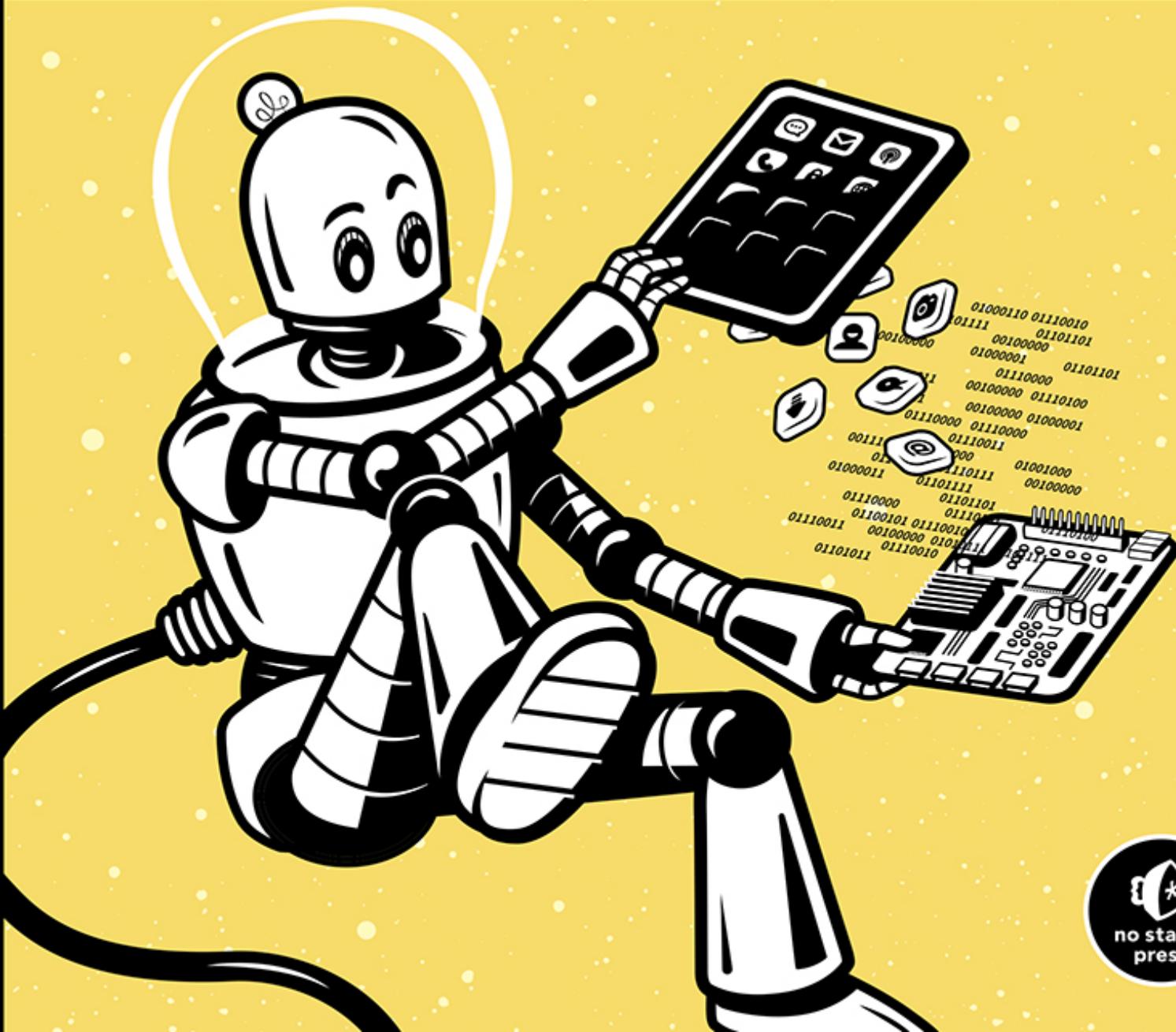


HOW COMPUTERS REALLY WORK

A HANDS-ON GUIDE TO THE INNER
WORKINGS OF THE MACHINE

MATTHEW JUSTICE



HOW COMPUTERS REALLY WORK

A Hands-On Guide to the Inner Workings of the Machine

by Matthew Justice



**no starch
press**

San Francisco

HOW COMPUTERS REALLY WORK. Copyright © 2021 by Matthew Justice.

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

ISBN-13: 978-1-71850-066-2 (print)

ISBN-13: 978-1-71850-067-9 (ebook)

Publisher: William Pollock

Executive Editor: Barbara Yien

Production Editor: Katrina Taylor

Developmental Editor: Alex Freed

Project Editor: Dapinder Dosanjh

Cover Design: Gina Redman

Interior Design: Octopod Studios

Technical Reviewers: William Young, John Hewes, and Bryan Wilhem

Copyeditor: Happenstance Type-O-Rama

Compositor: Happenstance Type-O-Rama

Proofreader: Happenstance Type-O-Rama

The following images are reproduced with permission:

Figures 7-7, 7-9, 7-10, and 7-11 ALU symbol was altered from the image created by Eadhem and is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license (https://commons.wikimedia.org/wiki/File:ALU_symbol-2.svg). Figures 11-3, 11-5, 11-14, 11-15, 11-16, 11-17, 12-4, 12-8, 12-9, 12-10, 13-5, 13-9 server icon is courtesy of Vecteezy.com.

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 1.415.863.9900; info@nostarch.com

www.nostarch.com

Library of Congress Cataloging-in-Publication Data:

Names: Justice, Matthew, author.

Title: How Computers Really Work : a hands-on guide to the inner workings of the machine / Matthew Justice.

Description: San Francisco : No Starch Press, Inc., [2020] | Includes index.

Identifiers: LCCN 2020024168 (print) | LCCN 2020024169 (ebook) | ISBN 9781718500662 (paperback) | ISBN 1718500661 (paperback) | ISBN

9781718500679 (ebook)

Subjects: LCSH: Electronic digital computers--Popular works.

Classification: LCC QA76.5 .J87 2020 (print) | LCC QA76.5 (ebook) | DDC
004--dc23

LC record available at <https://lccn.loc.gov/2020024168>

LC ebook record available at <https://lccn.loc.gov/2020024169>

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.

To my family, who believed in me while I tried something new.

About the Author

Matthew Justice is a software engineer. He spent 17 years at Microsoft where he took on various roles, including debugging the Windows kernel, developing automated fixes, and leading a team of engineers responsible for building diagnostic tools and services. He has worked on low-level software (the operating system) and on software far removed from the underlying hardware (such as web applications). Matthew has a degree in electrical engineering. When he's not writing code or building circuits, Matthew enjoys spending time with his family, hiking, reading, arranging music, and playing old video games.

About the Tech Reviewers

Dr. Bill Young is Associate Professor of Instruction in the Department of Computer Science at the University of Texas at Austin. Prior to joining the UT faculty in 2001, he had 20 years of experience in the industry. He specializes in formal methods and computer security, but often teaches computer architecture, among other courses.

Bryan Wilhelm is a software engineer. He has degrees in mathematics and computer science and has been working at Microsoft for 20 years in roles ranging from debugging the Windows kernel to developing business applications. He enjoys reading, science-fiction movies, and classical music.

John Hewes began connecting electrical circuits at an early age, moving on to electronics projects as a teenager. He later earned a physics degree and continued to develop his interest in electronics, helping school students with their projects while working as a science technician. John has taught electronics and physics up to an advanced level in the United Kingdom and ran a school electronics club for children aged 11 to 18 years, setting up the website <http://www.electronicsclub.info/> to support the club. He believes that everyone can enjoy building electronics projects, regardless of their age or ability.

BRIEF CONTENTS

Acknowledgments

Introduction

Chapter 1: Computing Concepts

Chapter 2: Binary in Action

Chapter 3: Electrical Circuits

Chapter 4: Digital Circuits

Chapter 5: Math with Digital Circuits

Chapter 6: Memory and Clock Signals

Chapter 7: Computer Hardware

Chapter 8: Machine Code and Assembly Language

Chapter 9: High-Level Programming

Chapter 10: Operating Systems

Chapter 11: The Internet

Chapter 12: The World Wide Web

Chapter 13: Modern Computing

Appendix A: Answers to Exercises

Appendix B: Resources

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

INTRODUCTION

Who Is This Book For?

About This Book

About Exercises and Projects

My Computing Journey

1

COMPUTING CONCEPTS

Defining a Computer

Analog and Digital

The Analog Approach

Going Digital

Number Systems

 Decimal Numbers

 Binary Numbers

Bits and Bytes

Prefixes

Hexadecimal

Summary

2

BINARY IN ACTION

Representing Data Digitally

 Digital Text

ASCII
Digital Colors and Images
Approaches for Representing Colors and Images
Interpreting Binary Data
Binary Logic
Summary

3 **ELECTRICAL CIRCUITS**

Electrical Terms Defined

Electric Charge
Electric Current
Voltage
Resistance
Water Analogy

Ohm's Law

Circuit Diagrams

Kirchhoff's Voltage Law

Circuits in the Real World

Light-Emitting Diodes

Summary

Project #1: Build and Measure a Circuit

Project #2: Build a Simple LED Circuit

4 **DIGITAL CIRCUITS**

What Is a Digital Circuit?
Logic with Mechanical Switches
The Amazing Transistor
Logic Gates
Designing with Logic Gates
Integrated Circuits

Summary

Project #3: Build Logical Operators (AND, OR) with Transistors

Project #4: Construct a Circuit with Logic Gates

5

MATH WITH DIGITAL CIRCUITS

Binary Addition

Half Adders

Full Adders

A 4-bit Adder

Signed Numbers

Unsigned Numbers

Summary

Project #5: Build a Half Adder

6

MEMORY AND CLOCK SIGNALS

Sequential Logic Circuits and Memory

The SR Latch

Using the SR Latch in a Circuit

Clock Signals

JK Flip-Flops

T Flip-Flops

Using a Clock in a 3-Bit Counter

Summary

Project #6: Construct an SR Latch Using NOR Gates

Project #7: Construct a Basic Vending Machine Circuit

Project #8: Add a Delayed Reset to the Vending Machine Circuit

Project #9: Using a Latch as a Manual Clock

Project #10: Test a JK Flip-Flop

Project #11: Construct a 3-bit Counter

7

COMPUTER HARDWARE

Computer Hardware Overview

Main Memory

Central Processing Unit (CPU)

 Instruction Set Architectures

 CPU Internals

 Clock, Cores, and Cache

Beyond Memory and Processor

 Secondary Storage

 Input/Output

Bus Communication

Summary

8

MACHINE CODE AND ASSEMBLY LANGUAGE

Software Terms Defined

An Example Machine Instruction

Calculating a Factorial in Machine Code

Summary

Project #12: Factorial in Assembly

Project #13: Examining Machine Code

9

HIGH-LEVEL PROGRAMMING

High-Level Programming Overview

Introduction to C and Python

Comments

Variables

 Variables in C

 Variables in Python

Stack and Heap Memory

The Stack
The Heap

Math
Logic

 Bitwise Operators
 Boolean Operators

Program Flow

 If Statements
 Looping

Functions

 Defining Functions
 Calling Functions
 Using Libraries

Object-Oriented Programming

Compiled or Interpreted

Calculating a Factorial in C

Summary

Project #14: Examine Variables

Project #15: Change the Type of Value Referenced by a Variable in Python

Project #16: Stack or Heap

Project #17: Write a Guessing Game

Project #18: Use a Bank Account Class in Python

Project #19: Factorial in C

10 **OPERATING SYSTEMS**

Programming Without an Operating System

Operating Systems Overview

Operating System Families

Kernel Mode and User Mode

Processes

Threads
Virtual Memory
Application Programming Interface (API)
The User Mode Bubble and System Calls
APIs and System Calls
Operating System Software Libraries
Application Binary Interface
Device Drivers
Filesystems
Services and Daemons
Security
Summary

Project #20: Examine Running Processes

Project #21: Create a Thread and Observe It

Project #22: Examine Virtual Memory

Project #23: Try the Operating System API

Project #24: Observe System Calls

Project #25: Use glibc

Project #26: View Loaded Kernel Modules

Project #27: Examine Storage Devices and Filesystems

Project #28: View Services

11
THE INTERNET

Networking Terms Defined
The Internet Protocol Suite

- Link Layer
- Internet Layer
- Transport Layer
- Application Layer

A Trip Through the Internet
Foundational Internet Capabilities

Dynamic Host Configuration Protocol
Private IP Addresses and Network Address Translation
The Domain Name System
Networking Is Computing
Summary

Project #29: Examine the Link Layer
Project #30: Examine the Internet Layer
Project #31: Examine Port Usage
Project #32: Trace the Route to a Host on the Internet
Project #33: See Your Leased IP Address
Project #34: Is Your Device's IP Public or Private?
Project #35: Find Information in DNS

12

THE WORLD WIDE WEB

Overview of the World Wide Web

The Distributed Web
The Addressable Web
The Linked Web
The Protocols of the Web
The Searchable Web

The Languages of the Web

Structuring the Web with HTML
Styling the Web with CSS
Scripting the Web with JavaScript
Structuring the Web's Data with JSON and XML

Web Browsers

Rendering a Page
The User Agent String

Web Servers

Summary

Project #36: Examine HTTP Traffic

- Project #37: Run Your Own Web Server**
- Project #38: Return HTML from Your Web Server**
- Project #39: Add CSS to Your Website**
- Project #40: Add JavaScript to Your Website**

13

MODERN COMPUTING

Apps

- Native Apps

- Web Apps

Virtualization and Emulation

- Virtualization

- Emulation

Cloud Computing

- The History of Remote Computing

- The Categories of Cloud Computing

The Deep Web and Dark Web

Bitcoin

- Bitcoin Basics

- Bitcoin Wallets

- Bitcoin Transactions

- Bitcoin Mining

Virtual Reality and Augmented Reality

The Internet of Things

Summary

Project #41: Use Python to Control a Vending Machine Circuit

A

ANSWERS TO EXERCISES

B

RESOURCES

Buying Electronic Components for the Projects

 7400 Part Numbers

 Shopping

Powering Digital Circuits

 USB Charger

 Breadboard Power Supply

 Power from a Raspberry Pi

 AA Batteries

Troubleshooting Circuits

Raspberry Pi

 Why Raspberry Pi

 Parts Needed

 Setting Up a Raspberry Pi

 Using Raspberry Pi OS

 Working with Files and Folders

INDEX

ACKNOWLEDGMENTS

An enormous thank you to my wife, Suzy, who acted as my informal editor, providing me with invaluable feedback. She scrutinized every word and every concept through multiple drafts of this book, helping me refine my ideas and express them clearly. She encouraged and supported me in this endeavor from concept to completion.

Thanks to my teenage daughters, Ava and Ivy, who read my early drafts and helped me see my writing through the eyes of younger learners. They helped me avoid confusing language and showed me where I needed to spend more time explaining things.

I want to express gratitude to my parents, Russell and Debby Justice, who always believed in me, and who provided me with ample opportunities to learn. My love of the written word comes from my Mom, and my engineering mindset comes from my Dad.

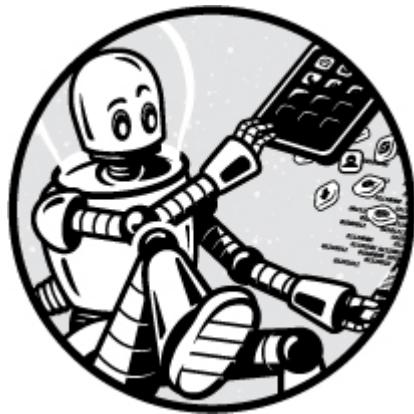
Thank you to the entire team at No Starch Press, especially Alex Freed and Katrina Taylor, and thanks to my copyeditor, Rebecca Rider. This was my first time authoring a book, and the editors at No Starch patiently guided me through the process. They saw opportunities for improvement that I hadn't considered, and they helped me communicate my ideas clearly. I've gained a new appreciation for the value that a publishing team can bring.

I'm thankful for this book's technical reviewers, John Hewes, Bryan Wilhelm, and Bill Young, who diligently examined the details of my writing. Their input resulted in content that's more accurate and more complete. They each brought a unique perspective and shared their valuable expertise.

Thanks to all the people at Microsoft who mentored me and collaborated with me throughout the years. I've been fortunate to work with some incredibly talented, intelligent, and knowledgeable folks—

there's too many of you to list! I'm only able to write a book like this because some great people at Microsoft took the time to share their knowledge with me.

INTRODUCTION



Are you curious about how computers work? Gaining a broad understanding of computing is often a long and winding path. The problem isn't a lack of documentation. A quick search online shows that there are many, many books and websites devoted to explaining computing. Programming, computer science, electronics, operating systems ... a wealth of information is out there. This is a good thing, but it can be daunting. Where should you begin? How does one topic connect to another? This book was written to give you an entry point for learning about key concepts of computing and how these concepts fit together.

When I worked as an engineering manager, I regularly interviewed people for software development jobs. I spoke to many candidates who knew how to write code, but a significant number of them didn't seem to know how computers really work. They knew how to make a computer do their bidding, but they didn't understand what was going on behind the scenes. Reflecting on those interviews, and on memories of my own struggles in trying to learn about computers, led me to write this book.

My goal is to present the fundamentals of computing in an accessible, hands-on way that makes abstract concepts more real. This book doesn't go deep on every topic presented, but instead, it presents the foundational concepts of computing and connects the dots between those concepts. I want you to be able to construct a mental picture of how computing works, enabling you to then dig deeper on the topics that interest you.

Computing is everywhere, and as our society depends more and more on technology, we need individuals who broadly understand computing. My hope is that this book will help you gain that broad perspective.

Who Is This Book For?

This book is for anyone who wants to understand how computers work. You don't need to have any prior knowledge of the topics covered—we begin with the basics. On the other hand, if you already have a background in programming or electronics, this book can help you expand your knowledge in other areas. The book is written for the self-motivated learner, someone who is comfortable with basic math and science, and who is already familiar with *using* computers and smartphones, but who still has questions about *how they work*. Teachers should find the content useful too; I believe the projects are a good fit for the classroom.

About This Book

This book looks at computing as a technology stack. A modern computing device, such as a smartphone, is composed of layers of technology. At the bottom of the stack we have hardware, and at the top of the stack we have apps, with multiple technology layers in between. The beauty of the layered model is that each layer benefits from all the capabilities of the lower levels, but any given layer only needs to build

upon the layer directly below it. After covering some foundational concepts, we'll progress through the technology stack from the bottom up, starting with electrical circuits and working our way up to the technologies that power the web and apps. Here's what we'll cover in each chapter:

Chapter 1: Computing Concepts Covers foundational ideas, such as understanding analog versus digital, the binary number system, and SI prefixes

Chapter 2: Binary in Action Looks at how binary can be used to represent both data and logical states, and includes an introduction to logical operators

Chapter 3: Electrical Circuits Explains basic concepts of electricity and electrical circuits, including voltage, current, and resistance

Chapter 4: Digital Circuits Introduces transistors and logic gates, and brings together concepts from Chapters 2 and 3

Chapter 5: Math with Digital Circuits Shows how addition can be performed with digital circuits and covers more details about how numbers are represented in computers

Chapter 6: Memory and Clock Signals Introduces memory devices and sequential circuits, and demonstrates synchronization through clock signals

Chapter 7: Computer Hardware Covers the major parts of a computer: processor, memory, and input/output

Chapter 8: Machine Code and Assembly Language Presents low-level machine code that processors execute, and covers assembly language, the human-readable form of machine code

Chapter 9: High-Level Programming Introduces programming languages that are independent from specific processors, and includes example code in C and Python

Chapter 10: Operating Systems Covers families of operating systems and the core capabilities of operating systems

Chapter 11: The Internet Looks at how the internet works, including the common suite of network protocols it uses

Chapter 12: The World Wide Web Explains how the web works, and looks at its core technologies: HTTP, HTML, CSS, and JavaScript

Chapter 13: Modern Computing Provides overviews of a handful of modern computing topics, such as apps, virtualization, and cloud services

As you read through this book, you'll come across circuit diagrams and source code used to illustrate concepts. These are intended as teaching tools, favoring clarity over performance, security, and other factors that engineers consider when designing hardware or software. In other words, the circuits and code in this book can help you learn how computers work, but they aren't necessarily examples of the best way to do things. Similarly, the book's technical explanations favor simplicity over completeness. I sometimes gloss over certain details to avoid getting mired in complexity.

About Exercises and Projects

Throughout the chapters you'll find exercises and hands-on projects. The exercises are problems for you to work out mentally or with pencil and paper. The projects go beyond mental exercises and often involve building a circuit or programming a computer.

You'll need to acquire some hardware to perform the projects (you can find a list of needed components in Appendix B). I've included these projects because I believe that the best way to learn is to try things yourself, and I encourage you to complete the projects if you want to get the most out of this book. That said, I've presented the chapter

material in a way that allows you to still follow along, even if you don’t build a single circuit or enter one line of code.

You can find the answers to exercises in Appendix A, and the details for each project are found at the end of the corresponding chapter. Appendix B contains information to help you get started with projects, and the project text points you there when needed.

A copy of the source code used in the projects is available at <https://www.howcomputersreallywork.com/code/>. You can also visit this book’s page at <https://nostarch.com/how-computers-really-work/> where we will provide updates.

My Computing Journey

My fascination with computers probably began with the video games I played as a kid. When I visited my grandparents, I’d spend hours playing Frogger, Pac-Man, and Donkey Kong on my aunt’s Atari 2600. Later, when I was in fifth grade, my parents gave me a Nintendo Entertainment System for Christmas, and I was thrilled! While I loved playing Super Mario Bros. and Double Dragon, somewhere along the way, I began to wonder how video games and computers worked. Unfortunately, my Nintendo game console didn’t provide me with many clues as to what was going on inside it.

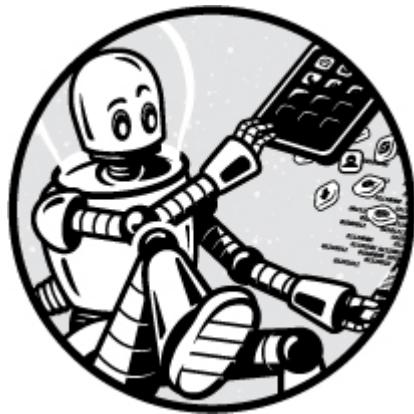
Around the same time, my family bought our first “real” computer, an Apple II GS, opening new doors for me to explore exactly how these machines functioned. Fortunately, my middle school offered a class on BASIC programming of Apple II computers, and I soon learned that I couldn’t get enough of coding! I would write code at school, bring home a copy of my work on floppy disk, and continue working at home. Throughout middle school and high school, I learned more about programming, and I greatly enjoyed it. I also began to realize that although BASIC and other similar programming languages make it relatively easy to tell a computer what to do, they also hide many details of how computers work. I wanted to go deeper.

In college I studied electrical engineering, and I began to understand electronics and digital circuits. I took classes on C programming and assembly language, and I finally got glimpses of how computers execute instructions. The low-level details of how computers work were beginning to make sense. While in college, I also started learning about this new thing called the World Wide Web; I even crafted my very own web page (this seemed like a big deal at the time)! I began programming Windows applications, and I was introduced to Unix and Linux. These topics sometimes seemed far removed from the hardware-specific details of digital circuits and assembly language, and I was curious to understand how it all fit together.

After college I was fortunate to get a job at Microsoft. In my 17 years there, I worked in various software engineering roles, from debugging the Windows kernel to developing web applications. These experiences helped me gain a broader and deeper understanding of computers. I worked with many incredibly smart and knowledgeable people, and I learned that there's always more to learn about computing. Understanding how computers work has been a lifelong journey for me, and I hope to pass on some of what I've learned to you through this book.

1

COMPUTING CONCEPTS



Computers are everywhere now: in our homes, our schools, our offices —you might find a computer in your pocket, on your wrist, or even in your refrigerator. It's easier than ever to find and use computers, but few people today really understand how computers work. This isn't surprising, since learning the complexities of computing can be overwhelming. The goal of this book is to lay out the foundational principles of computing in a way that anyone with curiosity, and a bit of a technical bent, can follow. Before we dig into the nuts and bolts of how computers work, let's take some time to get familiar with some major concepts of computing.

In this chapter we'll begin by discussing the definition of a computer. From there, we'll cover the differences between analog and digital data and then explore number systems and the terminology used to describe digital data.

