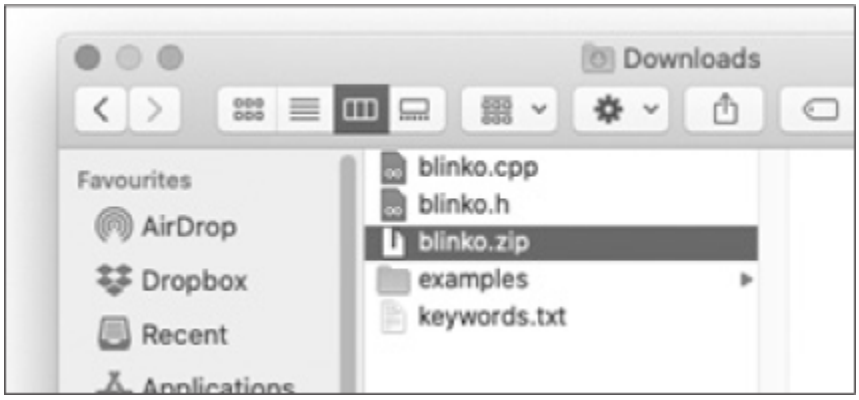


Figure A-7: Renaming Archive.zip *to* blinko.zip

You now have a library ZIP file that you can easily distribute to others or install yourself.

Ubuntu 20.04 LTS and Newer

To create a ZIP file with Ubuntu Linux, first place the custom Arduino library files you want to compress, along with the example sketch (stored in its own folder, as are all sketches), in one location. Figure A-8 shows an example of the library files together.

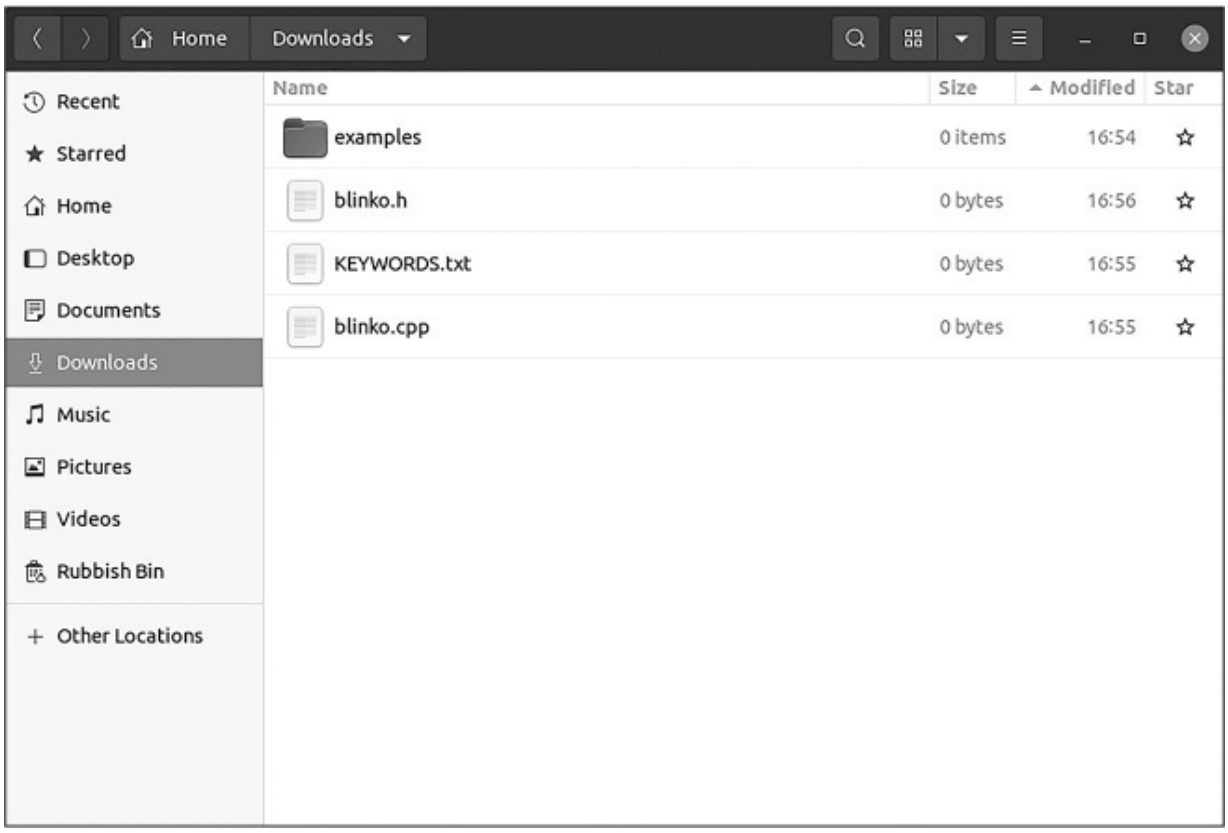


Figure A-8: Blinko Arduino library files

Select all the files, right-click the files, and select **Compress**, as shown in Figure A-9.