Defining a Computer

Let's start with a basic question: what is a computer? When people hear the word *computer*, most think of a laptop or desktop, sometimes

referred to as a personal computer or PC. That is one class of device that this book covers, but let's think a bit more broadly. Consider smartphones. Smartphones are certainly computers; they perform the same types of operations as PCs. In fact, for many people today, a smartphone is their primary computing device. Most computer users today also rely on the internet, which is powered by servers—another type of computer. Every time you visit a website or use an app that connects to the internet, you're interacting with one or more servers connected to a global network. Video game consoles, fitness trackers, smart watches, smart televisions ... all of these are computers!

A *computer* is any electronic device that can be programmed to carry out a set of logical instructions. With that definition in mind, it becomes clear that many modern devices are in fact computers!

EXERCISE 1-1: FIND THE COMPUTERS IN YOUR HOME

Take a moment and see how many computers you can identify in your home. When I did this exercise with my family, we quickly found about 30 devices!

Analog and Digital

You've probably heard a computer described as a digital device. This is in contrast to an analog device, such as a mechanical clock. But what do these two terms really mean? Understanding the differences between analog and digital is foundational to understanding computing, so let's take a closer look at these two concepts.

The Analog Approach

Look around you. Pick an object. Ask yourself: What color is it? What size is it? How much does it weigh? By answering these questions, you're describing the attributes, or *data*, of that object. Now, pick a different object and answer the same questions. If you repeat this

process for even more objects, you'll find that for each question, the potential answers are numerous. You might pick up a red object, a yellow object, or a blue object. Or the object could be a mix of the primary colors. This type of variation does not only apply to color. For a given property, the variations found across the objects in our world are potentially infinite.

It's one thing to describe an object verbally, but let's say you want to measure one of its attributes more precisely. If you wanted to measure an object's weight, for example, you could put it on a scale. The scale, responding to the weight placed upon it, would move a needle along a numbered line, stopping when it reaches a position that corresponds to the weight. Read the number from the scale and you have the object's weight.

This kind of measurement is common, but let's think a little more about how we're measuring this data. The position of the needle on the scale isn't actually the weight; it's a representation of the weight. The numbered line that the needle points to provides a means for us to easily convert between the needle's position, representing a weight, and the numeric value of that weight. In other words, though the weight is an attribute of the object, here we can understand that attribute through something else: the position of the needle along the line. The needle's position changes proportionally in response to the weight placed on the scale. Thus, the scale is working as an *analogy* where we understand the weight of the object through the needle's position on the line. This is why we call this method of measuring the *analog* approach.

Another example of an analog measuring tool is a mercury thermometer. Mercury's volume increases with temperature. Thermometer manufacturers utilize this property by placing mercury in a glass tube with markings that correspond to the expected volume of the mercury at various temperatures. Thus, the position of mercury in the tube serves as a representation of temperature. Notice that for both of these examples (a scale and a thermometer), when we make a measurement, we can use markings on the instrument to convert a position to a specific numeric value. But the value we read from the

instrument is just an approximation. The true position of the needle or mercury can be anywhere within the range of the instrument, and we round up or down to the nearest marked value. So although it may seem that these tools can produce only a finite set of measurements, that's a limitation imposed by the conversion to a number, not by the analogy itself.

Throughout most of human history, humans have measured things using an analog approach. But people don't only use analog approaches for measurement. They've also devised clever ways to store data in an analog fashion. A phonograph record uses a modulated groove as an analog representation of audio that was recorded. The groove's shape changes along its path in a way that corresponds to changes in the shape of the audio waveform over time. The groove isn't the audio itself, but it's an analogy of the original sound's waveform. Film-based cameras do something similar by briefly exposing film to light from a camera lens, leading to a chemical change in the film. The chemical properties of the film are not the image itself, but a representation of the captured image, an analogy of the image.

Going Digital

What does all this have to do with computing? It turns out that all those analog representations of data are hard for computers to deal with. The types of analog systems used are so different and variable that creating a common computing device that can understand all of them is nearly impossible. For example, creating a machine that can measure the volume of mercury is a very different task than creating a machine that can read the grooves on a vinyl disc. Additionally, computers require highly reliable and accurate representations of certain types of data, such as numeric data sets and software programs. Analog representations of data can be difficult to measure precisely, tend to decay over time, and lose fidelity when copied. Computers need a way to represent all types of data in a format that can be accurately processed, stored, and copied.

If we don't want to represent data as something with potentially infinitely varying analog values, what can we do? We can use a digital approach instead. A *digital* system represents data as a sequence of symbols, where each symbol is one of a limited set of values. Now, that description may sound a bit formal and a bit confusing, so rather than go deep on the theory of digital systems, I'll explain what this means in practice. In almost all of today's computers, data is represented with combinations of two symbols: 0 and 1. That's it. Although a digital system could use more than two symbols, adding more symbols would increase the complexity and cost of the system. A set of only two symbols allows for simplified hardware and improved reliability. All data in most modern computing devices is represented as a sequence of 0s and 1s. From this point forward in this book, when I talk about digital computers, you can assume that I am talking about systems that only deal with 0s and 1s and not some other set of symbols. Nice and simple!

It's a point worth repeating: everything on your computer is stored as 0s and 1s. The last photo you took on your smartphone? Your device stored the photo as a sequence of 0s and 1s. The song you streamed from the internet? 0s and 1s. The document you wrote on your computer? 0s and 1s. The app you installed? It was a bunch of 0s and 1s. The website you visited? 0s and 1s.

It may sound limiting to say that we can only use 0 and 1 to represent the infinite values found in nature. How can a musical recording or a detailed photograph be distilled down to 0s and 1s? Many find it counterintuitive that such a limited "vocabulary" can be used to express complex ideas. The key here is that digital systems use a *sequence* of 0s and 1s. A digital photograph, for example, usually consists of millions of 0s and 1s.

So what exactly are these 0s and 1s? You may see other terms used to describe these 0s and 1s: false and true, off and on, low and high, and so forth. This is because the computer doesn't literally store the number *0* or *1*. It stores a sequence of entries where each entry in the sequence can have only two possible states. Each entry is like a light switch that is

either on or off. In practice, these sequences of 1s and 0s are stored in various ways. On a CD or DVD, the 0s and 1s are stored on the disc as bumps (0) or flat spaces (1). On a flash drive, the 1s and 0s are stored as electrical charges. A hard disk drive stores the 0s and 1s using magnetization. As you'll see in Chapter 4, digital circuits represent 0s and 1s using voltage levels.

Before we move on, one final note on the term *analog*—it's often used to simply mean “not digital.” For example, engineers may speak of an “analog signal,” meaning a signal that varies continuously and doesn't align to digital values. In other words, it's a non-digital signal but doesn't necessarily represent an analogy of something else. So, when you see the term *analog*, consider that it might not always mean what you think.

Number Systems

So far, we've established that computers are digital machines that deal with 0s and 1s. For many people, this concept seems strange; they're used to having 0 through 9 at their disposal when representing numbers. If we constrain ourselves to only two symbols, rather than ten, how should we represent large numbers? To answer that question, let's back up and review an elementary school math topic: number systems.

Decimal Numbers

We typically write numbers using something called *decimal place-value notation*. Let's break that down. *Place-value notation* (or *positional notation*) means that each position in a written number represents a different order of magnitude; *decimal*, or *base 10*, means that the orders of magnitude are factors of 10, and each place can have one of ten different symbols, 0 through 9. Look at the example of place-value notation in Figure 1-1.



Figure 1-1: Two hundred seventy-five represented in decimal place-value notation

In Figure 1-1, the number two hundred seventy-five is written in decimal notation as 275. The 5 is in the ones place, meaning its value is $5 \times 1 = 5$. The 7 is in the tens place, meaning its value is $7 \times 10 = 70$. The 2 is in the hundreds place, meaning its value is $2 \times 100 = 200$. The total value is the sum of all the places: $5 + 70 + 200 = 275$.

Easy, right? You've probably understood this since first grade. But let's examine this a bit closer. Why is the rightmost place the ones place? And why is the next place the tens place, and so on? It's because we are working in decimal, or base 10, and therefore each place is a power of ten—in other words, 10 multiplied by itself a certain number of times. As seen in Figure 1-2, the rightmost place is 10 raised to 0, which is 1, because any number raised to 0 is 1. The next place is 10 raised to 1, which is 10, and the next place is 10 raised to 2 (10×10), which is 100.

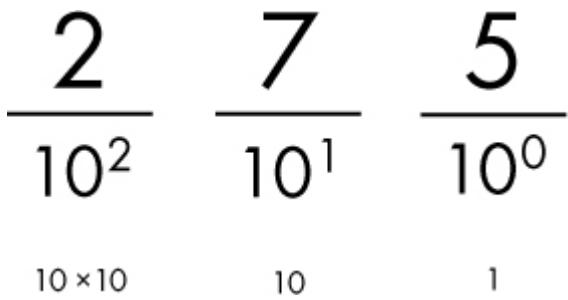


Figure 1-2: In decimal place-value notation, each place is a power of ten.

If we needed to represent a number larger than 999 in decimal, we'd add another place to the left, the thousands place, and its weight would be equal to 10 raised to 3 ($10 \times 10 \times 10$), which is 1,000. This pattern continues so that we can represent any large whole number by adding more places as needed.

We've established why the various places have certain weights, but let's keep digging. Why does each place use the symbols 0 through 9? When working in decimal, we can only have ten symbols, because by definition each place can only represent ten different values. 0 through 9 are the symbols that are currently used, but really any set of ten unique symbols could be used, with each symbol corresponding to a certain numeric value.

Most humans prefer decimal, base 10, as a number system. Some say this is because we have ten fingers and ten toes, but whatever the reason, in the modern world most people read, write, and think of numbers in decimal. Of course, that's just a convention we've collectively chosen to represent numbers. As we covered earlier, that convention doesn't apply to computers, which instead use only two symbols. Let's see how we can apply the principles of the place-value system while constraining ourselves to only two symbols.

Binary Numbers

The number system consisting of only two symbols is *base 2*, or *binary*. Binary is still a place-value system, so the fundamental mechanics are the same as decimal, but there are a couple of changes. First, each place represents a power of 2, rather than a power of 10. Second, each place can only have one of two symbols, rather than ten. Those two symbols are 0 and 1. Figure 1-3 has an example of how we'd represent a number using binary.

<u>1</u>	<u>0</u>	<u>1</u>
Fours place	Twos place	Ones place
2^2	2^1	2^0
$2 \times 2 = 4$	2	1

Figure 1-3: Five decimal represented in binary place-value notation

In Figure 1-3, we have a binary number: 101. That may look like one hundred and one to you, but when dealing in binary, this is actually a representation of five! If you wish to verbally say it, “one zero one binary” would be a good way to communicate what is written.

Just like in decimal, each place has a weight equal to the base raised to various powers. Since we are in base 2, the rightmost place is 2 raised to 0, which is 1. The next place is 2 raised to 1, which is 2, and the next place is 2 raised to 2 (2×2), which is 4. Also, just like in decimal, to get the total value, we multiply the symbol in each place by the place-value weight and sum the results. So, starting from the right, we have $(1 \times 1) + (0 \times 2) + (1 \times 4) = 5$.

Now you can try converting from binary to decimal yourself.

EXERCISE 1-2: BINARY TO DECIMAL

Convert these numbers, represented in binary, to their decimal equivalents.

10 (binary) = _____ (decimal)

111 (binary) = _____ (decimal)

1010 (binary) = _____ (decimal)

You can check your answers in Appendix A. Did you get them right? The last one might have been a bit tricky, since it introduced another place to the left, the eights place. Now, try going the other way around, from decimal to binary.

EXERCISE 1-3: DECIMAL TO BINARY

Convert these numbers, represented in decimal, to their binary equivalents.

3 (decimal) = _____ (binary)

8 (decimal) = _____ (binary)

14 (decimal) = _____ (binary)

I hope you got those correct too! Right away, you can see that dealing with both decimal and binary at the same time can be confusing, since a number like 10 represents ten in decimal or two in binary. From this point forward in the book, if there's a chance of confusion, binary numbers will be written with a 0b prefix. I've chosen the 0b prefix because several programming languages use this approach. The leading 0 (zero) character indicates a numeric value and the b is short for binary. As an example, 0b10 represents two in binary, whereas 10, with no prefix, means ten in decimal.

Bits and Bytes

A single place or symbol in a decimal number is called a *digit*. A decimal number like 1,247 is a four-digit number. Similarly, a single place or symbol in a binary number is called a *bit* (a binary digit). Each bit can either be 0 or 1. A binary number like 0b110 is a 3-bit number.

A single bit cannot convey much information; it's either off or on, 0 or 1. We need a sequence of bits to represent anything more complex. To make these sequences of bits easier to manage, computers group bits together in sets of eight, called *bytes*. Here are some examples of bits and bytes (leaving off the 0b prefix since they are all binary):

1 That's a bit.

0 That is also a bit.

11001110 That's a byte, or 8 bits.

00111000 That's also a byte!

10100101 Yet another byte.

0011100010100101 That's two bytes, or 16 bits.

NOTE

Fun fact: A 4-bit number, half a byte, is sometimes called a nibble (sometimes spelled nybble or nyble).

So how much data can we store in a byte? Another way to think about this question is how many unique combinations of 0s and 1s can we make with our 8 bits? Before we answer that question, let me illustrate with only 4 bits, as it'll be easier to visualize.

In Table 1-1, I've listed all the possible combination of 0s and 1s in a 4-bit number. I've also included the corresponding decimal representation of that number.

Table 1-1: All Possible Values of a 4-bit Number

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

As you can see in Table 1-1, we can represent 16 unique combinations of 0s and 1s in a 4-bit number, ranging in decimal value from 0 to 15. Seeing the list of combinations of bits helps to illustrate this, but we could have figured this out in a couple of ways without enumerating every possible combination.

We could determine the largest possible number that 4 bits can represent by setting all the bits to one, giving us 0b1111. That is 15 in

decimal; if we add 1 to account for representing 0, then we come to our total of 16. Another shortcut is to raise 2 to the number of bits, 4 in this case, which gives us $2^4 = 2 \times 2 \times 2 \times 2 = 16$ total combinations of 0s and 1s.

Looking at 4 bits is a good start, but previously we were talking about bytes, which contain 8 bits. Using the preceding approach, we could list out all combinations of 0s and 1s, but let's skip that step and go straight to a shortcut. Raise 2 to the power of 8 and you get 256, so that's the number of unique combinations of bits in a byte.

Now we know that a 4-bit number allows for 16 combinations of 0s and 1s, and a byte allows for 256 combinations. What does that have to do with computing? Let's say that a computer game has 12 levels; the game could easily store the current level number in only 4 bits. On the other hand, if the game has 99 levels, 4 bits won't be enough ... only 16 levels could be represented! A byte, on the other hand, would handle that 99-level requirement just fine. Computer engineers sometimes need to consider how many bits or bytes will be needed for storage of data.

Prefixes

Representing complex data types takes a large number of bits. Something as simple as the number 99 won't require more than a byte; a video in a digital format, on the other hand, can require billions of bits. To more easily communicate the size of data, we use prefixes like giga- and mega-. The *International System of Units (SI)*, also known as the *metric system*, defines a set of standard prefixes. These prefixes are used to describe anything that can be quantified, not just bits. We'll see them again in upcoming chapters dealing with electrical circuits. Table 1-2 lists some of the common SI prefixes and their meanings.

Table 1-2: Common SI Prefixes

Prefix name	Prefix symbol	Value	Base 10	English word
tera	T	1,000,000,000,000	10^{12}	trillion
giga	G	1,000,000,000	10^9	billion
mega	M	1,000,000	10^6	million
kilo	k	1,000	10^3	thousand
centi	c	0.01	10^{-2}	hundredth
milli	m	0.001	10^{-3}	thousandth
micro	μ	0.000001	10^{-6}	millionth
nano	n	0.000000001	10^{-9}	billionth
pico	p	0.000000000001	10^{-12}	trillionth

With these prefixes, if we want to say “3 billion bytes,” we can use the shorthand 3GB. Or if we want to represent 4 thousand bits, we can say 4kb. Note the uppercase B for byte and lowercase b for bit.

You’ll find that this convention is commonly used to represent quantities of bits and bytes. Unfortunately, it’s also often technically incorrect. Here’s why: when dealing with bytes, most software is actually working in base 2, not base 10. If your computer tells you that a file is 1MB in size, it is actually 1,048,576 bytes! That is approximately one million, but not quite. Seems like an odd number, doesn’t it? That’s because we are looking at it in decimal. In binary, that same number is expressed as 0b100000000000000000000000. It’s a power of two, specifically 2^{20} . Table 1-3 shows how to interpret the SI prefixes when dealing with bytes.

Table 1-3: SI Prefix Meaning When Applied to Bytes

Prefix name	Prefix symbol	Value	Base 2
tera	T	1,099,511,627,776	2^{40}
giga	G	1,073,741,824	2^{30}
mega	M	1,048,576	2^{20}
kilo	k	1,024	2^{10}

Another point of confusion with bits and bytes relates to network transfer rates. Internet service providers usually advertise in bits per second, base 10. So, if you get 50 megabits per second from your internet connection, that means you can only transfer about 6 megabytes per second. That is, 50,000,000 bits per second divided by 8 bits per byte gives us 6,250,000 bytes per second. Divide 6,250,000 by 2^{20} and we get about 6 megabytes per second.

SI PREFIXES FOR BINARY DATA

To address the confusion caused by multiple meanings of prefixes, a new set of prefixes was introduced in 2002 (in a standard called IEEE 1541) to be used for binary scenarios. When dealing with powers of 2, kibi- is to be used instead of kilo-, mebi- is to be used instead of mega-, and so on. These new prefixes correspond to base 2 values and are intended to be used in scenarios where the old prefixes were previously being used incorrectly. For example, since kilobyte might be interpreted as 1,000 or 1,024 bytes, this standard recommends that kibibyte be used to mean 1,024 bytes, while kilo- retains its original meaning so that a kilobyte is equal to 1,000 bytes.

This seems like a good idea, but at the time of this writing, these symbols haven't been widely adopted. Table 1-4 lists the new prefixes and their meanings.

Table 1-4: IEEE 1541-2002 Prefixes for Binary Data

Prefix name	Prefix symbol	Value	Base 2
tebi	Ti	1,099,511,627,776	2^{40}
gibi	Gi	1,073,741,824	2^{30}
mebi	Mi	1,048,576	2^{20}
kibi	Ki	1,024	2^{10}

This distinction is important because in practice, most software that displays the size of files uses the old SI prefix but calculates size using base 2. In other words, if your device says a file's size is 1KB, it means 1,024 bytes. On the other hand, manufacturers of storage devices tend to advertise the capacity of their devices using base 10. This means that a hard drive that is advertised as 1TB probably holds 1 trillion bytes, but if you connect that device to a computer, the computer will show the size as about 931GB (1

trillion divided by 2^{30}). Given the lack of standard adoption of the new prefixes, in this book, I will continue to use the old SI prefixes.

Hexadecimal

Before we leave the topic of thinking in binary, I'll cover one more number system: hexadecimal. Quickly reviewing, our "normal" number system is decimal, or base 10. Computers use binary, or base 2.

Hexadecimal is *base 16*! Given what you've already learned in this chapter, you probably know what that means. Hexadecimal, or just *hex* for short, is a place-value system where each place represents a power of 16, and each place can be one of 16 symbols.

As in all place-value systems, the rightmost place will still be the ones place. The next place to the left will be the sixteens place, then the 256s (16×16) place, then the 4,096s ($16 \times 16 \times 16$) place, and so on. Simple enough. But what about the other requirement that each place can be one of 16 symbols? We usually have ten symbols to use to represent numbers, 0 through 9. We need to add six more symbols to represent the other values. We could pick some random symbols like & @ #, but these symbols have no obvious order. Instead, the standard is to use A, B, C, D, E, and F (either uppercase or lowercase is fine!). In this scheme, A represents ten, B represents eleven, and so on, up to F, which represents fifteen. That makes sense; we need symbols that represent zero through one less than the base. So our extra symbols are A through F. It's standard practice to use the prefix 0x to indicate hexadecimal, when needed for clarity. Table 1-5 lists each of the 16 hexadecimal symbols, along with their decimal and binary equivalents.

Table 1-5: Hexadecimal Symbols

Hexadecimal	Decimal	Binary (4-bit)
0	0	0000
1	1	0001
2	2	0010

Hexadecimal	Decimal	Binary (4-bit)
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

What happens when you need to count higher than 15 decimal or 0xF? Just like in decimal, we add another place. After 0xF comes 0x10, which is 16 decimal. Then 0x11, 0x12, 0x13, and so on. Now take a look at Figure 1-4, where we see a larger hexadecimal number, 0x1A5.

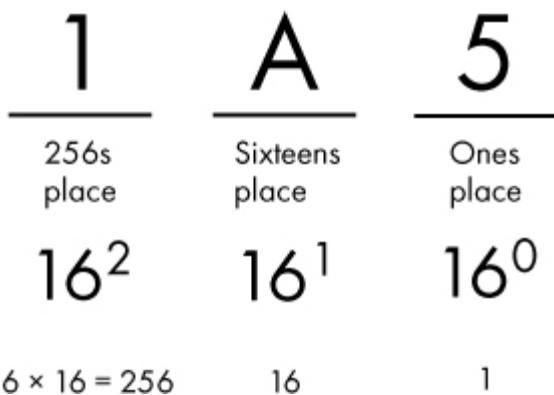


Figure 1-4: Hexadecimal number 0x1A5 broken out by place value

In Figure 1-4 we have the number 0x1A5 in hexadecimal. What's the value of this number in decimal? The rightmost place is worth 5. The next place has a weight of 16, and there's an A there, which is 10 in decimal, so the middle place is worth $16 \times 10 = 160$. The leftmost place

has a weight of 256, and there's a 1 in that place, so that place is worth 256. The total value then is $5 + 160 + 256 = 421$ in decimal.

Just to reinforce the point, this example shows how the new symbols, like A, have a different value depending on the place in which they appear. 0xA is 10 decimal, but 0xA0 is 160 in decimal, because the A appears in the sixteens place.

At this point you may be saying to yourself “great, but what use is this?” I’m glad you asked. Computers don’t use hexadecimal, and neither do most people. And yet, hexadecimal is very useful for people who need to work in binary.

Using hexadecimal helps overcome two common difficulties with working in binary. First, most people are terrible at reading long sequences of 0s and 1s. After a while the bits all run together. Dealing with 16 or more bits is tedious and error-prone for humans. The second problem is that although people are good at working in decimal, converting between decimal and binary isn’t easy. It’s tough for most people to look at a decimal number and quickly tell which bits would be 1 or 0 if that number were represented in binary. But with hexadecimal, conversions to binary are much more straightforward. Table 1-6 provides a couple of examples of 16-bit binary numbers and their corresponding hexadecimal and decimal representations. Note that I’ve added spaces to the binary values for clarity.

Table 1-6: Examples of 16-bit Binary Numbers as Decimal and Hexadecimal

	Example 1	Example 2
Binary	1111 0000 0000 1111	1000 1000 1000 0001
Hexadecimal	F00F	8881
Decimal	61,455	34,945

Consider Example 1 in Table 1-6. In binary, there’s a clear sequence: the first four bits are 1, the next eight bits are 0, and the last four bits are 1. In decimal, this sequence is obscured. It isn’t clear at all from looking at 61,455 which bits might be set to 0 or 1. Hexadecimal, on the

other hand, mirrors the sequence in binary. The first hex symbol is F (which is 1111 in binary), the next two hex symbols are 0, and the final hex symbol is F.

Continuing to Example 2, the first three sets of four bits are all 1000 and the final set of four bits is 0001. That's easy to see in binary, but rather hard to see in decimal. Hexadecimal provides a clearer picture, with the hexadecimal symbol of 8 corresponding to 1000 in binary and the hexadecimal symbol of 1 corresponding to, well, 1!

I hope you are seeing a pattern emerge: every four bits in binary correspond to one symbol in hexadecimal. If you remember, four bits is half a byte (or a nibble). Therefore, a byte can be easily represented with two hexadecimal symbols. A 16-bit number can be represented with four hex symbols, a 32-bit number with eight hex symbols, and so on. Let's take the 32-bit number in Figure 1-5 as an example.

8A52FF00

1000 1010 0101 0010 1111 1111 0000 0000

Figure 1-5: Each hexadecimal character maps to 4 bits

In Figure 1-5 we can digest this rather long number one half-byte at a time, something that isn't possible using a decimal representation of the same number (2,320,695,040).

Because it's relatively easy to move between binary and hex, many engineers will often use the two in tandem, converting to decimal numbers only when necessary. I'll use hexadecimal later in this book where it makes sense.

Try converting from binary to hexadecimal without going through the intermediate step of converting to decimal.

EXERCISE 1-4: BINARY TO HEXADECIMAL

Convert these numbers, represented in binary, to their hexadecimal equivalents. Don't convert to decimal if you can help it! The goal is to move directly from binary to hexadecimal.

10 (binary) = _____ (hexadecimal)

11110000 (binary) = _____ (hexadecimal)

You can check your answers in Appendix A.

Once you have the hang of binary to hexadecimal, try going the other way, from hex to binary.

EXERCISE 1-5: HEXADECIMAL TO BINARY

Convert these numbers, represented in hexadecimal, to their binary equivalents. Don't convert to decimal if you can help it! The goal is to move directly from hexadecimal to binary.

1A (hexadecimal) = _____ (binary)

C3A0 (hexadecimal) = _____ (binary)

You can check your answers in Appendix A.

Summary

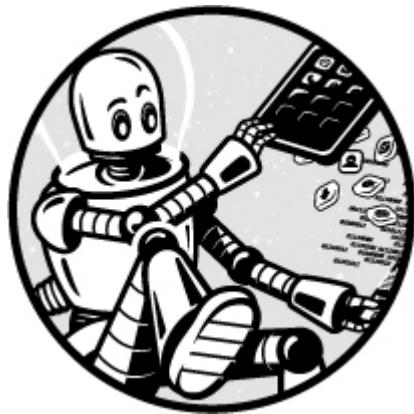
In this chapter, we covered some of the foundational concepts of computing. You learned that a computer is any electronic device that can be programmed to carry out a set of logical instructions. You then saw that modern computers are digital devices rather than analog devices, and you learned the difference between the two: analog systems are those that use widely varying values to represent data, whereas digital systems represent data as a sequence of symbols. After that, we explored how modern digital computers rely on only two symbols, 0 and 1, and learned about a number system consisting of only two symbols, base 2, or binary. We covered bits, bytes, and the standard SI prefixes (giga-, mega-, kilo-, and so on) you can use to more easily

describe the size of data. Lastly, you learned how hexadecimal is useful for people who need to work in binary.

In the next chapter we'll look more closely at how binary is used in digital systems. We'll take a look at how binary can be used to represent various types of data, and we'll see how binary logic works.

2

BINARY IN ACTION



In the previous chapter we defined a computer as an electronic device that can be programmed to carry out a set of logical instructions. We then learned at a high level how everything in a computer, from the data it uses to the instructions it carries out, is stored in binary, 0s and 1s. In this chapter, I shed some light on how exactly 0s and 1s can be used to represent nearly any kind of data. We also cover how binary lends itself to logical operations.

Representing Data Digitally

So far, we've focused on storing numbers in binary. More specifically, we covered how to store the positive integers, sometimes called whole numbers, and zero. However, computers store all data as bits: negative numbers, fractional numbers, text, colors, images, audio, and video, to name a few. Let's consider how various types of data might be represented using binary.

Digital Text

Let's begin with text as our first example of how bits, 0s and 1s, can represent something other than a number. In the context of computing, *text* means a collection of alphanumeric and related symbols, also called *characters*. Text is usually used to represent words, sentences, paragraphs, and so forth. Text does not include formatting (bold, italics). For the purposes of this discussion, let's limit our character set to the English alphabet and related characters. In computer programming, the term *string* is also commonly used to refer to a sequence of text characters.

Keeping that definition of text in mind, what exactly do we need to represent? We need A through Z, uppercase and lowercase, meaning A is a different symbol than a. We also want punctuation marks like commas and periods. We need a way to represent spaces. We also need digits 0 through 9. The digit requirement can be confusing; here I'm talking about including the *symbols* or *characters* that are used to represent the numbers 0 through 9, which is a different thing than storing the *numbers* 0 through 9.

If we add up all the unique symbols we need to represent, just described, we have around 100 characters. So, if we need to have a unique combination of bits to represent each character, how many bits do we need per character? A 6-bit number gives us 64 unique combinations, which isn't quite enough. But a 7-bit number gives us 128 combinations, enough to represent the 100 or so characters we need. However, since computers usually work in bytes, it makes sense to just round up and use a full 8 bits, one byte, to represent each character. With a byte we can represent 256 unique characters.

So how might we go about using 8 bits to represent each character? As you may expect, there's already a standard way of representing text in binary, and we'll get to that in a minute. But before we do that, it's important to understand that we can make up any scheme we want to represent each character, as long as the software running on a computer knows about our scheme. That said, some schemes are better than others for representing certain types of data. Software designers prefer schemes that make common operations easy to perform.

Imagine that you are responsible for creating your own system that represents each character as a set of bits. You might decide to assign 0b00000000 to represent character A, and 0b00000001 to represent character B, and so on. This process of translating data into a digital format is known as *encoding*; when you interpret that digital data, it's known as *decoding*.

EXERCISE 2-1: CREATE YOUR OWN SYSTEM FOR REPRESENTING TEXT

Define a way to represent the uppercase letters A through D as 8-bit numbers, and then encode the word *DAD* into 24 bits using your system. There's no single right answer to this; see Appendix A for an example answer. Bonus: show your encoded 24-bit number in hexadecimal too.

ASCII

Fortunately, we already have several standard ways to represent text digitally, so we don't have to invent our own! *American Standard Code for Information Interchange (ASCII)* is a format that represents 128 characters using 7 bits per character, although each character is commonly stored using a full byte, 8 bits. Using 8 bits instead of 7 just means we have an extra leading bit, left as 0. ASCII handles the characters needed for English, and another standard, called *Unicode*, handles characters used in nearly all languages, English included. For now, let's focus on ASCII to keep things simple. Table 2-1 shows the binary and hexadecimal values for a subset of ASCII characters. The first 32 characters aren't shown; they are control codes such as carriage return and form feed, originally intended for controlling devices rather than storing text.

EXERCISE 2-2: ENCODE AND DECODE ASCII

Using Table 2-1, encode the following words to ASCII binary and hexadecimal, using a byte for each character. Remember that there are different values for uppercase and lowercase letters.

- Hello
- 5 cats

Using Table 2-1, decode the following words. Each character is represented as an 8-bit ASCII value with spaces added for clarity.

- 01000011 01101111 01100110 01100110 01100101 01100101
- 01010011 01101000 01101111 01110000

Using Table 2-1, decode the following word. Each character is represented as an 8-bit hexadecimal value with spaces added for clarity.

- 43 6C 61 72 69 6E 65 74

Answers are in Appendix A.

Table 2-1: ASCII Characters 0x20 Through 0x7F

Binary	Hex	Char	Binary	Hex	Char	Binary	Hex	Char
00100000	20	[Space]	01000000	40	@	01100000	60	`
00100001	21	!	01000001	41	A	01100001	61	a
00100010	22	"	01000010	42	B	01100010	62	b
00100011	23	#	01000011	43	C	01100011	63	c
00100100	24	\$	01000100	44	D	01100100	64	d
00100101	25	%	01000101	45	E	01100101	65	e
00100110	26	&	01000110	46	F	01100110	66	f
00100111	27	'	01000111	47	G	01100111	67	g
00101000	28	(01001000	48	H	01101000	68	h
00101001	29)	01001001	49	I	01101001	69	i
00101010	2A	*	01001010	4A	J	01101010	6A	j
00101011	2B	+	01001011	4B	K	01101011	6B	k
00101100	2C	,	01001100	4C	L	01101100	6C	l
00101101	2D	-	01001101	4D	M	01101101	6D	m

Binary	Hex	Char	Binary	Hex	Char	Binary	Hex	Char
00101110	2E	.	01001110	4E	N	01101110	6E	n
00101111	2F	/	01001111	4F	O	01101111	6F	o
00110000	30	0	01010000	50	P	01110000	70	p
00110001	31	1	01010001	51	Q	01110001	71	q
00110010	32	2	01010010	52	R	01110010	72	r
00110011	33	3	01010011	53	S	01110011	73	s
00110100	34	4	01010100	54	T	01110100	74	t
00110101	35	5	01010101	55	U	01110101	75	u
00110110	36	6	01010110	56	V	01110110	76	v
00110111	37	7	01010111	57	W	01110111	77	w
00111000	38	8	01011000	58	X	01111000	78	x
00111001	39	9	01011001	59	Y	01111001	79	y
00111010	3A	:	01011010	5A	Z	01111010	7A	z
00111011	3B	;	01011011	5B	[01111011	7B	{
00111100	3C	<	01011100	5C	\	01111100	7C	
00111101	3D	=	01011101	5D]	01111101	7D	}
00111110	3E	>	01011110	5E	^	01111110	7E	~
00111111	3F	?	01011111	5F	_	01111111	7F	[Delete]

It's fairly straightforward to represent text in a digital format. A system like ASCII maps each character, or symbol, to a unique sequence of bits. A computing device then interprets that sequence of bits and displays the appropriate symbol to the user.

Digital Colors and Images

Now that we've seen how to represent numbers and text in binary, let's explore another type of data: color. Any computing device that has a color graphics display needs to have some system for describing colors. As you might expect, as with text, we already have standard ways of storing color data. We'll get to them, but first let's design our own system for digitally describing colors.

Let's limit our range of colors to black, white, and shades of gray. This limited set of colors is known as *grayscale*. Just like we did with text, let's begin by deciding how many unique shades of gray we want to represent. Let's keep it simple and go with black, white, dark gray, and light gray. That's four total grayscale colors, so how many bits do we need to represent four colors? Only 2 bits are needed. A 2-bit number can represent four unique values, since 2 raised to the power of 2 is 4.

EXERCISE 2-3: CREATE YOUR OWN SYSTEM FOR REPRESENTING GRayscale

Define a way to digitally represent black, white, dark gray, and light gray. There's no single right answer to this; see Appendix A for an example answer.

Once you've designed a system for representing shades of gray in binary, you can build on that approach and create your own system for describing a simple grayscale image. An image is essentially an arrangement of colors on a two-dimensional plane. Those colors are typically arranged in a grid composed of single-color squares called *pixels*. Here's a simple example in Figure 2-1.

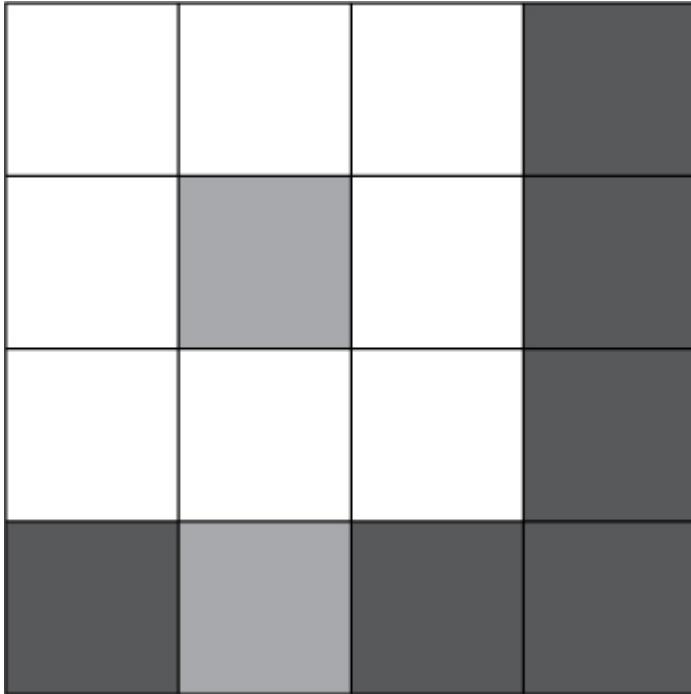


Figure 2-1: A simple image

The image in Figure 2-1 has a width of 4 pixels and a height of 4 pixels, giving it a total of 16 pixels. If you squint and use your imagination, you may see a white flower and a dark sky beyond. The image consists of only three colors: white, light gray, and dark gray.

NOTE

Figure 2-1 is composed of some really large pixels to illustrate a point. Modern televisions, computer monitors, and smartphone screens can also be thought of as a grid of pixels, but each pixel is very small. For example, a high definition display is typically 1920 pixels (width) by 1080 pixels (height), for a total of about 2 million pixels! As another example, digital photographs often contain more than 10 million pixels in a single image.

EXERCISE 2-4: CREATE YOUR OWN APPROACH FOR REPRESENTING SIMPLE IMAGES

Part 1 Building upon your previous system for representing grayscale colors, design an approach for representing an image composed of those colors. If you want to simplify things, you can assume that the image will always be 4 pixels by 4 pixels, like the one in Figure 2-1.

Part 2 Using your approach from part 1, write out a binary representation of the flower image in Figure 2-1.

Part 3 Explain your approach for representing images to a friend. Then give your friend your binary data and see if they can draw the image above without seeing the original image!

There's no single right answer to this; see Appendix A for an example answer.

In Exercise 2-4, in part 2, you acted like a computer program that was responsible for encoding an image into binary data. In part 3, your friend acted like a computer program that was responsible for the reverse, decoding binary data into an image. Hopefully she was able to decipher your binary data and draw a flower! If your friend pulled it off, then great, together you demonstrated how software encodes and decodes data! If things didn't go so well, and she ended up drawing something more like a pickle than a flower, that's okay too; you demonstrated how sometimes software has flaws, leading to unexpected results.

Approaches for Representing Colors and Images

As mentioned earlier, there are already standard approaches defined for representing colors and images in a digital manner. For grayscale images, one common approach is to use 8 bits per pixel, allowing for 256 shades of gray. Each pixel's value typically represents the intensity of light, so 0 represents no light intensity (black) and 255 represents full intensity (white), and values in between are varying shades of gray, from dark to light. Figure 2-2 illustrates various shades of gray using an 8-bit encoding scheme.

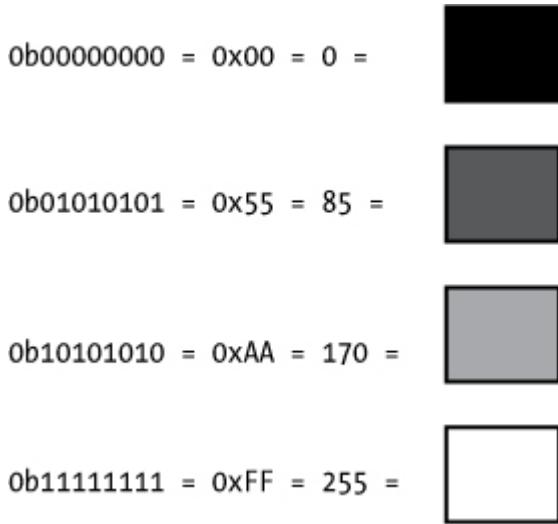


Figure 2-2: Shades of gray represented with 8 bits, shown as binary, hex, and decimal

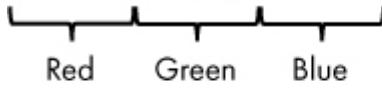
Representing colors beyond shades of gray works in a similar manner. Although grayscale can be represented with a single 8-bit number, an approach known as *RGB* uses three 8-bit numbers to represent the intensity of Red, Green, and Blue that combine to make a single color. Dedicating 8 bits to each of the three component colors means 24 bits are needed to represent the overall color.

NOTE

RGB is based on an additive color model, where colors are composed of a mix of red, green, and blue light. This is in contrast to the subtractive color model used in painting, where the mixed colors are red, yellow, and blue.

For example, the color red is represented in RGB with all 8 red bits set to 1, and the remaining 16 bits for the other two colors set to 0. Or if you wanted to represent yellow, which is a combination of red and green, but no blue, you could set the red and green bits to all 1s and leave the blue bits as all 0s. This is illustrated in Figure 2-3.

`11111111000000000000000000` = `0xFF0000` = Red



`111111111111111100000000` = `0xFFFF00` = Yellow



Figure 2-3: Red and yellow represented using RGB

In both the examples in Figure 2-3, the colors that are “on” are all 1s, but the RGB system allows for the red, blue, and green component colors to be partial strength as well. Each component color can vary from 00000000 (0 decimal/0 hex) to 11111111 (255 decimal/FF hex). A lower value represents a darker shade of that color, and a higher value represents a brighter shade of that color. With this flexibility of mixing colors, we can represent nearly any shade imaginable.

Not only are there standard ways of representing colors, but there are also multiple, commonly used approaches for representing an entire image. As you saw in Figure 2-1, we can construct images using a grid of pixels, with each pixel set to a particular color. Over the years, multiple image formats have been devised to do just that. A simplistic approach of representing an image is called a *bitmap*. Bitmap images store the RGB color data for each individual pixel. Other image formats, such as JPEG and PNG, use compression techniques to reduce the number of bytes required to store an image, as compared to a bitmap.

Interpreting Binary Data

Let’s examine one more binary value: `011000010110001001100011`. What do you think it represents? If we assume it is an ASCII text string, it represents “abc.” On the other hand, perhaps it represents a 24-bit RGB color, making it a shade of gray. Or maybe it is a positive integer, in which case it is 6,382,179 in decimal. These various interpretations are illustrated in Figure 2-4.

011000010110001001100011

abc
as ASCII

6,382,179
as a 32-bit integer



as an RGB color

Figure 2-4: Interpretations of 011000010110001001100011

So which is it? It can be any of these, or something else entirely. It all depends on the context in which the data is interpreted. A text editor program will assume the data is text, whereas an image viewer may assume it is the color of a pixel in an image, and a calculator may assume it is a number. Each program is written to expect data in a particular format, and so a single binary value has different meanings in various contexts.

We've demonstrated how binary data can be used to represent numbers, text, colors, and images. From this you can make some educated guesses about how other types of data can be stored, such as video or audio. There's no limit on what kinds of data can be represented digitally. The digital representation isn't always a perfect replica of the original data, but in many cases that isn't a problem. Being able to represent anything as a sequence of 0s and 1s is enormously useful, since once we've built a device that works with binary data we can adapt it, through software, to deal with any kind of data!

Binary Logic

We've established the utility of using binary to represent data, but computers do more than simply store data. They allow us to work with data as well. With a computer's help, we can read, edit, create, transform, share, and otherwise manipulate data. Computers give us the

capability to process data in many ways using hardware that we can program to execute a sequence of simple instructions—instructions like “add two numbers together” or “check if two values are equal.”

Computer processors that implement these instructions are fundamentally based on *binary logic*, a system for describing logical statements where variables can only be one of two values—true or false. Let’s now examine binary logic, and in the process, we’ll again see how everything in a computer comes down to 1s and 0s.

Let’s consider how binary is a natural fit for logic. Typically, when someone speaks of logic, they mean reasoning, or thinking through what is known in order to arrive at a valid conclusion. When presented with a set of facts, logic allows us to determine whether another related statement is also factual. Logic is all about truth—what is true, and what is false. Likewise, a bit can only be one of two values, 1 or 0. Therefore, a single bit can be used to represent a logical state of true (1) or false (0).

Let’s look at an example logical statement:

GIVEN a shape has four straight sides,
AND GIVEN the shape has four right angles,
I CONCLUDE that the shape is a rectangle.

This example has two conditions (four sides, four right angles) that must *both* be true for the conclusion to be true as well. For this kind of situation, we use the logical operator AND to join the two statements together. If either of the conditions is false, then the conclusion is false as well. I’ve expressed that same logic in Table 2-2.

Table 2-2: Logical Statement for a Rectangle

Four sides	Four right angles	Is a rectangle
False	False	False
False	True	False
True	False	False
True	True	True

Using Table 2-2, we can interpret each row as follows:

1. If the shape does *not* have four sides and does *not* have four right angles, it is *not* a rectangle.
2. If the shape does *not* have four sides and *does* have four right angles, it is *not* a rectangle.
3. If the shape *does* have four sides and does *not* have four right angles, it is *not* a rectangle.
4. If the shape *does* have four sides and *does* have four right angles, it *is* a rectangle!

This type of table is known as a *truth table*: a table that shows all the possible combinations of conditions (inputs) and their logical conclusions (outputs). Table 2-2 was written specifically for our statement about a rectangle, but really, the same table applies to any logical statement joined with AND.

In Table 2-3, I've made this table more generic, using A and B to represent our two input conditions, and Output to represent the logical result. Specifically, for this table Output is the result of A AND B.

Table 2-3: AND Truth Table (Using True and False)

A	B	Output
False	False	False
False	True	False
True	False	False
True	True	True

In Table 2-4, I've made one more modification to our table. Since this book is about computing, I've represented false as 0 and true as 1, just like computers do.

Table 2-4: AND Truth Table

A	B	Output
0	0	0
0	1	0
1	0	0

A	B	Output
1	1	1

Table 2-4 is the standard form of an AND truth table when you're dealing with digital systems that use 0 and 1. Computer engineers use such tables to express how components will behave when they're presented with a certain set of inputs. Now let's examine how this works with other logical operators and more complex logical statements.

Let's say you work at a shop that gives a discount to only two types of customers: children and people wearing sunglasses. No one else is eligible for a discount. If you wanted to state the store's policy as a logical expression, you could say the following:

GIVEN the customer is a child,
OR GIVEN the customer is wearing sunglasses,
I CONCLUDE that the customer is eligible for a discount.

Here we have two conditions (child, wearing sunglasses) where *at least one* condition must be true for the conclusion to be true. In this situation we use the logical operator OR to join the two statements together. If either condition is true, then the conclusion is true as well. We can express this as a truth table, as shown in Table 2-5.

Table 2-5: OR Truth Table

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

By observing the inputs and output in Table 2-5, we can quickly see that a discount will be given (Output = 1) when either the customer is a child (A = 1) or the customer is wearing sunglasses (B = 1). Note that the input column values for A and B are exactly the same for both Table 2-4 and Table 2-5. This makes sense, because both tables have two

inputs and thus the same possible set of input combinations. What differs is the Output column.

Let's combine AND with OR in a more complex logical statement. For the purposes of this example, assume that I go to the beach every day that is sunny and warm, and also assume that I go to the beach every year on my birthday. In fact, I only and always go to the beach under these specific circumstances—my wife says I'm overly stubborn in this way. Combining those ideas gives us the following logical statement:

GIVEN it is sunny **AND** **GIVEN** it is warm,
OR **GIVEN** it is my birthday,
I CONCLUDE that I am going to the beach.

Let's label our input conditions, then write a truth table for this expression.

Condition A It is sunny.

Condition B It is warm.

Condition C It is my birthday.

Our logical expression will look like this:

(A AND B) OR C

Just like in an algebraic expression, the parentheses around A AND B mean that part of the expression should be evaluated first. Table 2-6 gives us a truth table for this logical expression.

Table 2-6: (A AND B) OR C Truth Table

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1

A	B	C	Output
1	1	1	1

Table 2-6 is a bit more complex than a simple AND truth table, but it's still understandable. The table format makes it easy to look up a certain condition and see the outcome. For example, the third row tells us that if A = 0 (it is *not* sunny), B = 1 (it *is* warm), C = 0 (it is *not* my birthday), then Output = 0 (I'm *not* going to the beach today).

This kind of logic is something that computers regularly need to handle. In fact, as mentioned earlier, the fundamental capabilities of a computer distill down to sets of logical operations. Although a simple AND operator may seem far removed from the capabilities of a smartphone or laptop, these logical operators serve as the conceptual building blocks of all digital computers.

EXERCISE 2-5: WRITE A TRUTH TABLE FOR A LOGICAL EXPRESSION

Table 2-7 shows three inputs for a logical expression. Complete the truth table output for the expression (A OR B) AND C. The answer is in Appendix A.

Table 2-7: (A OR B) AND C Truth Table

A	B	C	Output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Besides AND and OR, several other common logical operators are used in the design of digital systems. I cover each operator in the following pages and provide a truth table for each. We use these again in Chapter 4 on digital circuits.

The logical operator NOT is just what it sounds like, the output is the opposite of the input condition. That is, if A is true, then the output is *not* true, and vice versa. As you can see in Table 2-8, NOT only takes a single input, rather than two inputs.

Table 2-8: NOT Truth Table

A	Output
0	1
1	0

The operator NAND means NOT AND, so the output is the reverse of AND. If both inputs are true, the result is false. Otherwise, the result is true. This is shown in Table 2-9.

Table 2-9: NAND Truth Table

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

The NOR operator means NOT OR, so the output is the reverse of OR. If both inputs are false, the result is true. Otherwise, the result is false. Table 2-10 shows this as a truth table.

Table 2-10: NOR Truth Table

A	B	Output
0	0	1
0	1	0
1	0	0

A	B	Output
1	1	0

XOR is Exclusive OR, meaning that only a single (exclusive) input can be true for the result to be true. That is, the output is true if only A is true or only B is true, while the output is false if both inputs are true. This is detailed in Table 2-11.