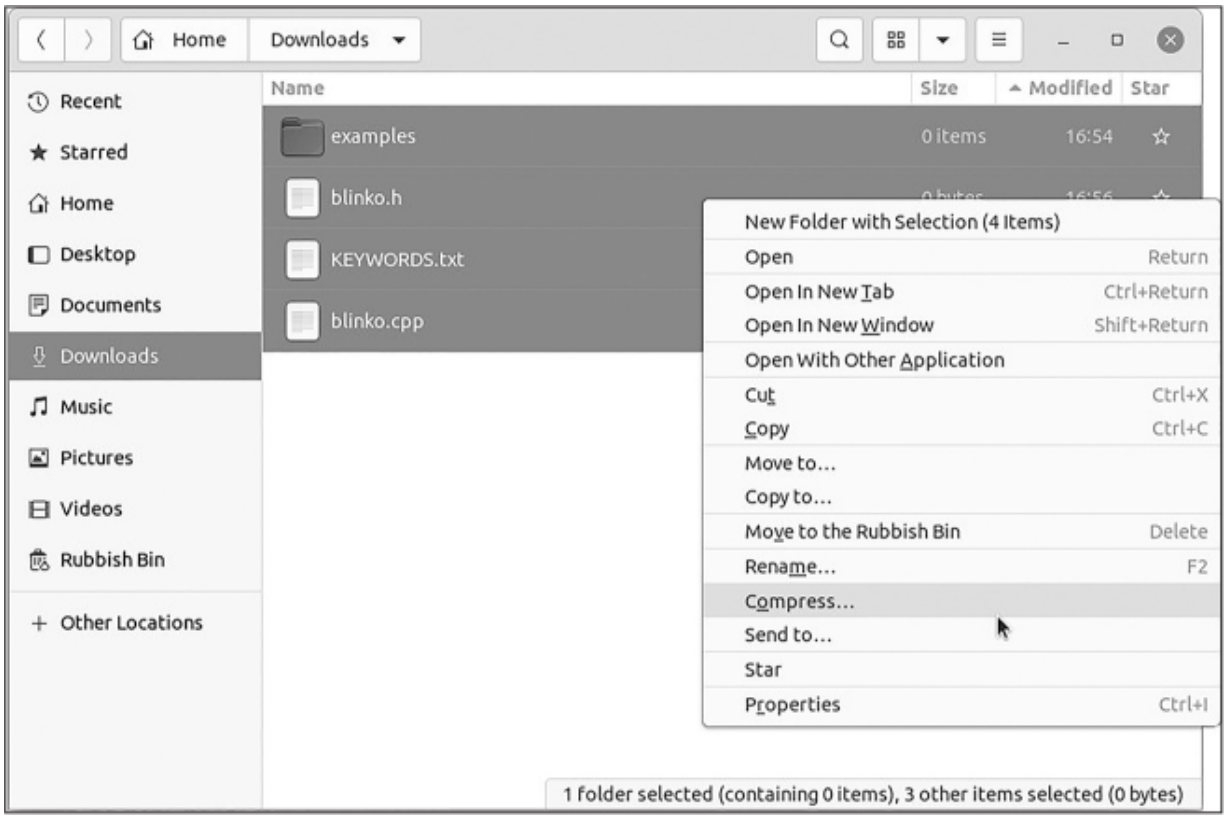


Figure A-9: Compressing the library files

A new window will appear, prompting you for an archive name. Enter *blinko* for the name and click the .zip radio button, and then click **Create**, as shown in Figure A-10.

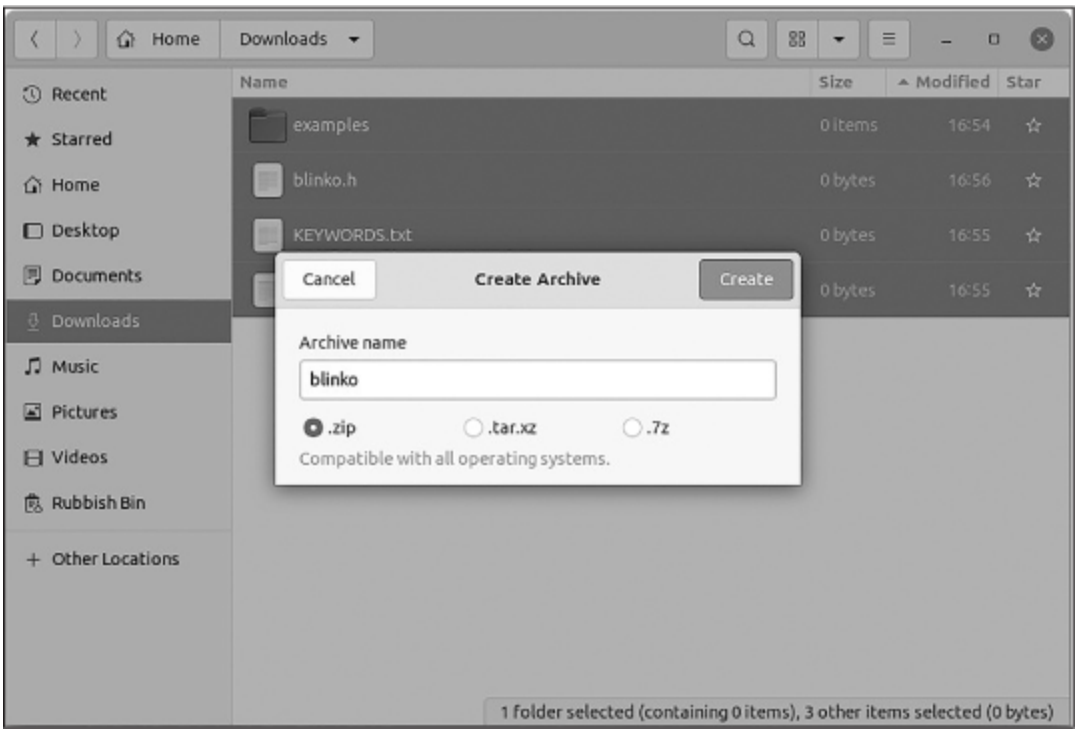


Figure A-10: Setting the ZIP filename

Once the compression is complete, you will find your new ZIP file in the same folder along with the library files. You now have a library ZIP file that you can easily distribute to others or install yourself.

Installing Your New Library

You can now install your library using the ZIP file method. Open the Arduino IDE and select **Sketch ➧ Include Library ▸ Add .ZIP Library**. A file dialog will open. Navigate to your ZIP file and select it for installation.

Once the library has been installed and you've restarted the Arduino IDE, you will see your library listed when you select **Sketch ▸ Include Library**, as shown in Figure A-11.

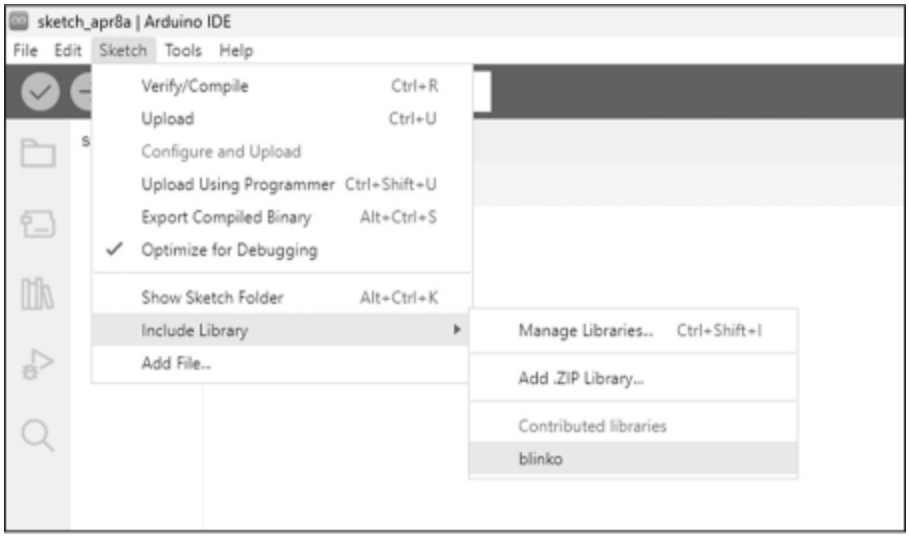


Figure A-11: An Arduino library, now available in the IDE

You can now easily access the sketch you have installed, such as the *blinko* library, by selecting **File ▸ Examples ▸ blinko**, as shown in Figure A-12.

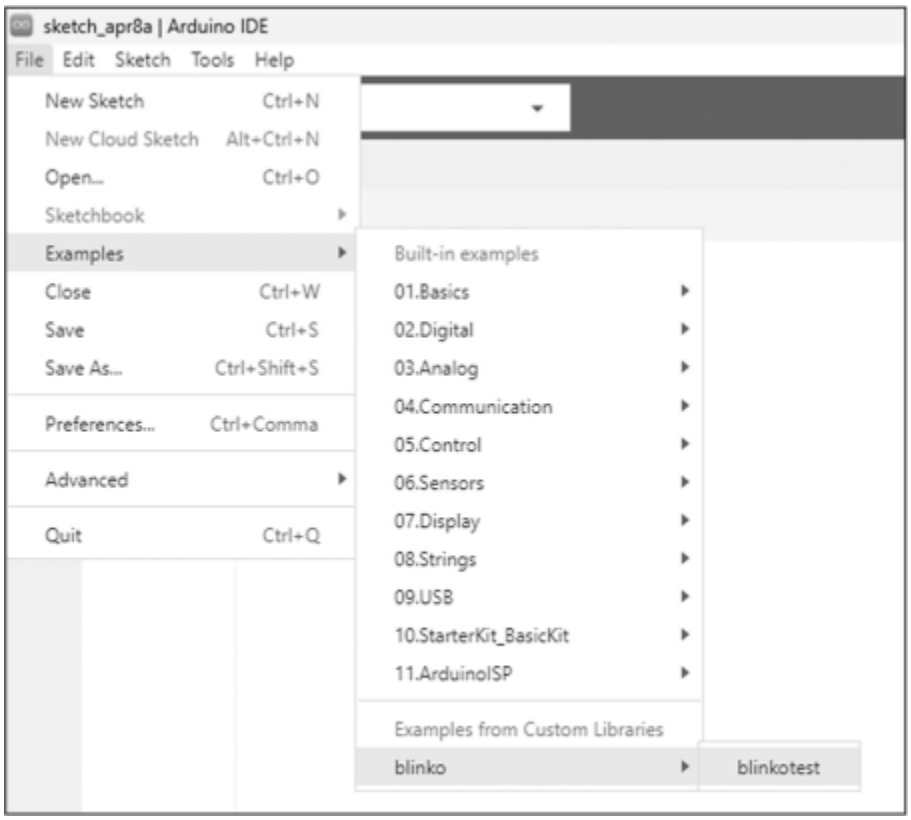


Figure A-12: The Arduino IDE's library example sketch menu showing the blinko example sketch