Table 2-11: XOR Truth Table

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

The study of logical functions of two-value variables (true or false) is known as *Boolean algebra* or *Boolean logic*. George Boole described this approach to logic in the 1800s, well before the advent of the digital computer. His work proved to be foundational to the development of digital electronics, including the computer.

Summary

In this chapter, we covered how binary is used to represent both data and logical states. You learned how 0s and 1s can be used to represent nearly any kind of data. We looked at text, colors, and images as examples of data in a binary format. You were introduced to various logical operators, such as AND and OR, and you learned about using truth tables to express a logical statement. Understanding this is important because the complex processors found in today's computers are based on an intricate system of logic.

We return to the topic of binary when we discuss digital circuits in Chapter 4, but first, to prepare you for that topic, we will take a detour in Chapter 3 to cover the fundamentals of electrical circuits. We'll

explore the laws of electricity, see how electrical circuits work, and get familiar with some basic components found in many circuits. You'll even have an opportunity to build your own circuits!

3

ELECTRICAL CIRCUITS



We've covered certain aspects of computing at a conceptual level; now let's change direction and take a look at the physical foundations of computing. Let's begin by reviewing our definition of a computer. A computer is an electronic device that can be programmed to carry out a set of logical instructions.

Computers are devices that follow rules designed by humans, but ultimately computers must behave according to another set of rules: the laws of nature. A computer is just a machine, and like all machines, it takes advantage of natural laws to accomplish a task. In particular, modern computers are electronic devices, so the laws of electricity are the natural foundation upon which these devices are built. To gain a well-rounded understanding of computing, you need to have a grasp of electricity and electrical circuits; that's what we cover in this chapter. Let's begin with electrical terms and concepts, learn some laws of electricity, cover circuit diagrams, and finally construct some simple circuits.

Electrical Terms Defined

For us to discuss electrical circuits, you first need to become familiar with several key concepts and terms. I'll now cover these concepts of electricity and explain how they relate to each other. This section is packed with details, so let's begin with an overview, in Table 3-1, before jumping into the particulars.

Table 3-1: Summary of Key Electrical Terms

Term	Explanation	Measured in	Water analogy
Electric charge	Causes matter to experience a force	Coulombs	Water
Electric current	The flow of charge	Amps	Flow of water through a pipe
Voltage	The difference in electric potential between two points	Volts	Water pressure
Resistance	A measure of the difficulty for current to flow through a material	Ohms	The width of a pipe

Table 3-1 gives a simple explanation of each term, lists its unit of measurement, and relates each term to its analog in a water-based system (shown in Figure 3-2, later in this chapter). If this doesn't immediately make sense, don't worry! We cover each term in greater depth in the following pages.

Electric Charge

In school, you may have learned that atoms are composed of positively charged protons, negatively charged electrons, and neutrons with no charge. *Electric charge* causes matter to experience a force; charges that

are not alike attract, whereas like charges repel. When explaining the concepts of electrical circuits, I like to use an analogy of water flowing through a pipe. In this analogy, electric charge is like water, and a wire is like a pipe.

The unit of measurement for charge is the *coulomb*. The charge on a proton or electron is a tiny fraction of the quantity of charge represented by a single coulomb.

Electric Current

Of particular relevance to our discussion is the transfer or movement of electric charge, known as *electric current*. This flow of charge through a wire is similar to the flow of water through a pipe. In everyday usage, including in this book, we say that “current is flowing” or that “current flows,” although more precisely it’s charge that flows, and current is a measurement of the intensity of that flow.

When representing current in an equation, the symbol I or i is used. Current is measured in *amperes*, sometimes just called *amps*, abbreviated as A . One ampere corresponds to one coulomb per second. Let’s say you have two wires, the first with $5A$ flowing through it, and the second with $1A$ flowing through it. Since amps represent a rate, the first wire has electric charge moving through it at five times the rate of the second wire.

Voltage

Since charge flows through a wire like water through a pipe, let’s expand that analogy further. So far, we have water (electric charge), a simple pipe (a copper wire), and the rate of water flow (electric current). To that, let’s add a pump, connected to the pipe that moves water through the pipe. The greater the water pressure, the faster the water flows through the pipe. Relating this to electrical circuits, the water pump represents a *power source*, a source of electrical energy, such as a battery.

Water pressure in this analogy represents a new concept: *voltage*. Just as water pressure influences the rate at which water flows through the

pipe, voltage influences current—the rate at which charge flows through a circuit. To understand voltage, recall from science class that *potential energy*, measured in *joules*, is the capacity for doing work. In the case of electricity, *work* means moving a charge from one point to another. *Electric potential* is potential energy per unit of electric charge, measured in joules per coulomb. Voltage is defined as the difference in electric potential between two points. That is, voltage is the work required per coulomb to move a charge from one point to another.

When representing voltage in an equation, the symbol V or v is used. Voltage is measured in *volts*, also abbreviated as V . Voltage is always measured between two points, such as the positive and negative terminals of a battery. A *terminal*, in this context, means an electrical connection point. The higher the voltage, the greater the “pressure” to move charge through a circuit from one terminal to another, and therefore the higher the current when a voltage source is connected in a circuit. However, a voltage can exist even if no current is flowing. For example, a 9-volt battery has a voltage of 9V across its terminals even when it isn’t connected to anything.

Resistance

Let’s go back to our water analogy. Another factor that affects the water flow is the width of the pipe. A very wide pipe allows water to flow in an unrestricted manner, but a narrow pipe impedes the flow. If we apply this analogy to circuits, the pipe’s width represents *electrical resistance* in a circuit. Resistance is a measure of the difficulty for current to flow through a *conductor*—a material that allows the flow of current. The higher the resistance of a material, the harder it is for current to flow through it.

When representing resistance in an equation, the symbol R is used. Resistance is measured in *ohms*, abbreviated as Ω (the Greek letter omega). A copper wire has very low resistance; for our purposes, let’s treat it as if it has no resistance at all, meaning current can freely flow through it. An electrical component called a *resistor* is used in circuits to

introduce specific amounts of resistance where needed. See Figure 3-1 for a photo of a typical resistor.

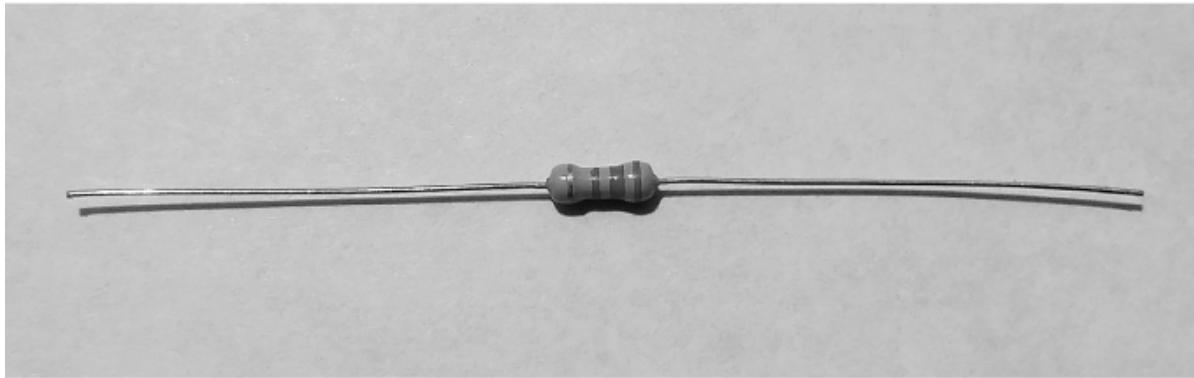


Figure 3-1: A resistor

Water Analogy

Now that we've covered key electrical concepts, let's revisit the water analogy we've been using to explain how electrical circuits work, as shown in Figure 3-2.

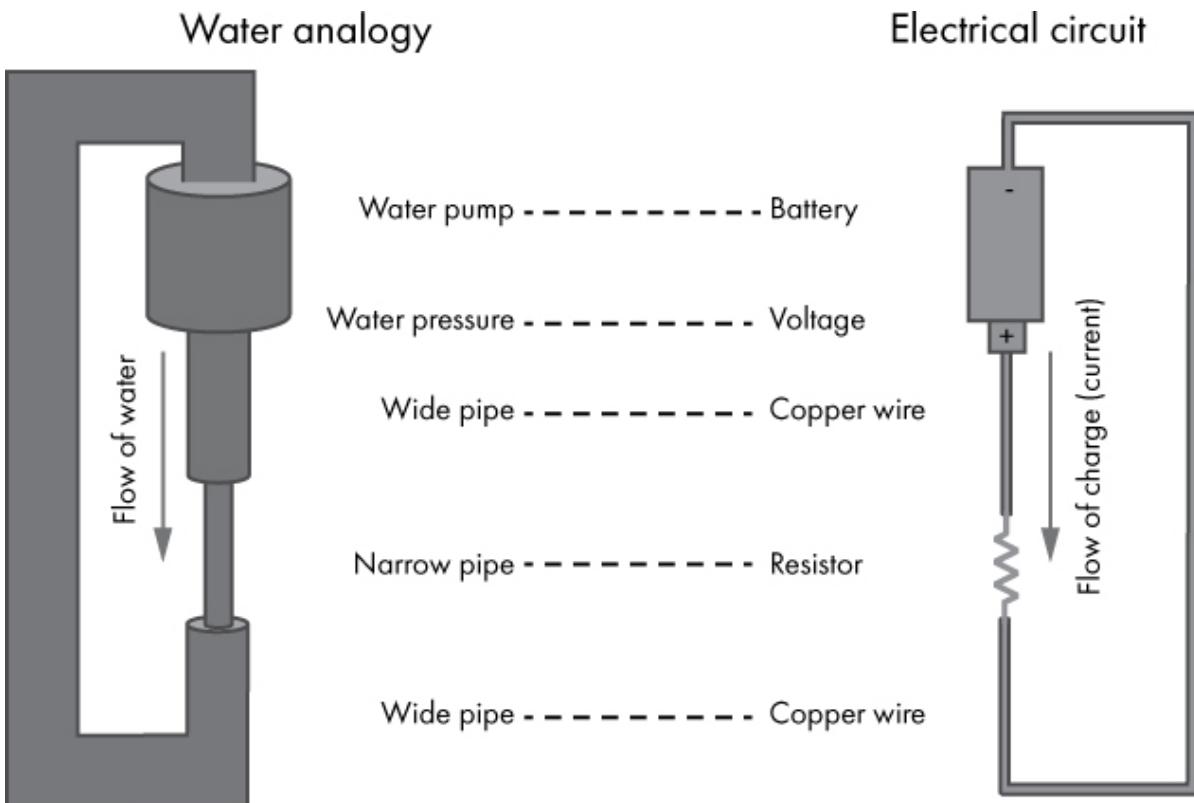


Figure 3-2: An electrical circuit described using a water analogy

The water pump moves water through a pipe, like a battery moves charge through a circuit. Just as water flows, so does electric charge; we call this current. Water pressure affects the rate at which water flows, and in the same way, voltage affects current—higher voltage means greater current. Water flows freely through a wide pipe, as current does through a copper wire. A narrow pipe restricts the flow of water, just as a resistor restricts current.

Now let's apply what you've learned to an example circuit consisting of a battery with a conductor attached to its terminals. Potential energy is stored in the battery. The voltage across the terminals of that battery is a certain number of volts and represents a difference in electric potential. When the conductor is attached to the terminals of the battery, the voltage acts like a pressure that moves charge through the conductor, creating a current. The conductor has a certain resistance; a low resistance results in a larger current, and a high resistance results in a smaller current.

Ohm's Law

The particulars of the relationship between current, voltage, and resistance is defined by *Ohm's law*, which tells us that the current flowing from one point to another is equal to the voltage across those points divided by the resistance between them. Or stated as an equation:

$$I = V / R$$

Let's say that you have a 9-volt battery with a $10,000\Omega$ resistor attached across its terminals. Ohm's law tells us that the current flowing through that resistor will be $9V / 10,000\Omega = 0.0009$ amps, or 0.9 milliamps (mA). See Table 1-2 for a reminder of why we use the “milli-” prefix here. Note that in this chapter, we're using base 10 again, so you can take off your base 2 hat and think about numbers like a normal human for a while!

AC AND DC

It is worth a brief pause to cover AC and DC. No, not the Australian rock band. *AC* stands for *alternating current*, an electric current that periodically changes direction. This is in contrast to *DC*, or *direct current*, where the current only flows in one direction. AC is used to transmit electricity from power plants to homes and businesses. Appliances, lamps, televisions, and other devices that plug directly into a wall socket without an adapter run on AC. Smaller electronics such as laptops and smartphones run on DC. When you charge a device like a smartphone, an adapter converts the AC from a wall socket to the DC that your device needs. Batteries also provide DC. The terms AC and DC are also applied to voltage (for example, a *DC voltage source*), in which case they essentially mean “alternating voltage” or “direct voltage.” All the circuits we deal with in this book are DC, so you don’t need to concern yourself with the details of AC, other than to know the distinction between the two.

Circuit Diagrams

When we are describing electrical circuits, a diagram can be a very helpful visual aid. Circuit diagrams are drawn using standard symbols to represent various circuit elements. The lines connecting those symbols represent wires. Let’s take a look at how some common circuit elements are represented in diagrams. Figure 3-3 shows the symbols for a resistor and a voltage source, such as a battery. The + indicates the positive terminal of the voltage source, and the – indicates the negative terminal. Said another way, the + terminal has a voltage that is positive in relation to the – terminal.

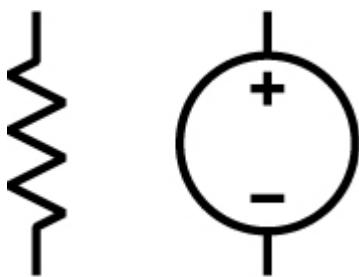


Figure 3-3: The symbols for a resistor (left) and voltage source (right)

Using these two symbols, we can draw a circuit diagram that represents our example circuit from earlier (a 9-volt battery with a $10,000\Omega$ resistor attached across its terminals). This diagram is shown in Figure 3-4. Note the $10k\Omega$ designator on the resistor; this is shorthand for $10,000\Omega$ (in other words, k is for kilo-, or thousand). Given our earlier Ohm's law calculation, we know that $0.9mA$ of current is flowing through the resistor, so that is indicated on the diagram as well.

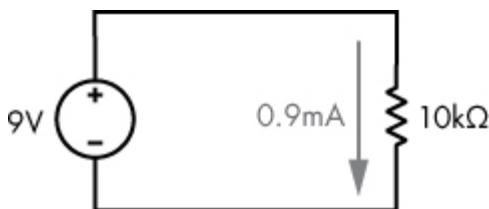


Figure 3-4: A 9-volt battery with a $10,000\Omega$ resistor attached across its terminals

We can also illustrate the current as a loop, as shown in Figure 3-5. This visual helps convey the idea that the current flows through the entire circuit, not just the resistor.

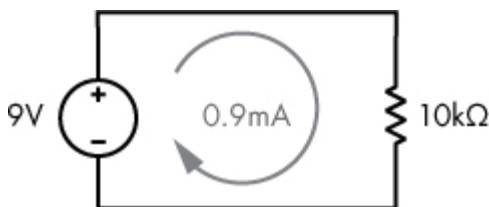


Figure 3-5: The flow of current illustrated as a loop

Speaking of loops, this is a good time to take a step back and revisit a term that has already been mentioned several times but I haven't yet defined. An *electrical circuit* is a set of electrical components connected in such a way that current flows in a loop, from the power source, through the circuit elements, and back to the source. Electricity aside, the general term *circuit* means a route that starts and finishes at the same place. This is an important concept to remember, because without a circuit, current will not flow. A circuit with a break in the loop is called an *open circuit*, and when a circuit is open, no current flows. On the

other hand, a *short circuit* is a path in a circuit that allows current to flow with little or no resistance, usually unintentionally.

EXERCISE 3-1: USING OHM'S LAW

Take a look at the circuit in Figure 3-6. What is the current, I ? The answer is in Appendix A.

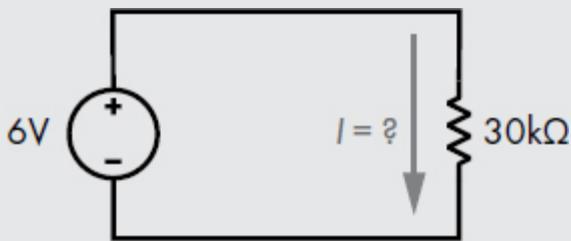


Figure 3-6: Find the current using Ohm's law.

In the context of DC circuits, *ground*, sometimes abbreviated GND, refers to a point we measure against other voltages in the circuit. In other words, ground is considered 0V, and we measure other voltages in the circuit relative to it. As we covered earlier, we always measure voltage between two points, so it's the difference in potential that matters, not the potential at a single point. By assigning ground as a reference point of 0V, we make it easier to talk about the relative voltage at other points in the circuit. In simple DC circuits like the ones we're discussing here, it's common for the negative terminal of the battery or other power source to be considered ground.

The term ground comes from the fact that some circuits are physically connected to the earth. They are literally connected to the ground, and that connection serves as a 0V reference point. Portable or battery-powered devices usually do not have a physical earth connection, but we still refer to a designated 0V reference point in those circuits as ground.

Sometimes engineers do not draw circuit diagrams as a loop; instead, they specifically identify the ground and voltage source

connections with the symbols shown in Figure 3-7. This makes for cleaner diagrams, but it doesn't change how the circuit is physically wired; current still flows from the positive to the negative terminal of the power source.

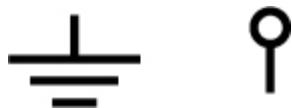


Figure 3-7: Symbols for ground (left) and a voltage source relative to ground (right)

As an example, in Figure 3-8 the circuit we discussed earlier is shown on the left, and on the right the same circuit is shown using the ground and voltage symbols introduced in Figure 3-7.

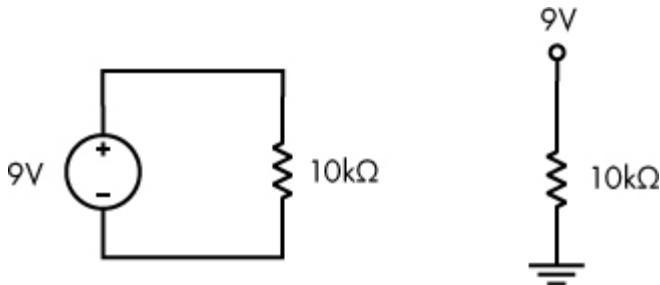


Figure 3-8: These two circuits are equivalent.

The two circuits in Figure 3-8 are functionally equivalent; only the diagram representation differs.

Kirchhoff's Voltage Law

Another principle that explains the behavior of circuits is *Kirchhoff's voltage law*, which tells us that the sum of the voltages around a circuit is zero. This means that if a voltage source supplies 9V to a circuit, then the various elements in that circuit must collectively "use" 9V. Each element around the circuit loop causes a decrease in electric potential. When this happens, we say that voltage is *dropped* across each element. Let's look at the circuit in Figure 3-9 as an example.

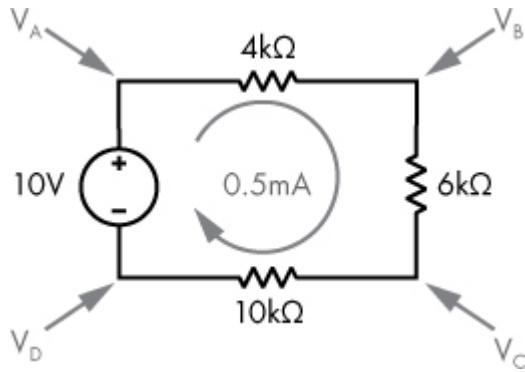


Figure 3-9: A circuit illustrating Kirchhoff's voltage law

In Figure 3-9 we have a 10-volt power supply connected to three resistors in a circuit. When resistors are connected along a single path (*in series*), the total resistance is simply the sum of the individual resistance values. In this case, that means that the total resistance is $4\text{k}\Omega + 6\text{k}\Omega + 10\text{k}\Omega = 20\text{k}\Omega$. Using Ohm's law, we can calculate the current through this circuit as $10\text{V} / 20\text{k}\Omega = 0.5\text{mA}$. The circuit has four points at which we could measure voltage, labeled as V_A , V_B , V_C , and V_D . We'll determine the voltage at each point relative to the negative terminal of our power supply.

Let's begin with the easy ones: V_D is directly connected to the negative terminal of the power supply, so $V_D = 0\text{V}$, or ground. Similarly, V_A is connected to the positive power terminal, so $V_A = 10\text{V}$. Now we know from Kirchhoff's voltage law that the resistors are each going to cause a voltage drop, so V_B must be something less than 10V, and V_C must be something less than V_B .

How much voltage is dropped across the $4\text{k}\Omega$ resistor? Ohm's law says that $V = I \times R$, so the voltage drop is $0.5\text{mA} \times 4\text{k}\Omega = 2\text{V}$. That means that V_B will be 2V less than V_A . So, $V_B = 10 - 2 = 8\text{V}$. Similarly, the voltage drop across the $6\text{k}\Omega$ resistor will be 3V. Therefore, $V_C = 8 - 3 = 5\text{V}$. Without even doing the math, we know from Kirchhoff's voltage law that 5V must be dropped across the $10\text{k}\Omega$ resistor, since it's the final circuit element before the negative terminal, at 0V. In Figure 3-10, our diagram is updated with voltages and voltage drops.

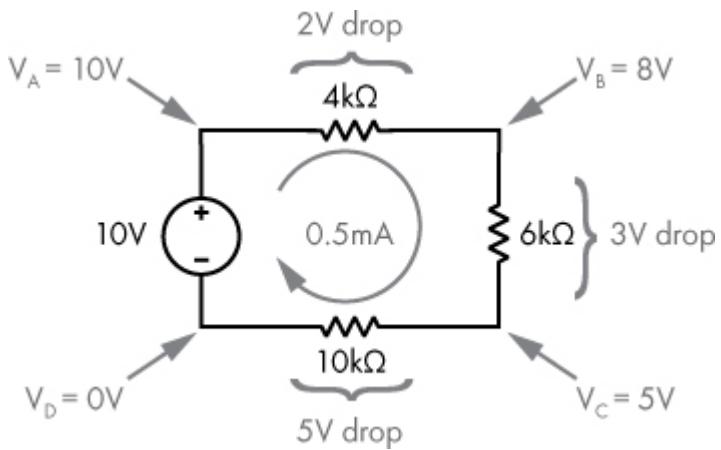


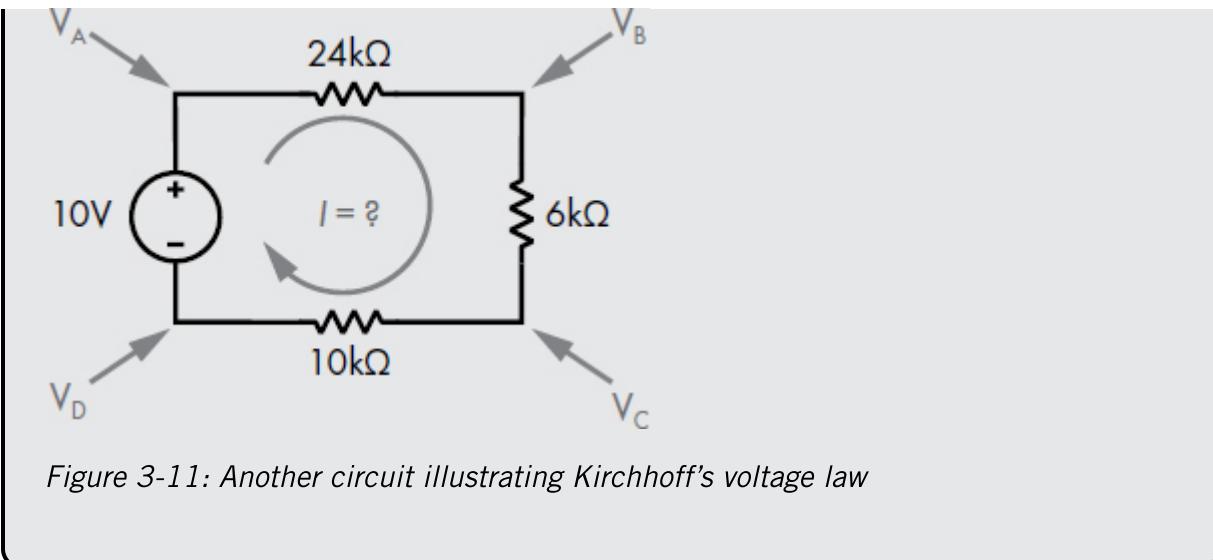
Figure 3-10: Voltage drops around a simple circuit

To recap this example, a voltage source supplies 10V, which we consider a positive voltage. The resistors each cause a drop in voltage, and we consider these voltage drops negative. If we sum the positive voltage source with the negative voltage drops, we get $10V - 2V - 3V - 5V = 0V$. The sum of the voltages around the circuit is 0, which is consistent with Kirchhoff's voltage law.