INDEX

A

rnating current), 250
ets, 93
controlling with Arduino, 100–102
optocouplers and, 94–95
remote control, 95–96
SMS remote control, 105–109
timer-controlled, 102–105
transmitter boards, hacking, 96–100
oints, 327–328
parameter, 256
nal Boards Manager URLs dialog, 33–34, 324–325
sure display
ting, 154–156
mple output, 155
s for, 154–156
ker ESP32 CAM module, 414
narlie, 58
ing current. *See* AC
s or amps (A), 250
`tone()` function, 7

ad() function, 38
o-digital converter (ADC) pins, 2
ite() function, 38
. app for Bluetooth control
ks, editing, 242–243
ons
dding, 239
arranging, 239–240
renaming, 241
ting, 236–245
nload button and QR code for, 243
properties, 237
lding, 244
l properties, editing, 238
ng and building, 243–244
interface, building, 238
. devices
tooth pairing with, 228
al I/O control with, 234–236
ling data to, 230
IDE. *See also* Serial Monitor
itional Boards Manager URLs dialog, 33–34, 324–325
ny85 microcontroller with, 33–35
rd manager selection, 325

iguring, 355
or ESP32, 324–326
ISP, 34–35
-to-Arduino communication
link, 309–312
example output, 310
ibraries, 312
arts for, 309
RS232, 298–301
assembly, 298–299
connection diagrams, 299
example output, 299
arts for, 298
-to-PC connections
link, 306–309
example output, 307
arts for, 306
ng, 294–295
Uno
ny85 pin references vs., 37
32 vs., 323–324
, 323–324
s, 14
er consumption, 252–253

n, 275
des, 163
timer, 49
ockets, 252
328P-PU, 31
diagram, 32
for, 274–275
ifications, 33
32U4 microcontroller, 159
terrupt() function, 264
5 microcontroller
uino functions for, 38
Arduino IDE, 33–35
uino Uno vs., 37
ega328P-PU, compared with, 32–33
tiplexing with, 65–67
k speed, changing, 41–42
external crystal subcircuit, 42–43
pins, 37
lator speed, 41–42
diagram, 32
gramming shield, 43–45
t buttons, adding, 37–38
ifications, 33

5 programming shield
ting, 43–45
ıe header pins, 44
s for, 43
for, 44
ting Data Transmission (project), 365–367

B

banks, 253–254
life, 253–255
coded decimal switches, 27
n-wheel, 27
ing, 27–29
ry, 27
ıumbers, displaying, 19–20, 113–115
555 timer IC, 77–78
ing, 21
; LED wave, creating, 20–21
() function, 315, 319
ditor, 241–242
:h
roid app for, creating, 236–245
uring data via, 231–233

ligital I/O control, 234–236
l converter modules, 226–228
ules, 226
name, updating, 245–246
ing
with Android devices, 228
with macOS devices, 227
with Windows devices, 227
updating, 245–246
ling data via, 228–231
to Android devices, 230–231
to macOS devices, 230
parts for, 228
to Windows devices, 228
minal HC-05 app, 230
, 225
51
sensor, 152–154, 231, 298, 311, 365
Manager, 34
arameter, 256
ler, 41–42
er, 352–353
ction, 355
`Message()` function, 361

ard, Arduino, 42
a 30-LED Matrix Display (project), 67–74
an Event Logger (project), 86–90
a Giant Digital Clock (project), 380–383
the “Hello, World” of Hardware (project), 35–37
a Hosted Web Page for a User Interface (project), 343–348
a Minimalist 5 V Arduino Circuit (project), 272–276
an MP3 Player Sound Board (project), 140–142
a PC-Controlled Relay Board (project), 116–122
a Simple MP3 Player (project), 137–140
a Text Capture Device (project), 218–223
a Time and Weather Server (project), 401, 407–411
a USB Shortcut Keyboard (project), 168–173
a Watchdog Timer (project), 50–55

C

, 277–279
lware, 279–280
ıry, 280
ocol, 278
.TXT file, 285
`mage()` function, 428
ıg Data for Real-Time Monitoring (project), 231–233

network, 105
USB-to-serial interface, 325
plexing, 58
ATtiny85 (project), 65–67
larger displays, 62
Six LEDs (project), 59–62
12 LEDs (project), 62–65
LED demonstration, 58–59
353–354
ce variable, 288
`rint()` function, 407, 411
`rintln()` function, 332–333, 339, 401
ee sketches
ssing files
acOS, 435–436
buntu LTS, 434–438
Windows, 434–435
ches, 251–252
ring a Project via USB Flash Drive (project), 197–200
er area network bus. *See* CAN bus
ing the Mains Outlet with SMS (project), 105–109
ing the Mains Outlet with a Timer (project), 102–105
ing Seven-Segment LED Displays (project), 21–24
ing the Transmitter Board (project), 100–102

ing Two or More Secondary Arduino Boards (project), 315–319
PS() function, 142
m software, 119, 229, 230, 294–295
ated Universal Time, 370
; an Arduino-to-Arduino Data Link (project), 309–312
; an Arduino-to-PC Data Link (project), 306–309
; an ATtiny85 Programming Shield (project), 43–45
; a Basic Text Web Server (project), 398–401
; a Blinking LED Wave (project), 20–21
; an Electronic Die (project), 24–25
; an ESP32 I/O Port Monitor (project), 401–407
; a Giant Seven-Segment LED Display (project), 125–132
; a Low-Power Periodic Data Logger (project), 260–263
; a PC Random Painter (project), 176–177
; a PC-to-Arduino Remote Control (project), 295–298
; a Quick-Read Thermometer (project), 39–41
; an RGB LED Tester (project), 214–218
; a Soft On/Off Switch (project), 78–81
; a Temperature and Air Pressure Display (project), 154–156
; a TPIC6B595 Binary Number Display (project), 113–115
oscillator circuit, 42–43
, 333–334
character displays, making, 73–74