You may wonder if this only works with certain values of resistors. After all, the math worked out very nicely in the example given, perhaps a little too nicely! In the following exercise, we change one of the resistors in the example circuit from $4\text{k}\Omega$ to $24\text{k}\Omega$, and you can see that Kirchhoff's voltage law still holds true.

EXERCISE 3-2: FIND THE VOLTAGE DROPS

Given the circuit in Figure 3-11, what is the current, I ? What is the voltage drop across each resistor? Find the labeled voltages: V_A , V_B , V_C , and V_D , each measured as relative to the negative terminal of the power supply. The answer is in Appendix A.



Circuits in the Real World

Let's look at how our simple 9V/10k Ω circuit (from Figure 3-4) can be constructed in the real world. In the photo in Figure 3-12, I've attached a 10k Ω resistor to a 9-volt battery using alligator clips.

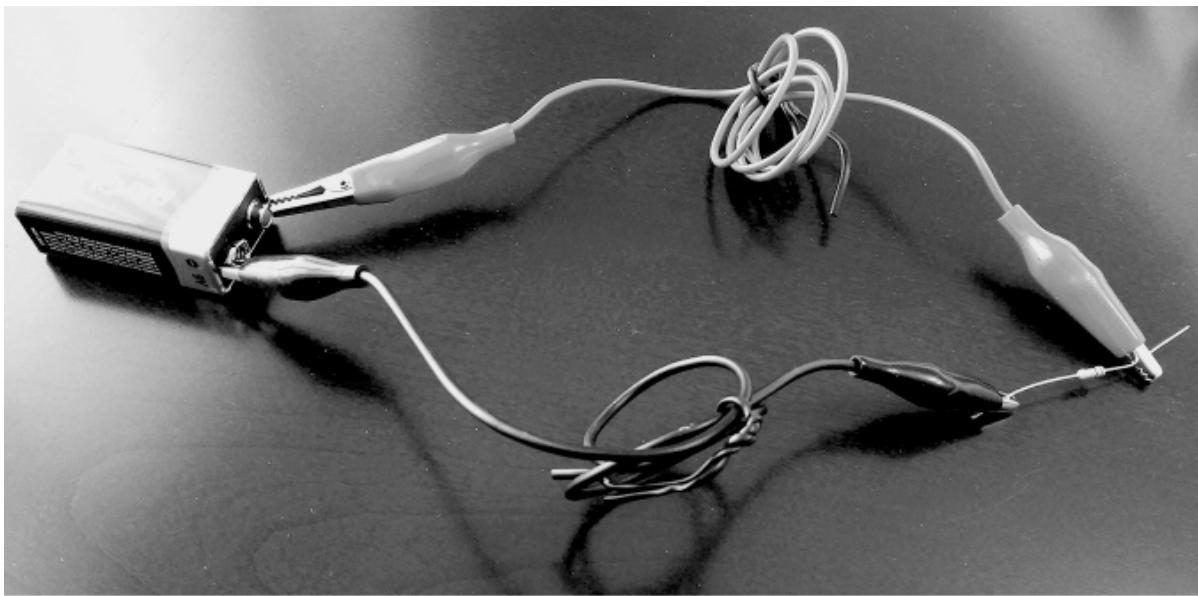


Figure 3-12: A 10k Ω resistor attached to a 9-volt battery using alligator clips

This approach works, but there are better ways to build circuits. A *breadboard* is a base for prototyping circuits. Historically, boards like

those used for baking bread were used for this purpose, but unfortunately today's breadboards don't have anything to do with bread! A breadboard (Figure 3-13) allows for easy connections of various electrical components.

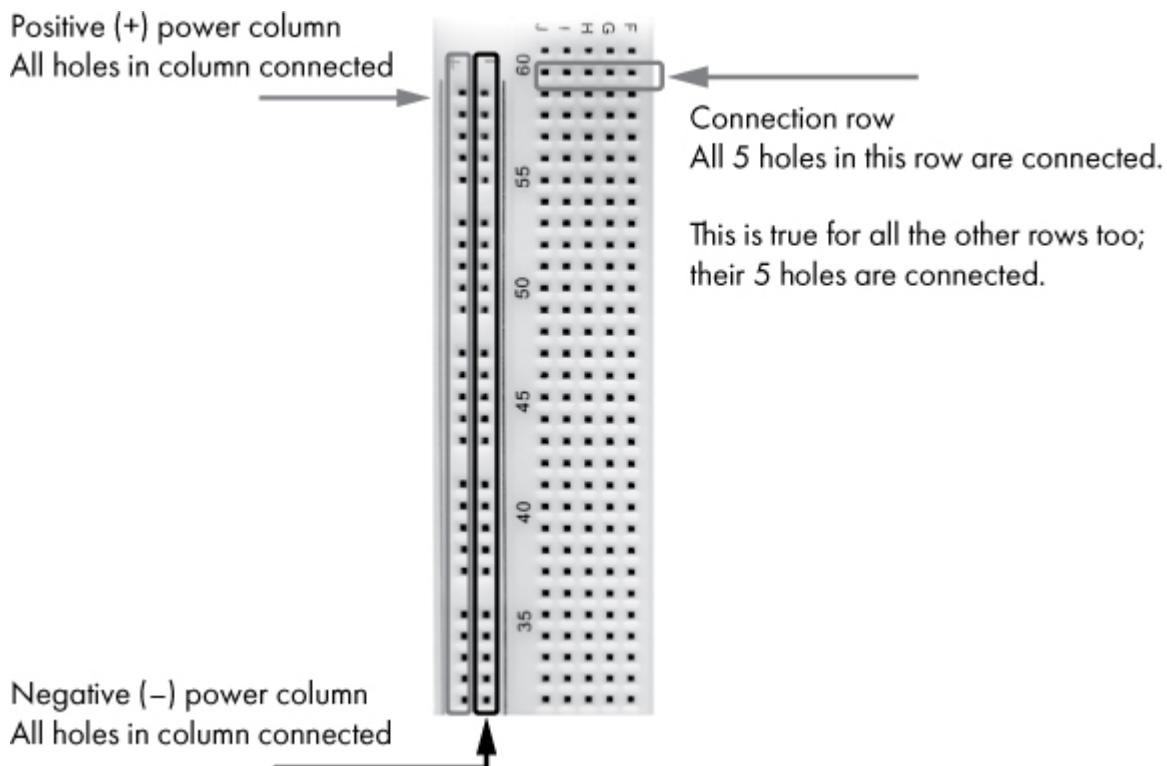


Figure 3-13: A section of a breadboard

As shown in Figure 3-13, along the edges of the breadboard are long columns usually labeled with + and -. These columns are often color coded as well, red for positive and blue for negative. All the holes along such an edge column are electrically connected, and the columns are intended to be used to provide power to the circuit, so typically your battery or other power source is connected to these columns. Likewise, the rows of typically five holes (in Figure 3-13, these are to the right of the edge columns) are also connected. Two components can be connected by simply putting an end of each component in the same row. No soldering, clips, or electrical tape required!

Figure 3-14 is a photo of the circuit shown in Figure 3-4 built on a breadboard.

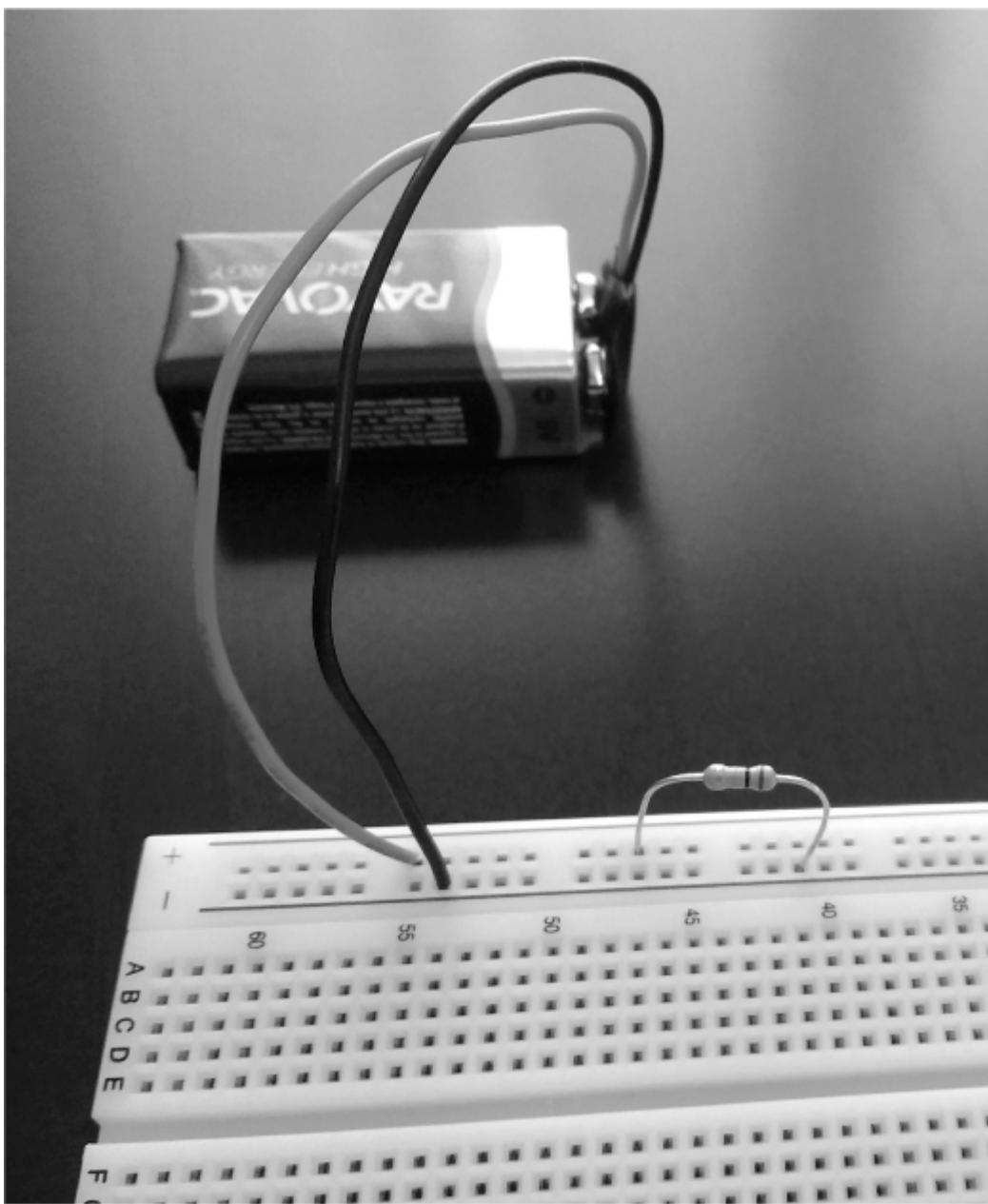


Figure 3-14: A simple circuit built on a breadboard

As you can see, this is a neater and simpler way of connecting electrical components. I optimized a bit by putting the ends of the resistor into the power columns to connect it directly to the battery.

NOTE

Please see Project #1 on page 45 to do the first project in this book! The previous exercises asked you to work through a problem mentally, whereas the projects ask you to do a little more, and this includes acquiring some hardware. There's some effort and cost associated, to be sure, but I believe getting your

hands dirty, so to speak, is the best way to really understand the concepts covered in this book. Turn to the end of this chapter to find the projects section. There you can build your own circuit!

Light-Emitting Diodes

The simple circuits we've discussed so far illustrate the basics of current and voltage, but they don't do anything visually interesting. I've found that the simplest way to turn a dull circuit into a delightful circuit is to add a *light-emitting diode (LED)*. A photo of a typical LED is shown in Figure 3-15.

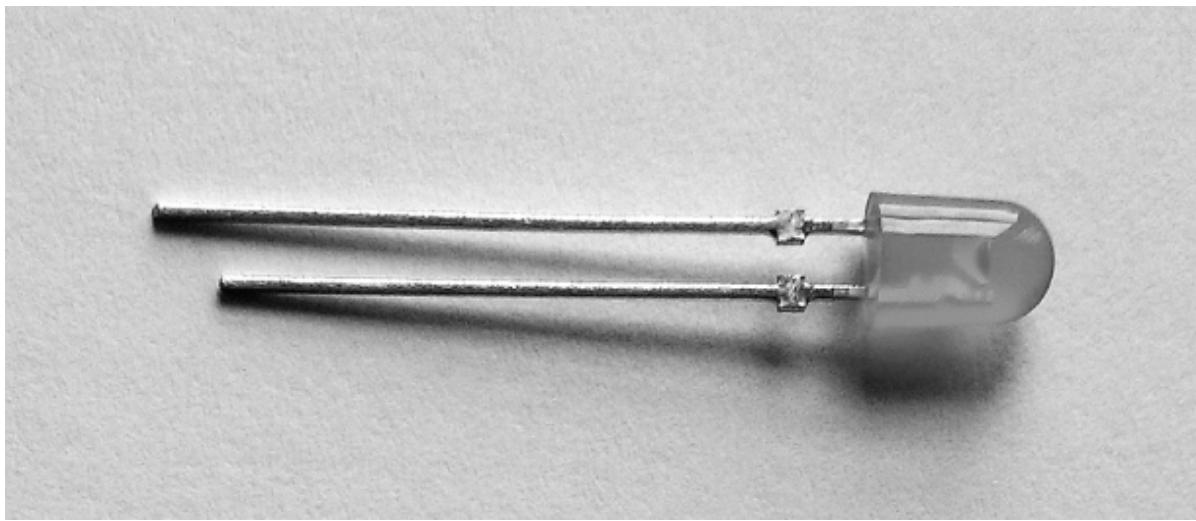


Figure 3-15: An LED

Let's cover LED basics and then add one to a circuit. The "light-emitting" part of the name is self-explanatory; this is a circuit element that emits light. Specifically, it's a diode that emits light. A *diode* is an electronic component that allows current to flow through it in only one direction. Unlike a resistor that allows current to flow in either direction, a diode has very low resistance in one direction (allowing current flow), and very high resistance in the other direction (impeding current flow). An LED is a special kind of diode that also lights up when current flows through it. LEDs are available in multiple colors. The circuit diagram symbol for an LED is shown in Figure 3-16.

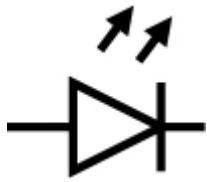


Figure 3-16: The symbol for an LED

To get an LED to emit light, we need to ensure the right amount of current flows through it. A standard red LED is rated for a maximum current of about 25mA; we don't want to exceed that maximum current, since doing so can damage the LED. Let's target 20mA as the amount of current we want to flow through our LED. A lower current value will still work, but the LED won't be as bright.

So how do we ensure that the right amount of current flows through our LED? We just need to choose an appropriate resistor to limit the amount of current in our circuit. But before we can do that, you need to know about one more property of the LED, *forward voltage*, which describes how much voltage is dropped across the LED when current flows through it. A typical red LED has a forward voltage of about 2V. Forward voltage is often notated as V_f .

Our circuit diagram with a battery, LED, and resistor to limit current is shown in Figure 3-17. The diagram also shows the desired current of 20mA.

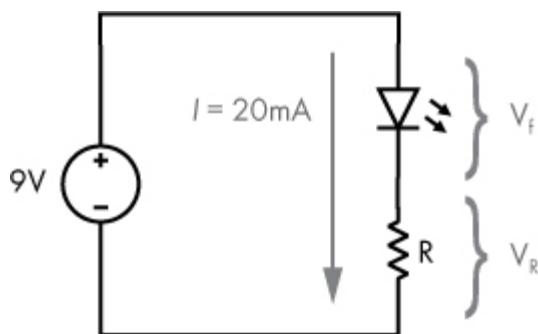


Figure 3-17: A basic circuit with an LED

In Figure 3-17, we have a 9-volt battery, an LED with a forward voltage of V_f , and a resistor with a resistance value of R . The voltage drop across the resistor is V_R . Keep in mind that the voltage drop across

the resistor varies with the amount of current that flows through it, unlike the LED, where the voltage drop is defined by its forward voltage property. In our previous circuit that only had a battery and a resistor (Figure 3-4), the entire 9V was dropped across the resistor. Now that we have two circuit elements connected to our battery, Kirchhoff's voltage law tells us that part of the voltage will drop across the LED and the rest of the voltage will drop across the resistor. As a reminder, you can think of the battery as supplying the voltage, and the other elements as using the voltage. If we apply this to our circuit (Figure 3-17), $V_f + V_R = 9V$.

Assuming we are using an LED with a standard forward voltage of 2V, $V_R = 9V - 2V = 7V$. Let's update our diagram with those voltage values, as shown in Figure 3-18.

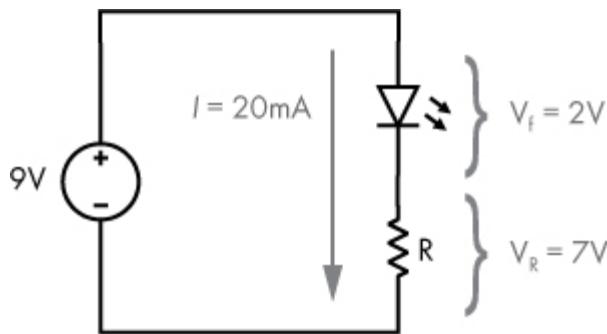


Figure 3-18: A basic circuit with an LED, voltage drops shown

That leaves us with only one unknown, R , the resistance of the resistor. We can calculate that using Ohm's law: $I = V / R$, or $R = V / I$. So that gives us $R = 7V / 20mA = 350\Omega$. And with that, we have the final piece of our puzzle on how to ensure that the right amount of current flows through our LED. We need a resistor of about 350Ω connected to our battery and LED.

NOTE

Please see Project #2 on page 50, where you can build an LED circuit yourself and see it light up!

Summary

In this chapter, we covered electrical circuits, the physical foundation of modern computing devices. You learned about electrical concepts such as charge, current, voltage, and resistance. You were introduced to two laws that govern the behavior of electrical circuits—Ohm’s law and Kirchhoff’s voltage law. You learned about circuit diagrams and how to construct your own circuits. Understanding the basics of electrical circuits will help you gain a foundational-level perspective on how computers work. The next chapter introduces digital circuits, bringing together the concepts of binary logic and electrical circuits.

PROJECT #1: BUILD AND MEASURE A CIRCUIT

You now know enough to build your own circuit. There’s no better way to learn than trying things yourself! To get started, you’re going to need some hardware, all of which you can purchase online or at a local electronics store, if you’re lucky enough to have one nearby. See “Buying Electronic Components” on page 333 for help getting these parts. Here’s what you’ll need for this project and the next:

- Breadboard (Either a 400-point or 830-point model)
- Resistors (An assortment of resistors. In this project you’ll specifically use a $10\text{k}\Omega$ resistor. Be sure to use a $10\text{k}\Omega$ resistor, not a 10Ω resistor. Using a resistor value that is too low will produce excessive current and may cause your circuit to get very hot!)
- Digital multimeter (You’ll need this to be able to check the voltage, current, and resistance of your circuit.)
- 9-volt battery
- 9-volt battery clip connector (This makes it much easier to connect your battery.)
- At least one 5mm or 3mm red LED (light emitting diode)
- Optional: Wire stripper (You may need one to strip away plastic from the ends of wires and expose the copper.)
- Optional: Alligator clips (These can make it easier to connect your battery to the breadboard or your multimeter to the circuit.)
- Optional: Breadboard jumper wires (Connect these to the ends of your 9-volt battery clip wires for easier insertion into a breadboard.)

Even at the low voltage levels we are working with, there is a small risk of a circuit component getting hot to the touch. With that in mind, I recommend that you

disconnect your power source, a battery in this case, while hooking things up, and only connect power as the final step of assembling your circuit.

Once you have your components, connect everything together:

1. Connect either end of a $10\text{k}\Omega$ resistor to the positive power column.
2. Connect the other end of the resistor to the negative power column.
3. Connect the red/positive wire of the battery clip to the positive power column on your breadboard.
4. Connect the black/negative wire of the battery clip to the negative power column on your breadboard.
5. Connect the battery clip to the terminals of the 9-volt battery.

The wiring of a 9-volt battery clip is sometimes flimsy, which makes it rather hard to insert into a breadboard. If you run into this problem, try connecting one end of a jumper wire to the flimsy battery wire and the other end to the breadboard. You can connect the two wires with electrical tape or alligator clips (see Figure 3-19), or you can even solder them together if you know how to use a soldering iron. If you try any of these methods, take care to keep the metal parts of the negative and positive wires separate. Accidentally connecting them will short circuit the battery, which will make the wires very warm and quickly drain the battery.

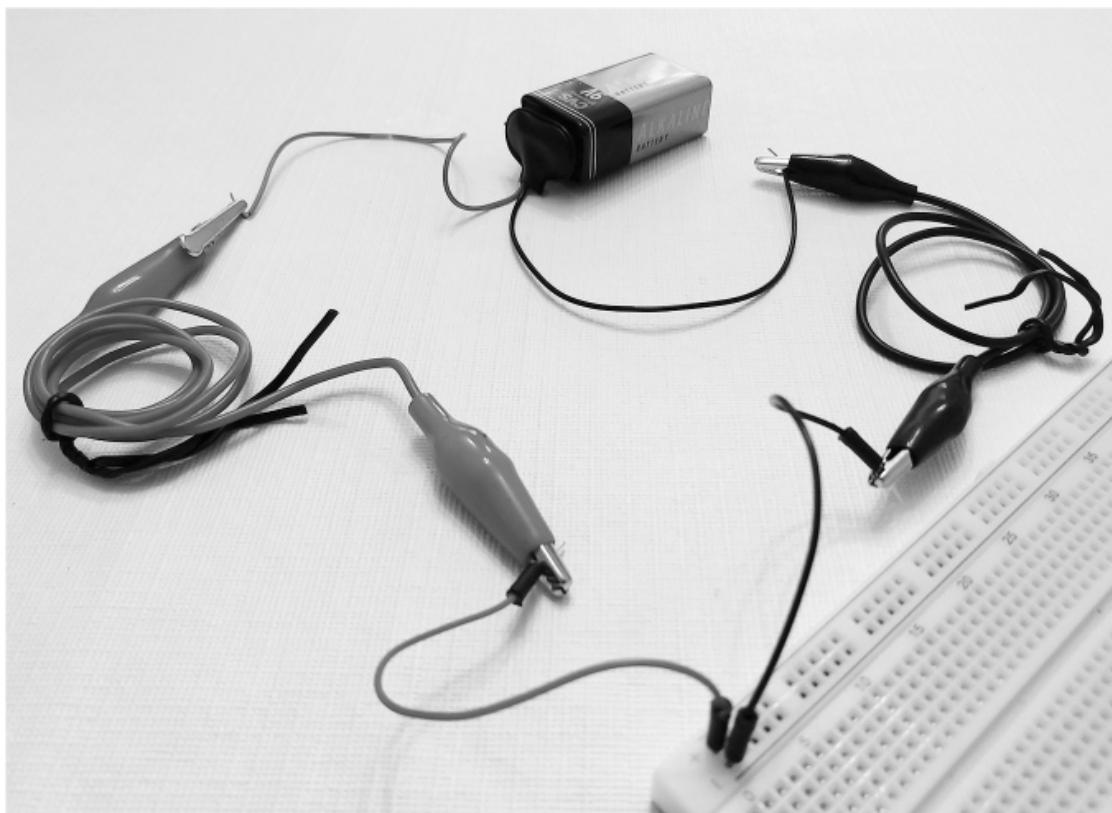


Figure 3-19: Using alligator clips and jumper wires to overcome flimsy 9-volt battery clip wiring

You may be wondering how to determine the ohm value of a resistor. Resistors are color-coded; the bands depict multipliers, and the colors represent numeric values. Many free resistor color code calculators and charts are online, so I won't go into the details here. For a $10\text{k}\Omega$ resistor, look for a resistor with brown, black, and orange bands, in that order. The fourth band will usually be gold or silver, indicating the manufacturer's tolerance, the allowed deviation from the stated value.