D

- ending to file, 183–187
- ction register, 14
- s, 306–312
- itoring, 281–284
- erical, reading, 192–195
- eval, 362–365, 371–374
- ling via Bluetooth, 228–231
- smission, automating, 365–367
- ing to USB flash drives, 183–187
- ging, 165–168, 189–192, 386–395
- uit schematic for, 260
- ting, 260–263
- s for, 260
- ng string, 411
- `ng.substring()` function, 374
- ct current), 250
- ta direction register), 14
- ction, 25
- function, 38, 72
- `roseconds()` function, 38
- `terrupt()` function, 264

ment boards, 323
'O control, 234–236
rolling from Android phone, 234–235
rolling from PC, 235
s for, 234
ins, 15
ead() function, 25, 27, 38
torage oscilloscope, 17
rite() function, 38, 58, 61, 348
urrent (DC), 250
igits() function, 72
ng Binary Numbers (project), 19–20
ng the Time and Date on an OLED (project), 374–377
ng Two Time Zones on an OLED (project), 378–380
ads. *See* online resources
real-time clock IC, 84, 102
real-time clock IC, 84–86, 102
real-time clock library, 258–260
ime package, 112
les, 341–342, 343–344
3

E

I, 122, 200
ie die, 24–25
g Arduino-to-Arduino Communication (project), 298–301
me, 370
amera board
ma socket, 422
ching the camera, 416–417
ging resistors, 422
e connections, 415–417
O pins, 428–429
view, 414
ng photos to memory card, 423–429
cting, 414–415
ng up, 415–417
aming video, 418–421
debugging, 421
example output, 419
frame rate, 410
frame size, 421
ino sketch file, 410, 420–421
IP address, 418, 421
parts for, 418
project files, downloading, 418
ng photos, 423–429

Fi antenna, external, 421–423
iScan example sketch, 415–416
evelopment boards, 323
O port monitor, 401–407
nple output, 403
s for, 402
icrocontroller, 322–324
ino Uno vs., 323–324
iguring Arduino IDE, 324–326
ription, 322
, 323–324
GPIO pins, 326–327
forwarding, 328
ng, 326–328
Fi connectivity, 327–328
me and weather server, 407–411
nple output, 408
ldress, 408
s for, 407
gger, building, 86–90
l erystal subcircuit, 42–43

F

lesystem, 180
nd() function, 189
er IC, 49, 52–53, 77–78, 81–82
able, 77–78
429
ives
ending data to file, 187–189
uino library installation, 181
rface module, 180–182
ing sensor data to, 189–192
eet configuration via, 198–200
ing numerical data, 192–195
ing text from, 201–203
rmatting, 179
rity key, 196–197
cting and preparing, 180
ing and displaying integers on, 194
ing data, 183–187
r, 184
voltage, 125
tocoupler, 94–95
ics, 275
cy counter, 17

G

t() function, 217
reenwich Mean Time), 370
, 79, 106, 250, 274, 275, 429
ıccount, 237
Apps Script, 388–391
Play, 230
Sheets, 166, 385
.es and data headings, 387
aring document, 386–388
Workspace, 385
ns, 328–340
32 camera board, 428–429
32 development boards, 326–327
e-width modulation, 340–343
ote-controlling, 329–332, 334–340
dule, 288
OLED displays, 149–153, 153
ıino library installation, 151
selection, 142
alization, 142
ıe header pins, 150
D-to-Arduino connections, 151

(GND) pin, 79, 106, 250, 274, 275, 429

H

y.io (website), 432
lex communication, 305
nction, 223
re programmer, 34–35
luetooth module, 226
iguring, 245
address, 246
ating name, 245–246
ype crystal, through-hole, 42
0-compatible LCD, 209
at, 48–49
t() function, 54–55
pedance (aka input state or high-Z state), 58
veb page
ding, 343–348
cycles, 343–344
ing LED on and off, 343
interface, 343–344
pping, 145
328

:, 333, 339–340
urces, 334
sponse, 401

I

address scanner
s for, 144
9548A breakout board for, 146–149
g, 144–146
and PS/2 keyboard
necting, 211
mple output, 213
ries, 214
ng, 211–214
58–354
t serial programmer (ICSP) pins, 270–272
ng Digital Output Pin Speed (project), 16–19
`oSDCard()` function, 428
eader pins, 44
ch file, 420–421
tput (I/O) pins, 1, 37, 58
ate (aka high impedance state or high-Z state), 58
oltage, 2

ables (website), 432
ed circuits (ICs), 1
of Things (IoT), 321
`tFunction()` function, 264–265
ot trigger, 266–269
uit schematic for, 267
s for, 267
rol, 234–236
1, 37, 58
monitor, ESP32, 401–407
mple output, 403
s for, 402
ernet of Things), 321
ess, 334, 398, 402, 408, 418, 421
N-MOSFET, 77

K

Your PC Awake Automatically (project), 174–175
d, PS/2
necting to LCD, 211–214
mple output, 208
ary installation, 207
i-DIN format in, 206

3574 LCD modules, 209–211
et breakout board, 206
ng, 208, 211–214
ng diagram for, 207
demulation
rating, 160
uino Keyboard library, 160, 162
mple output, 161–162
Leonardo, 160
ing data with, 165–168
ifiers, 162–164
sing and releasing one or more keys, 164–165
ial keys, 162–164
ping, 160–161
`.end()` function, 161
`.press(x)` function, 164
`.print()` function, 160
`.println()` function, 160
`.release(x)` function, 164–165
README.txt file, 10

L

75

shift register, 112
n() function, 211
r() function, 211
() function, 211
acklight() function, 211
eld, 209
P clock
mbly, 380–381
ding, 380–383
nection layout, 381
mple display, 381
s for, 380
•
king, 20–21
mma-style, 124–125
m-segment displays, 21–24
er, 214–218
ing on and off, 343
lo, 159
mega32U4 microcontroller in, 159
oing PC awake with, 174–175
board emulation with, 160–162
se emulation with, 173–175
cting in IDE, 161

nverter modules, 226–228
ing with Android devices, 228
ing with macOS devices, 227
ing with Windows devices, 227
;
uino-to-Arduino data link, 312
bus library, 280, 287
76, 181
231 RTC library, 84
LCD and PS/2 keyboard, 214
alling, 438–439
client library, 370–371
-style PC keyboard, 207
capture device, 223
2 library, 152
versalTelegramBot, 355
flash drive installation, 181
keyboard emulation, 160, 162
Fi, 332
Manager, 151, 153, 181, 207, 209, 256, 258, 355
itting diodes. *See* LEDs
() function, 200
() function, 262
() function, 90

Data with USB Keyboard Emulation (project), 165–168
Interrupt Triggers (project), 266–269
OBDII Car Data (project), 285–288
Sensor Data (project), 189–192
Time and Temperature Data (project), 386–395
ole, 64, 69
e, 58

M

dress, 246
devices
tooth pairing with, 227
ling data to, 230
() function, 104
) function, 104
e sockets, 252
integrated, 58
oger, 119, 229–230, 294
r cards
natting, 423
ng photos to, 423–429
xide semiconductor field-effect transistors (MOSFETs), 76–77, 80–81, 83–84

ip MCP2515 CAN controller IC, 279
ip MCP2551 CAN transceiver IC, 279
ip Technology, 159
function, 38
) cards, 220
natting, 423
ng photos to, 423–429
ft
re, 351
el, 166, 233
t, 176–177
ps (mA), 250, 255
function, 38, 332, 361, 394
x connector (aka uFL), 422
b server, 397
c text web server, 398–401
32 I/O port monitor, 401–407
32 time and weather server, 407–411
) Inventor, 237–243
r keys, 162–164
name, updating, 245–246
ing Engine Data (project), 281–284
's, 76–77, 80–81, 83–84
gin() function, 173–174

ick() function, 175
mulation, 173–174
d() function, 173–174
ve(x,y,z) function, 173–174
ess() function, 175
lease() function, 175
yer
ding, 137–140
mands, 136–137
ıd board, 140–142
circuit, 135
, 133
300 MP3 module, 147
buttons with one analog output, 1
ttons, 3–4
ttons, 5–7
uttons, 7–11
xing, 58

N

ine character), 332
k Time Protocol (NTP), 370
ıt library, 370–371

ks, 374–383
character (\n), 332
metal hydride (NiMH) cells, 254
eries, avoiding, 255
ET, 52, 53, 80
28
y-open (NO) contacts, 82–83
Network Time Protocol
t timeClient() function, 373
ssages parameter, 361

O

On-Board Diagnostic II), 278–280
data, 285–288
II-to-shield cable, 279
lays, 149–153
Arduino library installation, 151
ont selection, 142
nitialization, 142
nline header pins, 150
OLED-to-Arduino connections, 151
-zone NTP clock

assembly, 378
building, 378–380
example display, 378
parts for, 378
clock
building, 374–377
connecting to Wi-Fi, 375
example output, 375
parts for, 374
d Diagnostic II. *See* OBDII
esources
roject file, 244
uino forum, 432
uino language reference, 38
I codes, 163
ny package, 34
era project files, 418
bus library, 280
40 drivers, 325
Term software, 119, 229–230, 294
231 RTC library, 84
32 package, 325
s, 152
caday.io, 432

ructables, 432
App Inventor, 237
omin 6x5 font, 73
P, 328
client library, 370
servers, 370
II, 280
on, 325
lit, 432
ch download, 50, 70
gram, 352
Sine, 105
offsets, 370
i signal strength, 328
plat, 141
g current, 125
pers, 94
onstration circuit, 95
3, 94
matic symbol, 94
r
racy, 42
de, 41–42
cope, 17

ogic devices, 83–84
oltage, calculating, 2

P

oat() function, 312
infrared (PIR) motion sensor, 82
8
-to-Arduino connections, 128
olled relay board, 116–122
LCD modules, 209–211
ecting to Arduino, 209
CD, 309, 312
functions, 211
ry, 209
ng, 210
om painter
ting, 176–177
nple results, 176
duino remote control, 295–298
mbly, 295
nple output, 296
s for, 295
data logger, 260–263