Now you have your circuit built, but how can you tell if anything is happening? Unfortunately, this circuit doesn't visually indicate that it's working, so it's time to break out the multimeter and measure its various properties. To use a multimeter, you'll need two test leads (the cables used for measurement). Unless your multimeter's test leads are hard-wired, it will probably have two or three input terminals for connecting test leads, as shown in Figure 3-20.

As shown in Figure 3-20, connect one lead to the input terminal labeled "COM" (meaning "common"). Conventionally we connect the black lead to the COM terminal. If your meter only has two input terminals, just connect the second lead, usually colored red, to the second terminal. Meters with three terminals typically have a COM input, a high-current input, and a low-current input. You'll want to connect your second lead to the low-current input in this case; it's typically labeled with the various measurement types it supports, something like $\text{V } \Omega \text{ mA}$. Usually the high-current input terminal is labeled as A, 10A, 10A max, or something along those lines. Again, that's not the terminal you want to use. Some multimeters have four inputs; if this is the case, you must use a different input terminal for low-current measurements versus voltage and resistance measurements. No matter what kind of meter you have, check the instruction manual if you have questions.

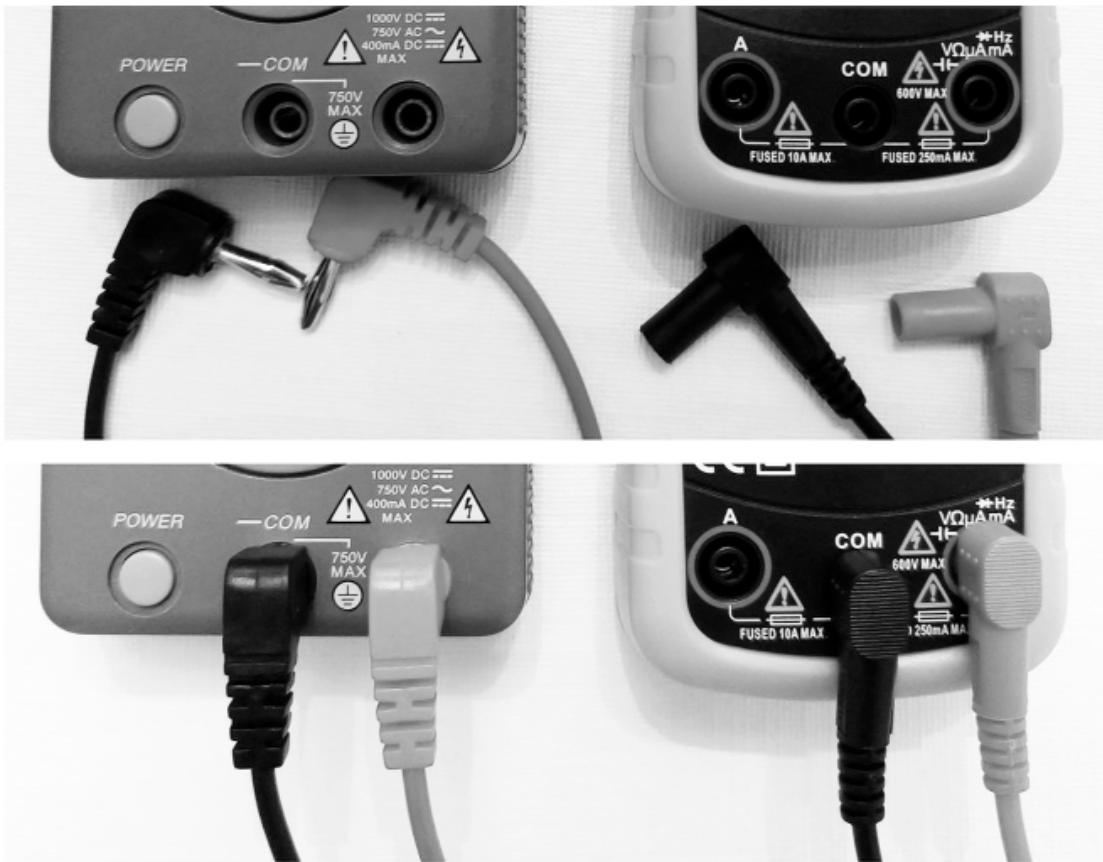


Figure 3-20: Connecting test leads to a multimeter. Meter with two inputs (left); meter with three inputs (right).

Your multimeter will provide a way to select whether you're measuring voltage, current, or resistance. Let's begin with voltage. Set your multimeter to measure voltage, DC. This may be indicated with V and the letters DC beside it, or you may see a series of dashes beside the V to indicate DC (versus a wavy line that indicates AC).

Once you have your multimeter set up to read DC voltage, measure the voltage across the resistor by touching one lead to the left side of the resistor (metal to metal) and the other lead to the right side. Keep in mind that voltage is always measured across two points, so it makes sense that we need to measure on both sides of the resistor. Since we are powering this circuit with a 9-volt battery, and the only circuit element present is the resistor, expect to measure approximately 9V. While you are measuring, you may notice that the value shown by the meter keeps changing a little (usually just the least significant digit, on the right). This is a result of the way digital meters work and does not mean your meter or your circuit is faulty.

Now try swapping the leads, putting the left lead on the right side of the resistor and vice versa. You should see the voltage value become negative (or positive if it was previously measuring negative). This is because the meter is measuring difference in potential and it treats the lead connected to COM as 0V; the measurement shown is the voltage difference at the other lead. In Figure 3-21, you can see the voltage measurement

of my circuit—9.56V. That sounds about right since a new battery often has slightly higher voltage than advertised.

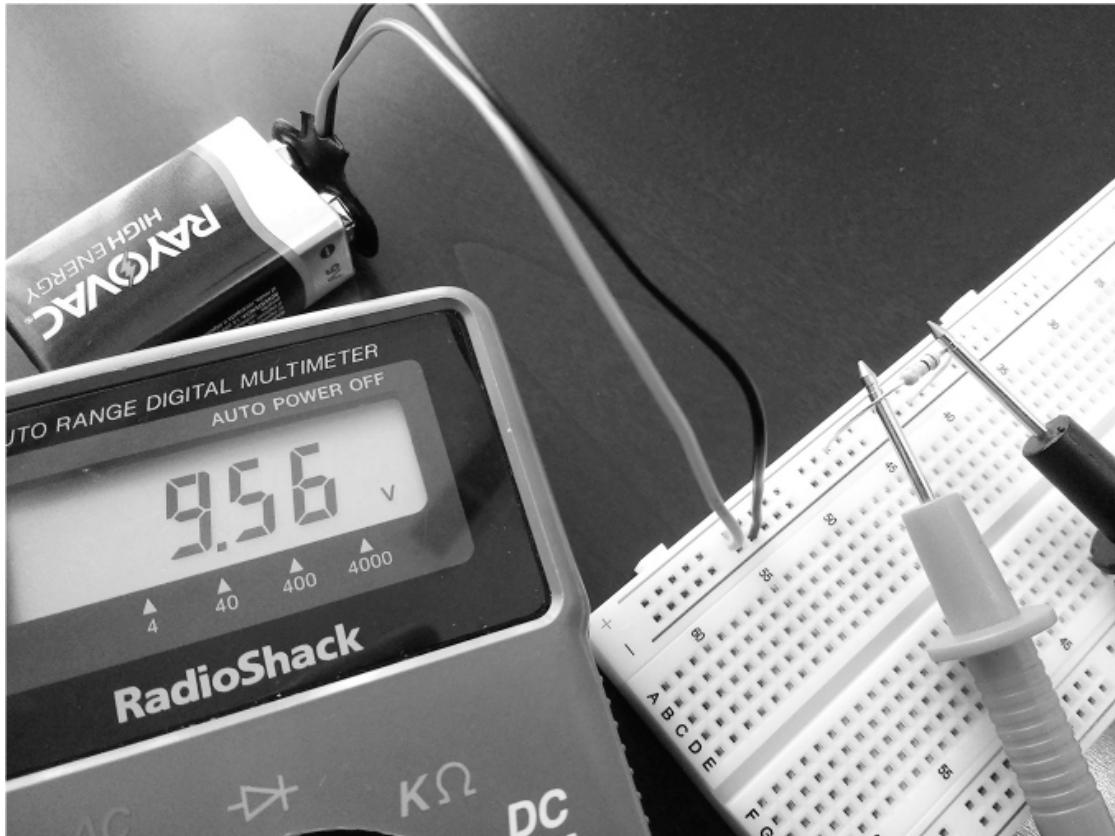


Figure 3-21: Measuring voltage

Next let's measure the resistance of the resistor. First, disconnect your multimeter from the circuit to prevent accidental damage when you change the setting. Then set your multimeter to the resistance setting, which may be marked as Ω . With nothing connected, your multimeter is likely to show a 1 on the left or OL. This means that the resistance value is too large to display—the resistance of air is very high! Touch the multimeter leads together and the display should show zero. To measure the resistance of the resistor, you'll want to disconnect it from the battery, but you can keep it attached to the breadboard if that is helpful. To ensure you see an accurate reading, avoid touching the circuit components and the metal of the leads while measuring; if you do, the resistance of your body may alter the value. You can see my resistance measurement in Figure 3-22; in my case the value was 9.88k Ω . The resistor I used had bands of brown, black, and orange (indicating 10,000 Ω) followed by a gold band (indicating 5 percent tolerance), so this measurement looks good. That is, 9.88k Ω is within 5 percent of 10k Ω .

At this stage, you know the measured values for both voltage and resistance, so you can calculate the expected current using Ohm's law—the current is voltage divided by resistance. For my circuit, that value will be $9.56V / 9,880\Omega = 0.97mA$. Before measuring the current in your circuit, perform that same calculation using your voltage and

resistance measurements so you can see how much current you should expect to find flowing through your circuit.

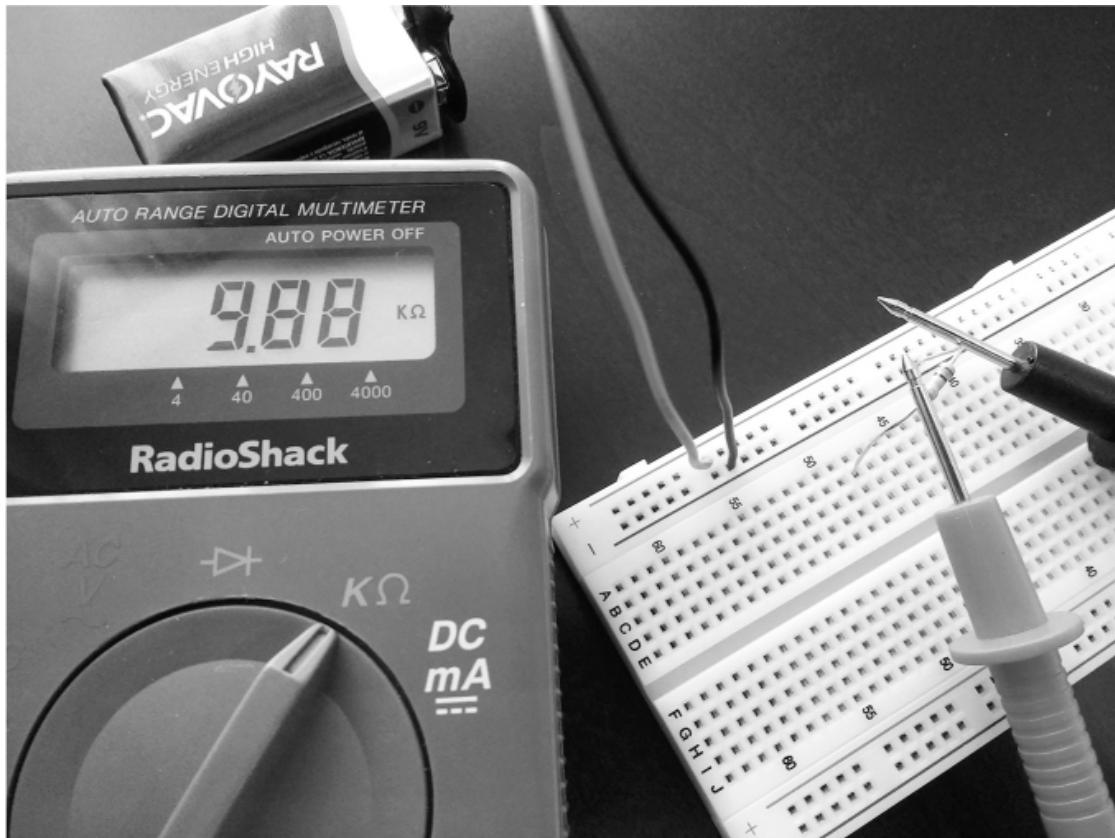


Figure 3-22: Measuring resistance

Now measure the current through your circuit and see how closely it matches your calculation. Measuring current is a bit different from measuring voltage or resistance. For the multimeter to be able to measure current, the current needs to flow through it. In other words, the meter must become part of the circuit, as illustrated in Figure 3-23.

Remember to disconnect your multimeter before you set it to measure DC current. The DC current symbols may be A or mA with DC or a series of dashes to indicate DC. Some meters have a single setting for measuring both DC and AC; in that case, the symbol may show both a straight line and a wavy line. Once your multimeter is on the correct setting, connect it to your circuit as shown in Figure 3-23.

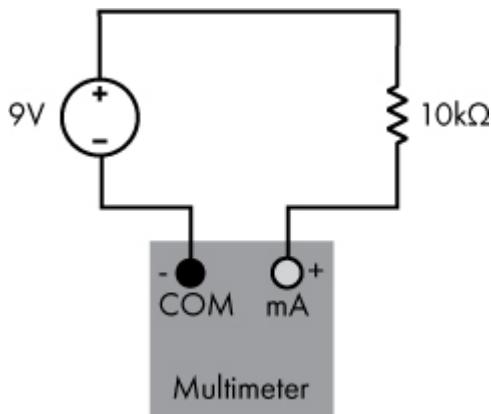


Figure 3-23: How to connect a multimeter when measuring current

Hopefully your measurement came close to your calculation. As shown in Figure 3-24, my current measurement was 0.97mA, which is the same as my earlier calculation.



Figure 3-24: Measuring current. The right side of the resistor and the black wire from the battery don't need to be connected to the breadboard.

If you've been following along to this point, then congratulations are in order! You've just built and measured an electrical circuit, or at the very least, you read about how I did it. And yet, you may feel a little underwhelmed. I understand. Honestly, aside from its

excellent educational value, this circuit is a bit boring and useless! In the next project, let's make something more interesting.

PROJECT #2: BUILD A SIMPLE LED CIRCUIT

Now it's your turn to build an LED circuit and see it light up! Assuming you did the previous project, you can keep the 9-volt battery clip hooked up to your breadboard, but let's swap out the $10\text{k}\Omega$ resistor for a more appropriately sized resistor for this circuit. From our calculations earlier in this chapter in the section entitled "Light-Emitting Diodes" on page 42, we found that we want a 350Ω resistor to deliver 20mA of current. If I look in my big variety pack of resistors, I'm not likely to find a 350Ω resistor, and neither are you. The closest resistor you are likely to find is 330Ω .

Now we could use more than one resistor here to get to exactly 350Ω , but is that necessary? Is 330Ω close enough? Let's find out. If we keep the 9-volt battery and still assume that we need to drop 7V across the resistor, then Ohm's law tells us that the current through our circuit will now be $I = V / R = 7\text{V} / 330\Omega = 21.2\text{mA}$. That will be just fine, since a typical red LED is rated for maximum current of about 25mA .

It's worth pointing out that any LEDs you purchased may have different characteristics from what I've described. If your LED came with a data sheet, or if it has specs listed online, check the specs and do your own calculations: What's the actual maximum or desired current for your specific LED? What's the actual forward voltage for your specific LED? Note that this usually varies by LED color.

You need to know one more thing about LEDs before building your circuit. Unlike a resistor, where current can flow either way, the LED is designed to allow current flow in only one direction, so you need to know which end is which. The positive side (the *anode*) usually has a longer lead, and the negative side (the *cathode*) has a shorter lead, as shown in Figure 3-25. Current flows from positive to negative.

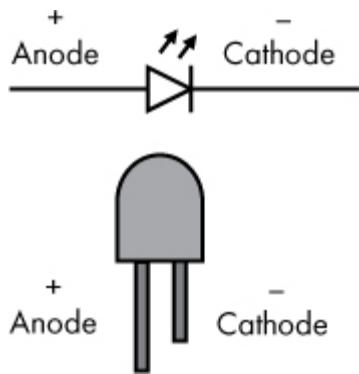


Figure 3-25: Identifying the anode and cathode on an LED

Once you have your numbers worked out and you know what value of resistor you want to use, connect things as follows, and as shown in the circuit diagram in Figure 3-26.

1. Temporarily disconnect the 9-volt battery from your breadboard.
2. Connect the longer lead on the LED to the positive power column.
3. Connect the shorter lead on the LED to any unused row on the breadboard.
4. Connect one end of the resistor to the same row as the shorter lead of the LED.
5. Connect the other end of the resistor to the negative power column.
6. Reconnect the 9-volt battery to your breadboard.

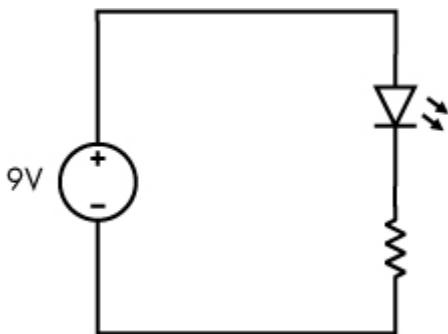


Figure 3-26: A simple LED circuit

Figure 3-27 is a photo of my LED, hooked up as described. Check out the glow! Hopefully your LED also lit up once the circuit was complete.

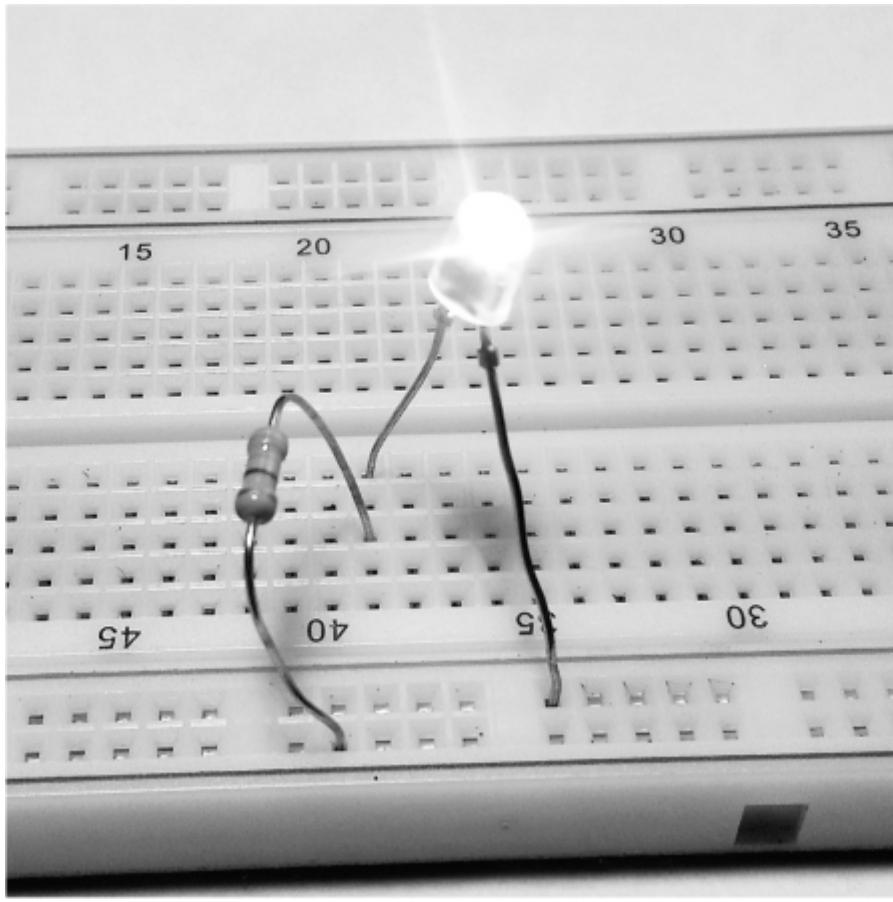


Figure 3-27: A glowing LED; attached battery not shown

4

DIGITAL CIRCUITS



Thus far, we've covered two aspects of computing. First, computers work in a binary system of 0s and 1s. Second, computers are electronic devices built upon electrical circuits. Now it's time to bring those two aspects of computing together. In this chapter we define what it means for an electrical circuit to be digital. We look at approaches for implementing digital circuits, including the role that transistors play. Finally, we examine logic gates and integrated circuits, the building blocks of more complex components that we'll cover in the chapters ahead.

What Is a Digital Circuit?

You may have noticed that the circuits we built in the previous chapter weren't digital—they were analog. In those circuits, voltage, current, and resistance could vary over a wide range of values. That isn't surprising; our world is naturally analog! However, computers work in the digital realm, and to understand computers we need to understand circuits that are digital. If we want our circuits to be digital, we must

first define what that means in the context of electronics, and then we can use analog components to build a digital circuit.

Digital circuits deal with signals that represent a limited number of states. This book deals with *binary* digital circuits, so 0 and 1 are the only two states to consider. We typically use voltage to represent a 0 or 1 in a digital circuit, where 0 is a low voltage and 1 is a high voltage. Usually low means 0V, and high tends to be 5V, 3.3V, or 1.8V, depending on the design of the circuit. In reality, digital circuits don't need a precise voltage to register as high or low. Instead, usually a range of voltages registers as high or low. For example, in a nominal 5-volt digital circuit, an input voltage of anywhere between 2V and 5V registers as high, and anywhere between 0V and 0.8V is considered low. Any other voltage level results in undefined behavior in the circuit and should be avoided.

Usually ground is the lowest voltage in a digital circuit, and all other voltages in that circuit are positive compared to ground. If a digital circuit is battery powered, we consider the negative terminal of the battery to be ground. The same goes for other kinds of DC power supplies; the negative terminal or wire is considered ground.

When referring to the 0 and 1 states in digital circuits, lots of terms and abbreviations get thrown around that all mean the same thing. These terms are often used interchangeably. Here are some common terms for low and high:

Low voltage Low, LO, off, ground, GND, false, zero, 0

High voltage High, HI, on, V+, true, one, 1

Logic with Mechanical Switches

Now that we've established that high and low voltages represent 1 and 0 in our digital circuit, let's consider how we can build a digital circuit. We want a circuit where the input and output voltages are always a predetermined high or low value, or at least within an allowable range.

To help us accomplish this, let's bring in a circuit element that's very simple and familiar: a mechanical switch. A *switch* is useful because it's digital in nature. It's on or off. When the switch is on, it acts like a simple copper wire through which current flows freely. When the switch is off, it acts like an open circuit and no current can flow. We represent a switch with the symbol shown in Figure 4-1.



Figure 4-1: The circuit diagram symbol for a switch—open/off (left), closed/on (right)

The switch symbol conveys the idea that a switch is an open circuit when in the off position, and the switch is a closed circuit when in the on position. You can think of the switch symbol, and a switch itself, as being like a fence gate that's either open or closed. Current flows through a switch when it's closed. In the real world, switches come in various shapes and sizes, as seen in Figure 4-2.

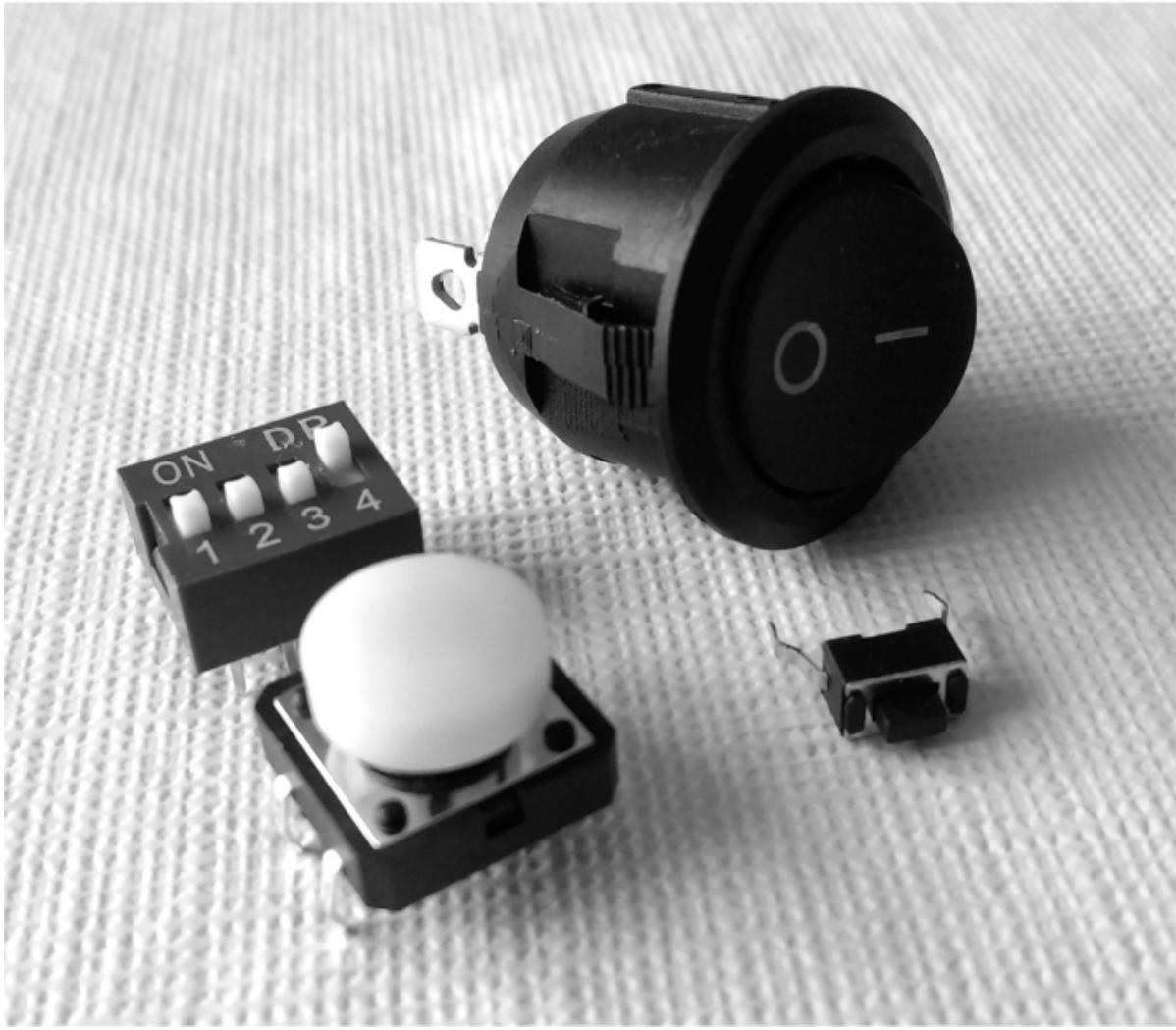


Figure 4-2: Some electrical switches

Note that in Figure 4-2, two of the switches, those closest to us, are *pushbuttons*, which you might not usually think of as switches.

Pushbuttons are also known as *momentary switches*, since the switch is closed only while the button is pressed. Removing pressure on the button opens the switch.

Now that we've introduced a circuit element that can easily turn on and off, let's use switches to build a digital circuit that acts like a logical AND operator. If you remember from Chapter 2, a two-input logical AND outputs a 1 if both inputs are 1, and a 0 otherwise. As a reminder, the truth table for AND is repeated in Table 4-1.

Table 4-1: AND Truth Table

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Let's now translate this to a circuit, with the following guidelines:

1. Inputs A and B are represented by mechanical switches. We represent 0 with an open switch, and we use a closed switch to represent 1.
2. The output is determined by voltage at a particular point in our circuit, called V_{out} .
3. If V_{out} is approximately 5V, the output is a logical 1, and if V_{out} is approximately 0V, the output is a logical 0.

Consider the circuit shown in Figure 4-3. This is a logical AND implemented with switches.

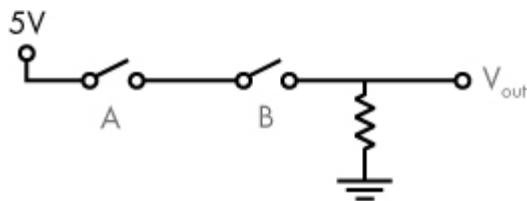


Figure 4-3: A logical AND implemented with switches

If either switch in Figure 4-3 is off (open/0), no current flows and the voltage at V_{out} is 0V. If both switches are on (closed/1), this makes a path to ground, current flows, and the voltage at V_{out} is 5V. In other words, if both A and B are 1, then the output is 1.

Let's take the same approach with a logical OR. The truth table for OR, first covered in Chapter 2, is shown in Table 4-2.

Table 4-2: OR Truth Table

A	B	Output
---	---	--------

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Look at the circuit shown in Figure 4-4, a logical OR implemented with switches.

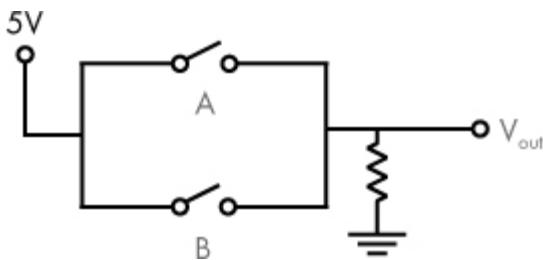


Figure 4-4: A logical OR implemented with switches

In Figure 4-4, when both switches are off (open/0), no current flows and the voltage at V_{out} is 0V, a logical 0. When either switch is turned on (closed/1), then current flows, and the voltage at V_{out} is 5V. In other words, if A or B is 1, then the output is 1.

The Amazing Transistor

In our quest to design digital circuits, the switch-based circuits just discussed are a good conceptual start. However, in computing devices, we can't practically use mechanical switches. The inputs to a computer are numerous, and flipping switches to control those inputs isn't a great design. Also, computing devices need to connect multiple logical circuits together—the output of one circuit needs to become the input of another. To accomplish this, our switches need to be electrically, rather than mechanically, controlled. We don't want a mechanical switch; we want an electronic switch. Fortunately, there's a circuit component that can act as an electronic switch: the transistor!

A *transistor* is a device used to switch or amplify current. For our purposes, let's focus on the switching capabilities of the transistor. The transistor is the basis of modern electronics, including computing devices. There are two main types of transistors: bipolar junction transistors (BJTs) and field-effect transistors (FETs). The differences between the two types aren't relevant for our discussion here; to keep things simple, let's focus on only one type: BJTs.

BJTs have three terminals: base, collector, and emitter. There are two types of BJTs: NPN and PNP. The two differ in the way they respond to the application of current to the base terminal. For our purposes, we focus on NPN BJTs. See Figure 4-5 for a circuit diagram and photo of an NPN transistor.



Figure 4-5: The symbol for (left) and photo of (right) an NPN transistor

In an NPN transistor, applying a small current at the base allows a larger current to flow from the collector to the emitter. In other words, if we think of the transistor as a switch, then applying current at the base is like turning the transistor on, and removing that current turns the transistor off. Let's look at how a transistor can be wired as an electronic switch, as shown in Figure 4-6.

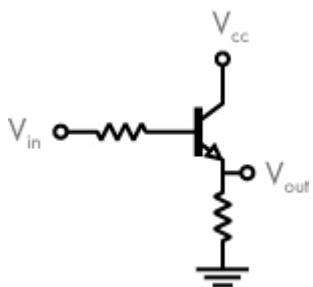


Figure 4-6: An NPN transistor as a switch

In Figure 4-6 an NPN transistor is connected to a couple of resistors with various labeled voltages. V_{cc} is a positive supply voltage applied to the collector terminal. This supplies power to our circuit. In case you are wondering, the “cc” in V_{cc} stands for “common collector,” and V_{cc} is the typical designation for the positive supply voltage in NPN-based circuits. V_{out} is the voltage we wish to control; we want this voltage to be high when our transistor-as-a-switch is on and low when our switch is off. V_{in} acts as the voltage that electrically controls our switch. Rather than flipping a mechanical device, we can use voltage V_{in} to control our switch.

Let’s consider what happens if we set V_{in} to either a low or high voltage. If V_{in} is low, connected to ground, then no current flows to the base of the transistor. With no current at the base, the transistor acts like an open circuit between the collector and emitter. That means that V_{out} is also low. Figure 4-7 illustrates this.

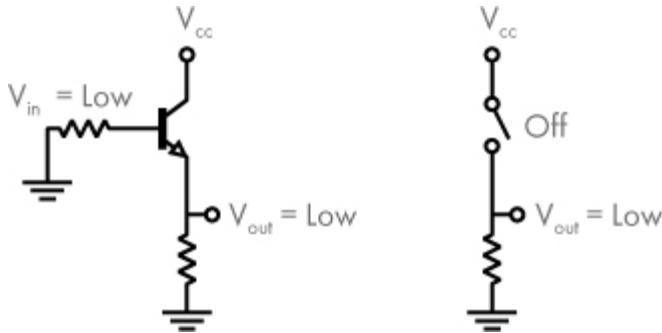


Figure 4-7: An NPN transistor as a switch in the off state

On the left in Figure 4-7 is the transistor-based circuit we’re discussing, and on the right is a switch-based circuit that represents the same state. In other words, the circuit on the left is effectively the same as the circuit on the right; I’ve just replaced the transistor with a switch to make it clear that the transistor, in this state, is like an open switch.

If V_{in} is low, then no current flows. On the other hand, if V_{in} is high, then a current flows to the base of the transistor. This current causes the transistor to conduct current from the collector to the emitter. That

means that V_{out} is effectively connected to V_{cc} and so the output is high, as shown in Figure 4-8.

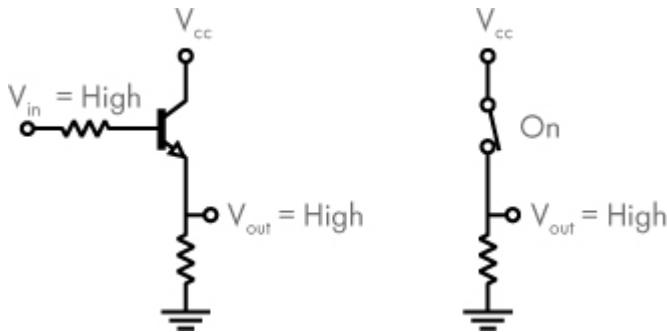


Figure 4-8: An NPN transistor as a switch in the on state

On the left in Figure 4-8 is the transistor-based circuit we're discussing, and on the right is a mechanical switch-based circuit with the same effective state. The transistor, in this state, is like a closed switch.

Logic Gates

Now that we've established that a transistor can act as an electrically controlled switch, we can build circuit elements that implement logical functions where the inputs and outputs are high and low voltages. Such components are known as *logic gates*. Let's begin by taking our earlier design for an AND circuit and replacing the mechanical switches with transistors. The advantage of this is that we can alter the input to the circuit just by changing a voltage; we don't need to flip a mechanical switch. Though mechanical switches are a good way for humans to interact with circuits, electronic switches allow multiple circuits to interact with each other—the output of one circuit can easily become the input of another.

Previously we built an AND circuit using mechanical switches (see Figure 4-3). Let's use transistors as switches to accomplish the same thing, as shown in Figure 4-9.

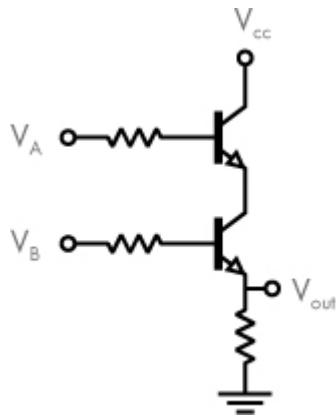


Figure 4-9: Logical AND implemented with transistors

In Figure 4-9, if V_A and V_B are high (a logical 1), then current flows through both transistors and V_{out} is also high (a logical 1). If V_A or V_B is low (a logical 0), then current doesn't flow and V_{out} is also low (a logical 0). This circuit implements a logical AND.

A similar approach can be used to implement a logical OR with transistors. I'll leave that as a design exercise and project for you to complete.

EXERCISE 4-1: DESIGN A LOGICAL OR WITH TRANSISTORS

Draw a circuit diagram for a logical OR circuit that uses transistors for inputs A and B. Adapt the circuit in Figure 4-4 that uses mechanical switches, but use NPN transistors instead. See Appendix A for a solution.

NOTE

Please see Project #3 on page 66, where you can build the circuits for logical AND and logical OR with transistors.

We've just seen how a logic gate, a circuit that implements a logical function, can be constructed with transistors and resistors. From this point forward, I am going to hide the details of how logic gates are implemented; instead we'll consider the entire gate as a single circuit element. This isn't just a theoretical way of looking at logic gates; it's

aligned with the way these circuit elements are used in the real world. Logic gates are available for purchase already assembled and physically packaged as a circuit component, so there's usually no need for you to build them yourself from transistors, except as a learning exercise. Standard circuit symbols are defined for the various logic gates. You can see some of the most common ones in Figure 4-10.

While reviewing the various logic gates in Figure 4-10, note the small circles added to the various symbols to represent NOT or inversion. A NOT gate is the simplest example of this; it takes a single input, and its output is the inversion of that input. So, 1 becomes 0, and 0 becomes 1. The output of a NAND gate is the same as NOT AND; its output is the inversion of the output of a regular AND gate. The same is true of NOR. You'll see the small circle crop up in other places in logic symbols to indicate NOT or inversion.

Before we move on, let's pause here and reflect on a certain aspect of what we just covered. First, we examined how a logic gate works internally. Then we took that design, wrapped it up in a package, and gave it a name and a symbol. We deliberately obscured the implementation of the circuit, while continuing to document its expected behavior. We put the design details of the logic gate into what is known as a *black box*, an element where inputs and outputs are known, but the internal details are hidden. Another term for this approach is *encapsulation*, a design choice that hides internal details of a component while documenting how to interact with that component.

Type	Symbol	Truth table															
AND		<table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Out</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	Out	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Out															
0	0	0															
0	1	0															
1	0	0															
1	1	1															
OR		<table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Out</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	A	B	Out	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Out															
0	0	0															
0	1	1															
1	0	1															
1	1	1															
NAND		<table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Out</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	Out	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Out															
0	0	1															
0	1	1															
1	0	1															
1	1	0															
NOR		<table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Out</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	Out	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Out															
0	0	1															
0	1	0															
1	0	0															
1	1	0															
NOT		<table border="1"> <thead> <tr> <th>A</th><th>Out</th></tr> </thead> <tbody> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </tbody> </table>	A	Out	0	1	1	0									
A	Out																
0	1																
1	0																
XOR		<table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Out</th></tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	A	B	Out	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Out															
0	0	0															
0	1	1															
1	0	1															
1	1	0															

Figure 4-10: Common logic gates

Encapsulation is a design principle that's found throughout modern technology. It's used when designers of a component want others to be able to easily use their creation and build upon it without needing to fully understand the details of its implementation. This approach also allows for improvements to be made inside "the box," and as long as the inputs and outputs continue to behave the same, the box can continue

to be used as it always has been. Another advantage of encapsulation is that a team can work collaboratively on a large project, with portions of the project encapsulated. This frees individuals from needing to understand every detail of every component. Encapsulating transistors in a logic gate is the first example of encapsulation in this book, but you'll see it multiple times as we proceed.

Designing with Logic Gates

In Chapter 2, we saw how multiple logical operators can be combined to create more complex logical statements. Let's now extend this idea to logic gates. Once a logical statement or truth table is written, that logic can be physically implemented in hardware using logic gates. Let's apply this principle to the truth table we previously created (Table 2-6) for the following statement:

IF it is sunny AND warm, OR it is my birthday, THEN I am going to the beach.

We simplified that to

(A AND B) OR C

Let's now represent this statement using a diagram with logic gates, as shown in Figure 4-11.

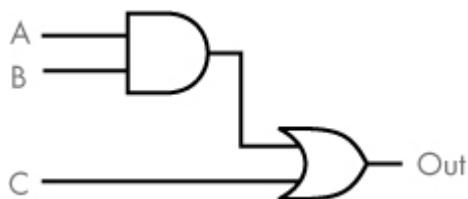


Figure 4-11: Logic gate diagram for $(A \text{ AND } B) \text{ OR } C$

If both A and B are 1, then the output of the AND gate will be 1. The output of the AND connects to the input of the OR gate, along with C. If either the AND output or C is 1, then the overall output will be 1.

When we combine logic gates in such a way that the output is a function of the present inputs, the circuit is known as a *combinational logic* circuit. That is, a certain set of present inputs will always produce the same output. This is in contrast to *sequential logic*, in which the output is a function of both present and past inputs. We'll cover sequential logic later in this book. For now, try your hand at designing a circuit that represents the logical expression described in Exercise 4-2.

EXERCISE 4-2: DESIGN A CIRCUIT WITH LOGIC GATES

In Chapter 2, Exercise 2-5, you created the truth table for (A OR B) AND C. Now build on that work and translate that truth table and logical expression into a circuit diagram. Draw a logic gate diagram (similar to the one in Figure 4-11) for the circuit using logic gates. See Appendix A for the answer.

Integrated Circuits

As mentioned previously, companies manufacture and sell ready-to-use digital logic gates. Hardware designers can buy these gates and start building their logic without worrying about how the gate circuits work internally. These logic gates are an example of an integrated circuit. An *integrated circuit (IC)* contains multiple components on a single piece of silicon in a package that has external electrical contact points, or *pins*. ICs are also known as *chips*.

In this book we primarily look at ICs that are packaged in a *dual in-line package (DIP)*, a rectangular enclosure with two parallel rows of pins. These pins are spaced so they can easily be used on a breadboard. Manufacturers build ICs from tiny transistors, much smaller than discrete transistors like the one shown earlier in Figure 4-5. A *discrete component* is an electronic device containing only a single element, such as a resistor or transistor. ICs result in small circuits that can operate faster and are less expensive than the same circuit built from discrete transistors.

The logic circuits we discussed earlier that used resistors and transistors are known as resistor-transistor logic (RTL) circuits. Manufacturers built early digital logic circuits this way, but later they used other approaches, including diode-transistor logic (DTL), and transistor-transistor logic (TTL). The 7400 series is the most popular line of TTL logic circuits. This line of integrated circuits includes logic gates and other digital components. Introduced in the 1960s, the 7400 line and its descendants are still a standard for digital circuits. I'm going to focus on the 7400 series to give you real-world examples of how integrated circuits are used.

Let's examine a specific 7400 series integrated circuit. The 7432 chip, shown in Figure 4-12, contains four OR gates.

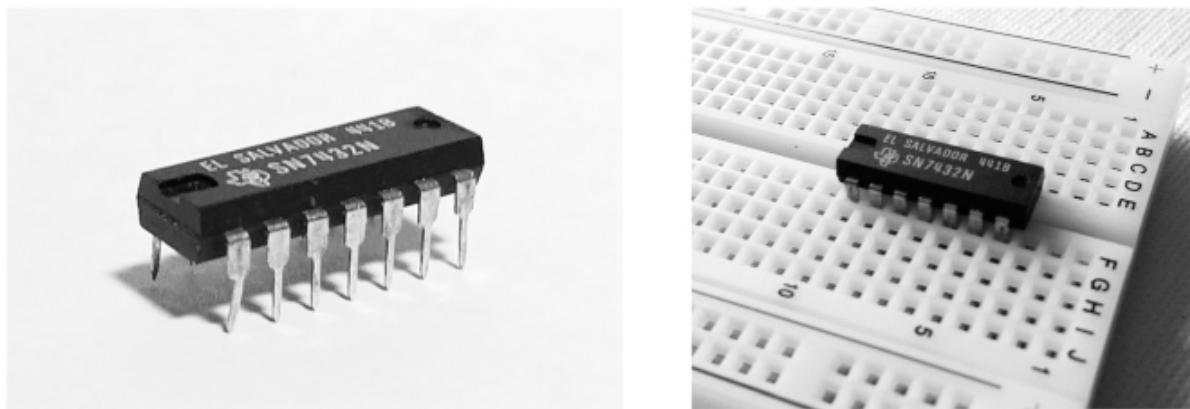


Figure 4-12: SN7432N integrated circuit in a dual in-line package (left), shown in a breadboard (right)

The 7432 IC comes in a package with 14 pins. Each of the four OR gates requires 3 pins, so that's 12 pins, plus 1 pin for positive voltage (V_{cc}) and 1 pin for ground, giving us 14 pins in total. Speaking of voltage, the 7400 series operates with an expected V_{cc} of 5V. That is, high voltage, a logical 1, is ideally 5V, and low voltage is 0V. However, in practice an input voltage of anywhere between 2V and 5V registers as high, and anywhere between 0V and 0.8V is considered low.

You can see in Figure 4-12 that the 7432 package has 7 pins on each side and neatly fits into a breadboard. When placing such a chip in a breadboard, be sure to place the chip so that it straddles the gap in the

center of the breadboard to ensure that pins directly across from each other (example: pins 1 and 14) aren't accidentally connected. Note the half-circle notch in the packaging; this tells you which way to orient the chip when identifying the pins.

You can see the arrangement of the circuit within the package in Figure 4-13. This is a *pinout* diagram—a diagram that labels the electrical contacts, or pins, of a component. The purpose of such a diagram is to show the component's external connection points, but usually a pinout diagram does not document the internal design of the circuit.

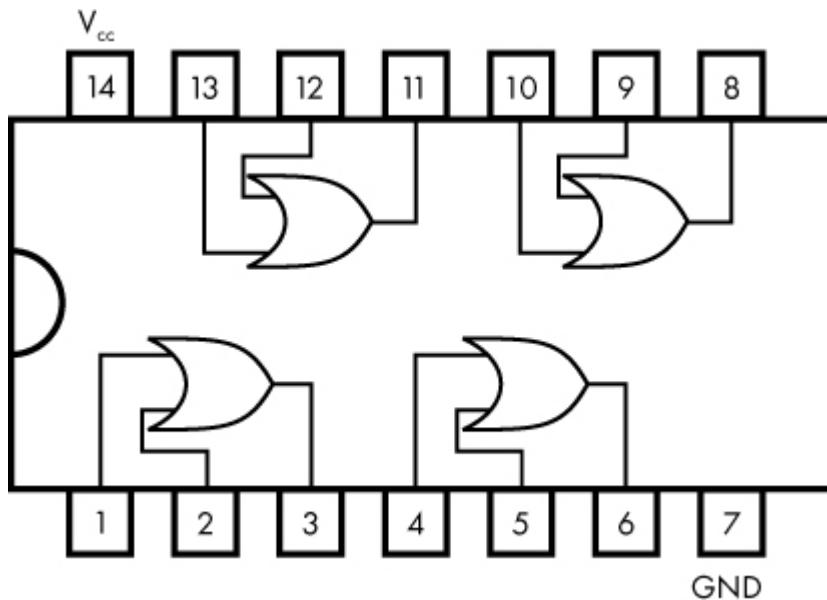


Figure 4-13: A pinout diagram showing the pin arrangement of a 7432 IC

Let's say you want to use one of the four OR gates in the 7432 chip, and you select the gate in the lower left of the pinout diagram in Figure 4-13, connected to pins 1, 2, and 3. To use this gate, you would connect the pins as shown in Table 4-3.

Table 4-3: Connecting a Single OR Gate in a 7432 IC

Pin	Connection
1	This is the A input of the OR gate. Connect to either 5V or GND for 1 or 0, respectively.

Pin	Connection
2	This is the B input of the OR gate. Connect to either 5V or GND for 1 or 0, respectively.
3	This is the output of the OR gate. Expect it to be either 5V or GND for 1 or 0, respectively.
7	Connect to ground.
14	Connect to a 5-volt power source.

The 7400 series contains hundreds of components. We don't cover them all here, but in Figure 4-14 you can see the pinouts of four common logic gates available in the series. You can do a quick online search to find pinouts for other 7400 ICs.

Equipped with the pinout diagrams for each of these integrated circuits, you now have the knowledge you need to physically construct the circuit you previously designed in Exercise 4-2.