) function, 14–15, 38, 61
function, 348
function, 348
function, 348
-15
og-to-digital converter pins, 2
D12 pins, list of, 275
in, 429
pin, 79, 106, 250, 274, 275, 429
e header pins, 44, 150
ins, 1, 37, 58
ls, 275
pin, 274
0 pin, 274
d, 16–19
pin, 429
1 pin, 274
pin, 429
pin, 429
pin, 429
pin, 79, 106, 128, 324
-style LEDs, 124–125
matic symbol for, 125
A USB-to-serial cable, 414

rogrammable logic controllers), 289
mmands, 19, 20
gister, 200
–15
nders, 1
arding, 328
ipulation, 13
linking LED wave, creating, 20–21
caution in, 15
controlling seven-segment LED displays with, 21–23
displaying binary numbers, 19–20
electronic die, 24–25
ncreasing digital output pin speed, 16–17
/O pins, 37
in speed test, 18
reading BCD switches, 27–29
reading multiple digital inputs simultaneously, 24–25
itor, 401–407
50
ly voltage, 125
-sourced, 251
d, 251–252
onsumption
ino, 252–253

ery life estimation, 253–255
lware-based reduction, 269–270
suring, 251–252
ware-based reduction, 255–260, 263–266
own/Wake Periodic mode, 256–257, 274
g Off the Arduino Automatically (project), 81–82
device, 143
function, 229
circuit boards. *See* PCBs
)function, 229
essages() function, 361
mable logic controllers (PLCs), 289
ming shield, 43–45
e PC keyboard
mple output, 208
ry installation, 207
i-DIN format in, 206
3574 LCD modules, 209–211
et breakout board, 206
ng, 208, 211–214
ng diagram for, 207
)function, 38
idth modulation, 76–77, 214, 218, 340–343
ote ESP32 control, 335, 343–348. *See also under* remote control

ıeel BCD switches, 27
ɜe pulse-width modulation
de() function, 348

Q

s, 243–244
ad thermometer
ting, 39–41
s for, 39

R

numbers, 132
painter, 176–177
t() function, 217
BCD Switches (project), 27–29
input, 3–7, 24–25
Multiple Digital Inputs Simultaneously (project), 24–25
Six Buttons with One Analog Input (project), 5–7
Three Buttons with One Analog Input (project), 3–5
) function, 197
ad() function, 11
sure() function, 154

erature() function, 154
() function, 202–203
eclock, 84–86, 223, 288
307 real-time clock IC, 84, 102
231 real-time clock IC, 84–86, 102
231 real-time clock library, 258–260
l signal strength indicator (RSSI), 416, 423
s, 14–15
(int a) function, 122
int a) function, 122
control, 295–298, 312–315, 328–340, 355–361
utlets, 95–96
mands, sending, 356
32 PWM control, 335, 343–348
duty cycles, 343–344
urning LED on and off, 343
user interface, 343–344
sages, retrieving, 361
remote control, 105–109
Telegram, 351–367
Arduino IDE, configuring, 355
automating data transmission via, 365–367
configuring account, 352–354
of four LEDs, 355–361

retrieving remote data, 362–365
UniversalTelegramBot library, 355
Controlling Four GPIO Pins (project), 334–340
Controlling Four LEDs (project), 355–361
Controlling a Single GPIO Pin (project), 328–334
Control Operation (project), 312–315
delay variable, 284, 288
divider formula, 5
ng Remote Data (project), 362–365
ng the Time and Date (project), 371–374
D tester
ting, 214–218
s for, 214–215
M and time values, 218
CD switches, 27
us, 290–291
uino-to-Arduino communication with, 298–301
e, 292, 298
necting to, 291–293
configurations, 301–302
lines, 290
data, 290–291
o-Arduino remote control with, 295–298
32-to-TTL module, 292

ld, 291–292
et, 292
ages, 291
us, 304–305
uino-to-Arduino data link with, 309–312
uino-to-PC data link with, 306–309
necting to, 305–306
lines, 304–305
-duplex communication, 305
tiple-secondary-device control, 315–319
ote control operation with, 312–315
matic symbol for interface module, 306
smit/receive terminals, 305
ages, 305
ceived signal strength indicator), 416, 423
274
in, 274

S

`(boolean _halt)` function, 223
ies, 8, 16, 22, 26, 28, 36, 39, 43, 51, 59, 63, 66, 79, 81, 88, 101–102, 107, 114, 117, 126, 138, 155, 169, 190, 198, 215, 219, 260, 267, 231, 234, 273, 307, 310, 313, 355, 362, 375, 391, 402, 408

deploying, 389–390
ory cards, 220
message, 220
natting, 423
ng photos to, 423–429
ngs, 428
ry device, 143
() function, 236
er() function, 130
ers() function, 383
us() function, 122
egin() function, 301–302
onitor, 4–5, 15, 115, 153, 208, 327–329, 361, 371, 377, 398, 402, 408, 411, 416, 419, 421
rint() function, 160, 200
rintln() function, 160, 183, 332
eadBytesUntil() function, 301
egin() function, 332
ltime() function, 88, 103
) function, 259–260, 262–263
ata() function, 104
5 shift register, 111, 113
pendrow() function, 389
() function, 38, 124

isters, 1, 15
keyboard, building, 173
us() function, 122
DA shield, 105
DF shield, 105
ds, 105, 106
mote control, 105–109
ver control, 75
rating Arduino from external devices, 82–84
uino self-off, 81–82
231 real-time clock IC, 84–86
ta logger, 86–90
timer IC in bistable mode, 77–78
SFETs, 76–77
mally open (NO) contacts, 82–83
ff switch, 78–81
put logic devices, 83–84
es, 58
eSerial, 15, 106, 228
oard, building, 140–142
ensors, 266
un, 325
keys, 162–164
sheet, 166

88
2, 398, 402, 408
ng Video from a Basic Camera Server (project), 418–420
function, 407
tom function, 264
s, 27

T

hoto() function, 428
Photos and Saving Them to a Memory Card (project), 423–429
3A breakout board, 146–149
ress configuration, 148
selection, 149
ıe header pins, 147
3A I²C switch IC, 144
n, 351
account, creating, 352–353
ID, 353–354
iguring account, 352–355
nloading and installing, 352
ote control, 351–367
Arduino IDE, configuring, 355
utomating data transmission via, 365–367

of four LEDs, 355–361
retrieving remote data, 362–365
UniversalTelegramBot library, 355
n, 352–353
name, 352
network, 105
ture display
ting, 154–156
mple output, 155
s for, 154–156
ay size, 223
ture device
ding, 218–223
laying text, 219
ries, 223
n operation set up, 223
s for, 218
shield, 105–107
429
-hole HC-49-type crystal, 42
d weather server, ESP32, 407–411
mple output, 408
ldress, 408
s for, 407

nt.getDay() function, 374
nt.getEpochTime() function, 373
nt.getFormattedDate() function, 373–374
nt.getHours() function, 374
nt.getMinutes() function, 374
nt.setTimeOffset() function, 380
ime variable, 400
ntrolled mains outlet, 102–105
ive (TTL) logic data, 290
e, 105
352–353
95 shift register, 112–113
ry number display, 113–115
380–383
control with, 113
tiple, using, 123–124
uts, 113
matic symbol for, 112
tters, hacking, 96–100
R point, 84
c data, 290
) function, 90, 104–105
function, 104–105
D() function, 72

or Keypad with One Analog Output (project), 7–11
N-MOSFET, 76–77
in, 274

U

rary, 151–152
ns, 15
a mini ipex connector), 422
al TelegramBot, 355
, 429
, 429
270–272
o micro-USB cable, 160
tor, 251
.closeFile() function, 184, 187
.setFileName() function, 195
.writeChar(x) function, 184
.writeNum(x) function, 184
.writeNumln(x) function, 184
h drives
ending data to file, 187–189
uino library installation, 181
rface module, 180–182

ing sensor data to, 189–192
eet configuration via, 198–200
ing numerical data, 192–195
ing text from, 201–203
rmatting, 179
rity key, 196–197
cting and preparing, 180
ing and displaying integers on, 194
ing data, 183–187
board emulation
vating, 160
uino Keyboard library, 160, 162
mple output, 161–162
Leonardo, 160
ing data with, 165–168
ifiers, 162–164
sing and releasing one or more keys, 164–165
ial keys, 162–164
ping, 160–161
e, 173–174
urity key, 195–197
reed power, 251
S232, 293
cable, 415

)function, 200
Android App for Bluetooth Control (project), 236–245
uetooth for Digital I/O Control (project), 234–236
I2C Bus Scanner (project), 144–146
USB Security Key (project), 195–197

V

429
reaming, 418–421
gging, 421
nple output, 419
e rate, 410
e size, 421
sketch file, 410, 420–421
ldress, 418, 421
video stream, 419
s for, 418
eet files, downloading, 418
79, 106, 128, 324
`tolMP3()` function, 139
`p()` function, 55, 73, 104, 139, 168, 183, 197, 308, 318–319, 332, 373, 377, 383, 401
`dNumbers()` function, 131–132

up() function, 55, 73, 88, 90, 102–103, 109, 130, 183, 189, 197, 264, 284, 287, 301, 308, 361, 394, 401, 421, 428
dividers, 2–3
, 250

W

ı interrupt mode, 263–266
)g timer, 47
ding, 50–55
ıit configuration, 49–50
ry, 48–49
ng diagram, 48
/), 250
ver, 397–401
s. *See* online resources
, Andrew, 258

ss points, 327–328, 332. *See also* SSID
rna, external, 421–423
ıectivity, 327–328, 375
ıry, 332
rsketch, 327, 415–416
use() function, 174–175

/s devices
tooth pairing with, 227
ling data to, 228
inTransmission(0x27) function, 211
remote-control transmitters
ng links to button pads, 99
ery, 96
rol buttons, desoldering and removing, 98–99
cing, 96–100
97
pads, 98
power wires, 97
ng inside, 97
egers() function, 195
RF_REG() function, 428

Y

MP3 module, 133–137

Z

t, 141
files

.acOS, 435–436
buntu LTS, 434–438
Vindows, 434–435