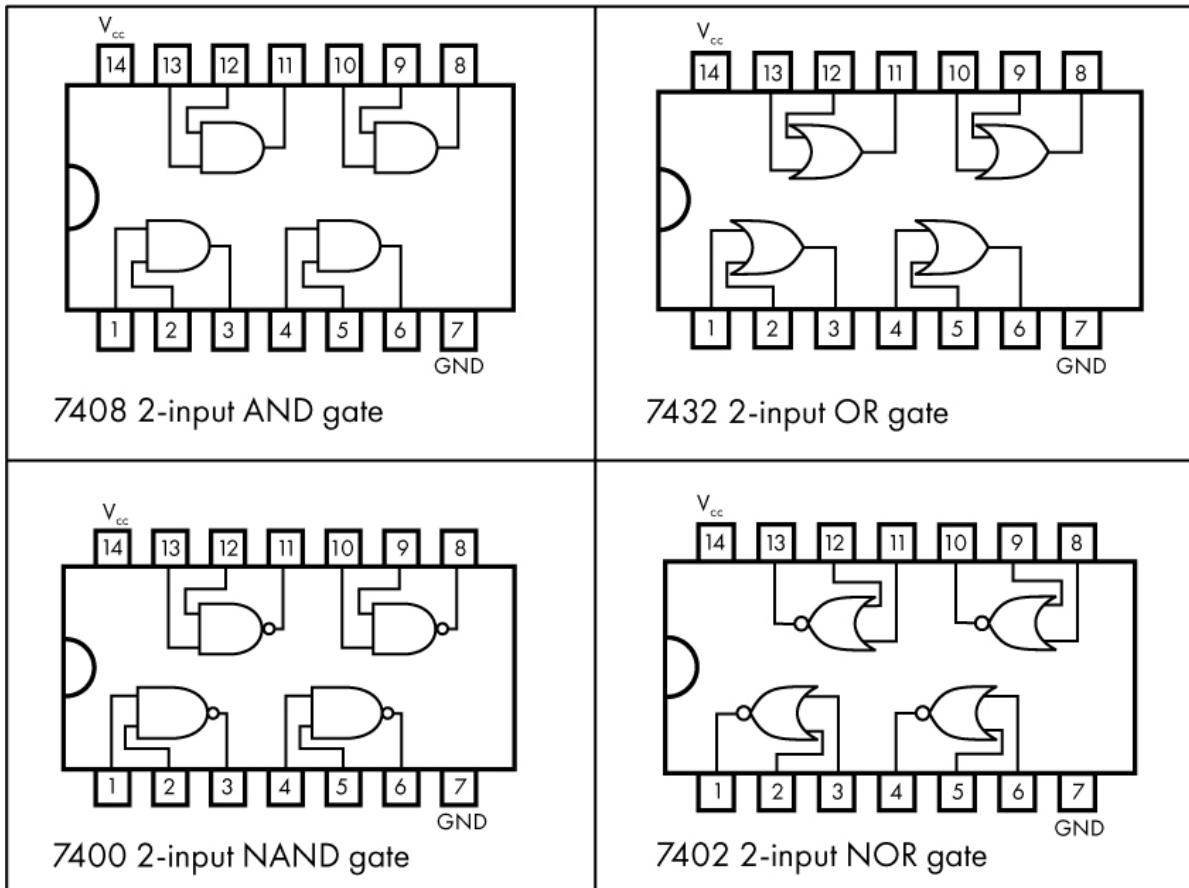


Figure 4-14: Pinout diagrams for common 7400 series integrated circuits

NOTE

Please see Project #4 page 68, where you can build a circuit that implements the logical expression $(A \text{ OR } B) \text{ AND } C$.

Summary

In this chapter, we covered binary digital circuits, circuits where we use voltage levels to represent a logical 1 or 0. You learned how switches can be used to construct logical operators, such as AND, in a physical circuit. We covered the limitations of using mechanical switches for that purpose, and we introduced a new circuit element that can act as an electrically controlled switch—the transistor. You learned about logic gates, circuit elements that implement a logical function. We covered integrated circuits, including the 7400 series.

In the next chapter we'll explore how logic gates can be used to build circuits that handle one of the fundamental abilities of a computer—math. You'll see how simple logic gates, when used together, allow for more complex functions. We'll also cover the representation of integers within a computer, using signed and unsigned numbers.

PROJECT #3: BUILD LOGICAL OPERATORS (AND, OR) WITH TRANSISTORS

In this project, you'll construct the physical circuits for a logical AND and a logical OR using transistors. To build these circuits, you'll need the following components:

- Breadboard (either a 400-point or 830-point model)
- Resistors (an assortment of resistors)
- 9-volt battery
- 9-volt battery clip connector
- 5mm or 3mm red LED
- Jumper wires (designed for use in a breadboard)
- Two transistors (Model 2N2222 in TO-92 packaging [also known as PN2222])

See “Buying Electronic Components” on page 333 for help getting these parts.

Before you begin hooking things up, you should know that some transistors and integrated circuits are *electrostatic-sensitive devices*, meaning they can be damaged by static discharge. Have you ever walked on carpet and then experienced a shock of static electricity when you touched something? That jolt of electricity can be fatal to electronic components. Even a static discharge too small for you to notice can damage an electronic component.

Professionals in the electronics industry avoid this problem by wearing a grounded wrist strap, by working in antistatic workspaces, and by wearing special clothing. Such precautions aren't followed by most hobbyists, but you should at least be aware of the risk of damaging your transistors or integrated circuits from a static discharge. Try to avoid static buildup and touch a grounded surface (like the screw that holds electric outlet covers in place) to discharge any static before you handle electrostatic-sensitive devices.

Now let's get back to the project at hand. The 2N2222 in a TO-92 package has three pins. If you hold the transistor with the flat surface facing toward you and the pins pointing down, then the left pin is the emitter, the middle pin is the base, and the right

pin is the collector (see Figure 4-5). You can also search online for “2N2222 TO-92” for more details about this particular transistor.

Follow the diagram in Figure 4-15 to build an AND circuit with a 9-volt battery, transistors, resistors, and an LED.

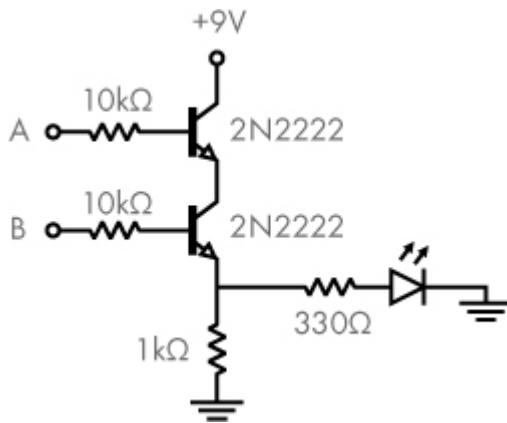


Figure 4-15: AND circuit diagram with suggested resistors, transistors, and an output LED

A and B should be connected to either 9V (for 1) or ground (for 0) to test inputs. The LED should turn on when the expected output is 1. See Table 4-1 for the expected output given for various combinations of inputs. Remember that +9V in the diagram represents a connection to the positive terminal of the battery, and the ground symbol in the diagram represents a connection to the negative terminal of the battery. Also, remember that an LED is designed to allow current flow in only one direction, so be sure to connect the shorter pin to ground. If your circuit isn't working as expected, check out “Troubleshooting Circuits” on page 340.

The constructed circuit should look something like Figure 4-16. Of course, your specific layout of parts on the breadboard may vary from mine.

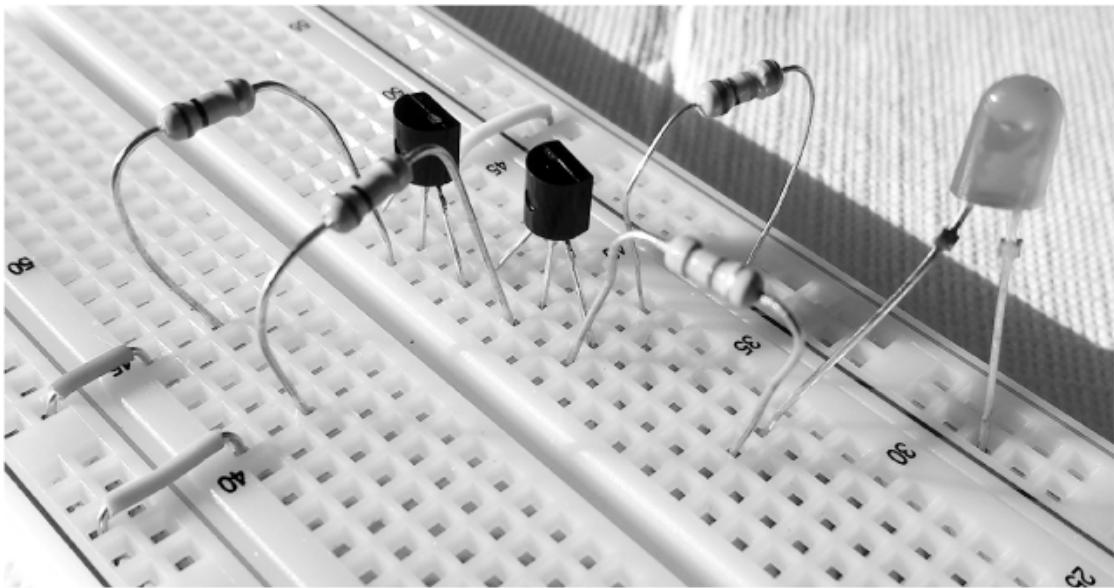


Figure 4-16: The AND circuit shown in Figure 4-15, built on a breadboard

Once you have a working AND circuit, let's move on to a similar OR circuit. Follow the diagram in Figure 4-17 to build an OR circuit with a 9-volt battery, transistors, resistors, and an LED.

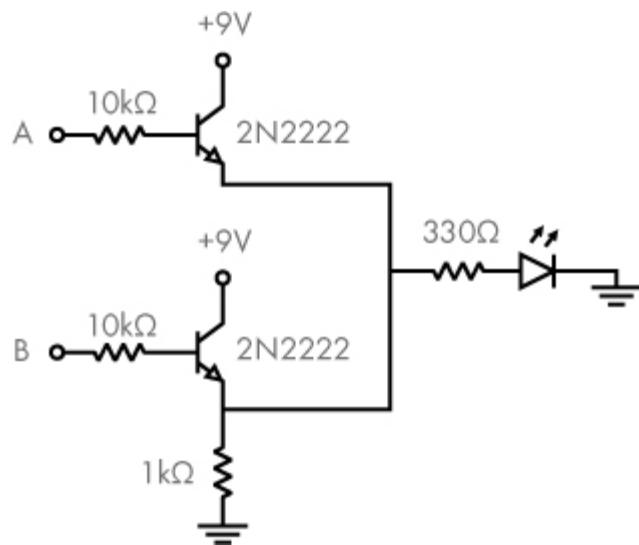


Figure 4-17: OR circuit with suggested resistors, transistors, and an output LED

As with the previous circuit, A and B should be connected to either 9V (for 1) or ground (for 0) to test inputs. The LED should turn on when the expected output is 1. See Table 4-2 for the expected output given for various combinations of inputs.

PROJECT #4: CONSTRUCT A CIRCUIT WITH LOGIC GATES

In Exercise 4-2, you drew a circuit diagram for (A OR B) AND C. If you skipped that exercise, I suggest you go back and do it before continuing here. The result should be something similar to the diagram shown in Figure 4-18.

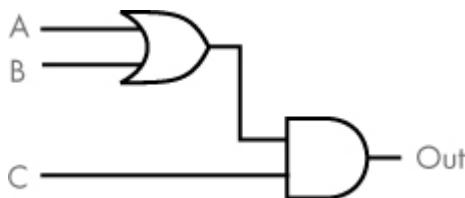


Figure 4-18: Logic gate diagram for (A OR B) AND C

Now let's physically construct that circuit! Connect the output pin to an LED (remember to include a resistor too) so that you can see if the output is 0 or 1. Your three inputs (A, B, C) can be directly connected to 5V or ground. Try connecting different combinations of the inputs to ensure your logic works as expected.

For this project, you need the following components:

- Breadboard
- LED
- A current-limiting resistor to use with your LED; approximately 220Ω
- Jumper wires
- 7408 integrated circuit (contains four AND gates)
- 7432 integrated circuit (contains four OR gates)
- 5-volt power supply
- Three pushbuttons or slide switches that will fit a breadboard (for bonus project)
- Three 470Ω resistors (for bonus question)

Since this circuit requires a 5-volt power supply, rather than a 9-volt battery, take a look at “Powering Digital Circuits” on page 336 for some options on how to set this up. Also, as before, see “Buying Electronic Components” on page 333 for help getting these parts.

You may have noticed that the list of components recommends a 220Ω resistor rather than the 330Ω resistor we used previously. This is because we've lowered the source voltage from 9V to 5V. The specific value you need for your circuit will depend on the forward voltage of the LED you are using, as we covered in Chapter 3. That said, this resistor value doesn't have to be precise. You can use a 220Ω , 200Ω , or 180Ω resistor—all of these values are readily available. The wiring diagram in Figure 4-19 shows the details of how to construct this circuit.

Keep in mind that the pins of the chip, once placed on the breadboard, are electrically connected to an entire row. Remember to place the integrated circuits so that they straddle the gap in the center of the breadboard to ensure that pins directly across from each other aren't accidentally connected. The completed circuit, if built on a breadboard, will look similar to the photo shown in Figure 4-20.

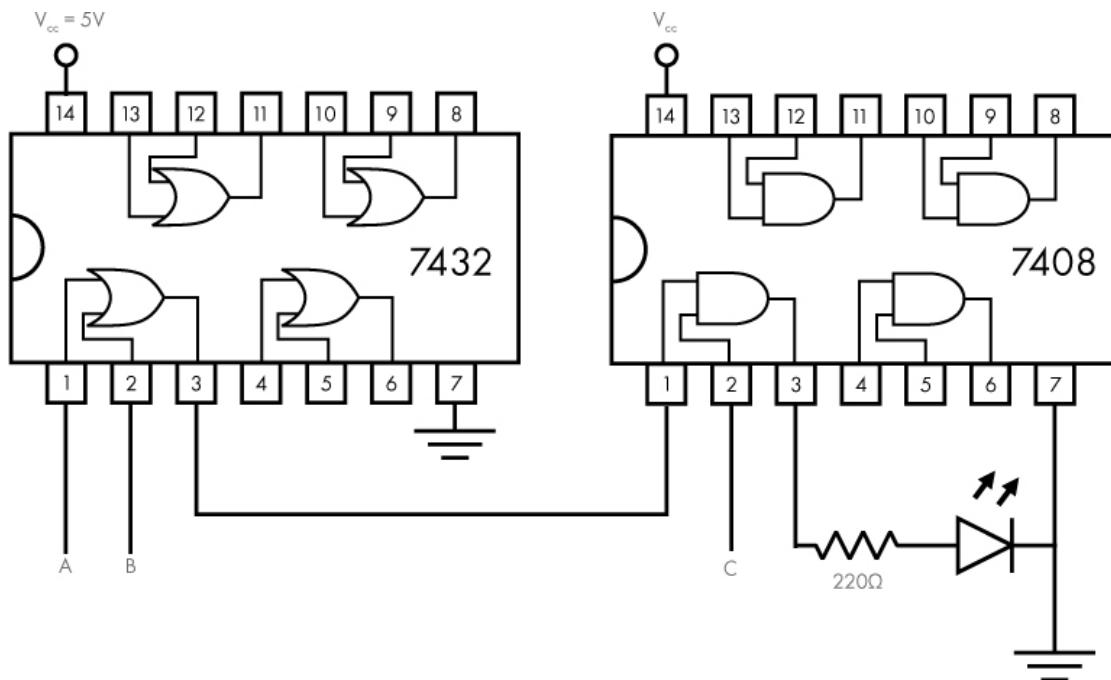


Figure 4-19: Wiring diagram for $(A \text{ OR } B) \text{ AND } C$

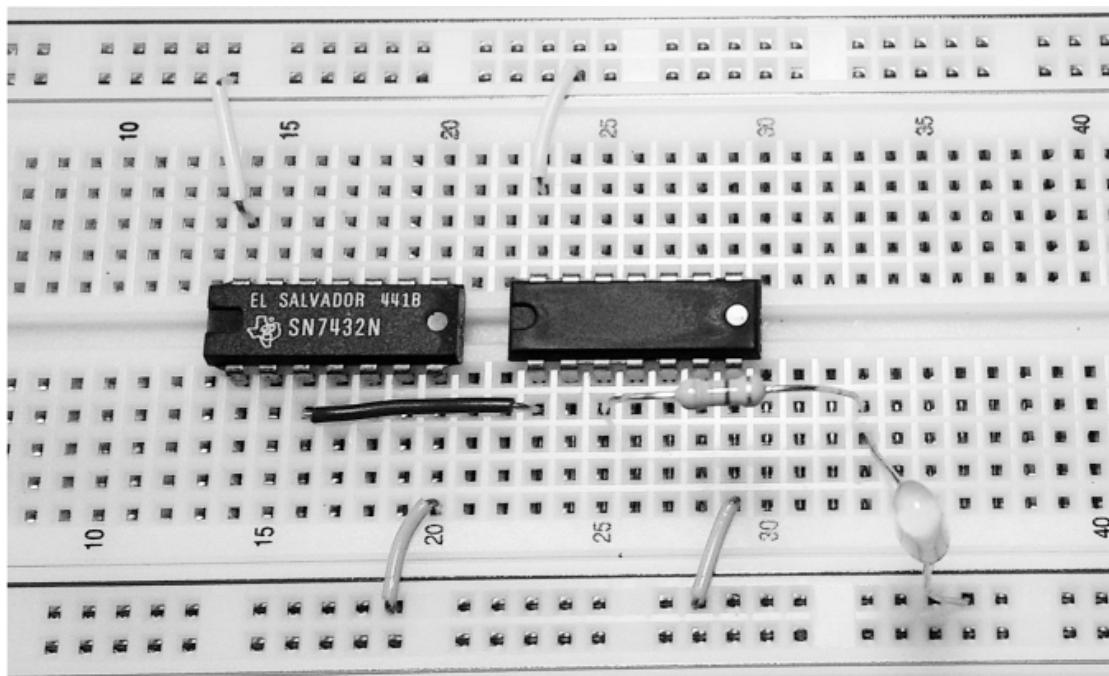


Figure 4-20: Breadboard implementation of (A OR B) AND C with inputs A, B, and C left disconnected

Note that in Figure 4-20 the 7432 IC is on the left and the 7408 is on the right. In this particular layout the power column along the top is connected to 5V and the negative power column at the bottom is connected to ground, but neither connection is shown in the photo. Also note that inputs A, B, and C are disconnected here; they will need to be attached to ground or 5V to test various inputs.

Once you've constructed this circuit, you can try connecting different combinations of the inputs to ensure your logic works as expected. Connect an input to 5V to represent a logical 1, or to ground to represent a logical 0. Check the circuit's behavior against the truth table for (A OR B) AND C shown in Table 4-4. If your circuit isn't working as expected, check out "Troubleshooting Circuits" section on page 340.

Table 4-4: (A OR B) AND C—Truth Table

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Manually moving the input wires between ground and 5V isn't ideal. A better design would be to connect inputs A, B, and C to mechanical switches so you can easily change the inputs without rewiring the circuit. As a bonus project, let's add some mechanical switches to control our inputs. Your first instinct may be to hook up a switch between an input and V_{CC} , as shown in Figure 4-21, where closing the switch connects the input to 5V, a logical 1.

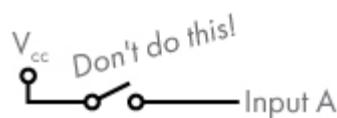


Figure 4-21: A switch between V_{CC} and the input. Hint: don't do this.

Unfortunately, there's a problem with the approach shown in Figure 4-21. The closed switch works as expected, but the open switch does not. You might expect an open switch to result in 0V at input A, but that isn't necessarily the case. When the switch is open, the

voltage at input A “floats” and is an unpredictable value. Remember that input A in Figure 4-21 represents the input pin of a 7432 OR gate. This input is designed to be connected to a high or low voltage; leaving it disconnected puts the logic gate in a state that is undefined. We need to wire our switch so that a predictable low voltage is present when the switch is closed. As shown in Figure 4-22, we can do this with a *pull-down resistor*—a regular resistor used for the purpose of “pulling” an input low when that input isn’t connected high.

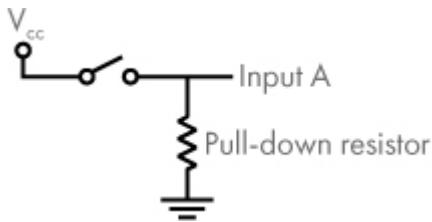


Figure 4-22: Use a pull-down resistor to ensure digital inputs are correct.

Let’s consider what will happen when we add a pull-down resistor as shown in Figure 4-22. To know how the input of a 7432 integrated circuit will respond under various conditions, we can look up the voltage and current characteristics as described in the manufacturer’s data sheet. I won’t go into the details here (feel free to look online for the data sheet for your specific 7432 chip), but in summary, when the switch is open, a small current flows from input A, through the resistor, to ground.

If we use a resistor with a low value, the current flowing from input A results in a voltage at the input that is low enough to register as a logical 0. When the switch is closed, the input is directly connected to V_{cc} and the input will be a logical 1. For 74LS series components (discussed in Appendix B), a pull-down resistor value of 470Ω or $1k\Omega$ usually works for logic gate inputs. I recommend those specific values since they are commonly available and meet our requirements. Values higher than $1k\Omega$ do not reliably work as pull-down resistors for 74LS components. When you use pull-down resistors, you can build the full circuit with switches as shown in Figure 4-23.

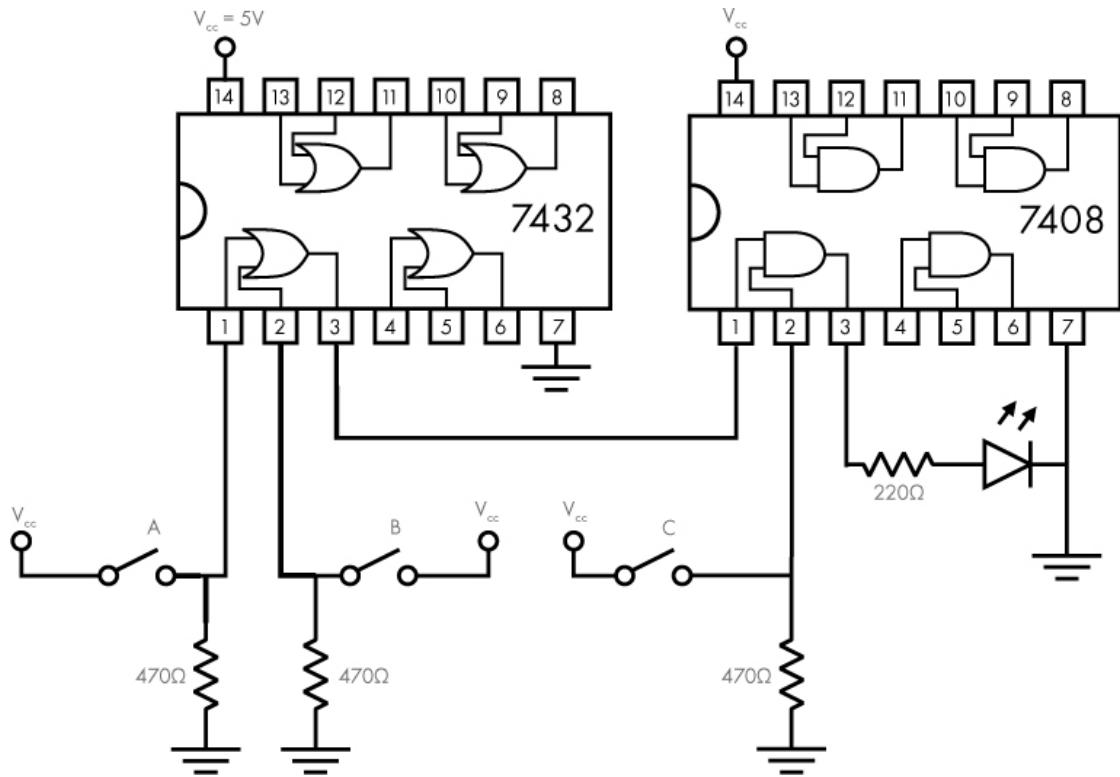


Figure 4-23: Wiring diagram for (A OR B) AND C with switches added to control the inputs

The completed circuit, if built on a breadboard, will look similar to the photo shown in Figure 4-24. In my circuit, I used pushbuttons for switches, as seen in the lower-left corner. If you happen to look very closely, you might see that the pull-down resistors in this photo are $1\text{k}\Omega$, which differs from the diagram's suggested value of 470Ω , but either will work.

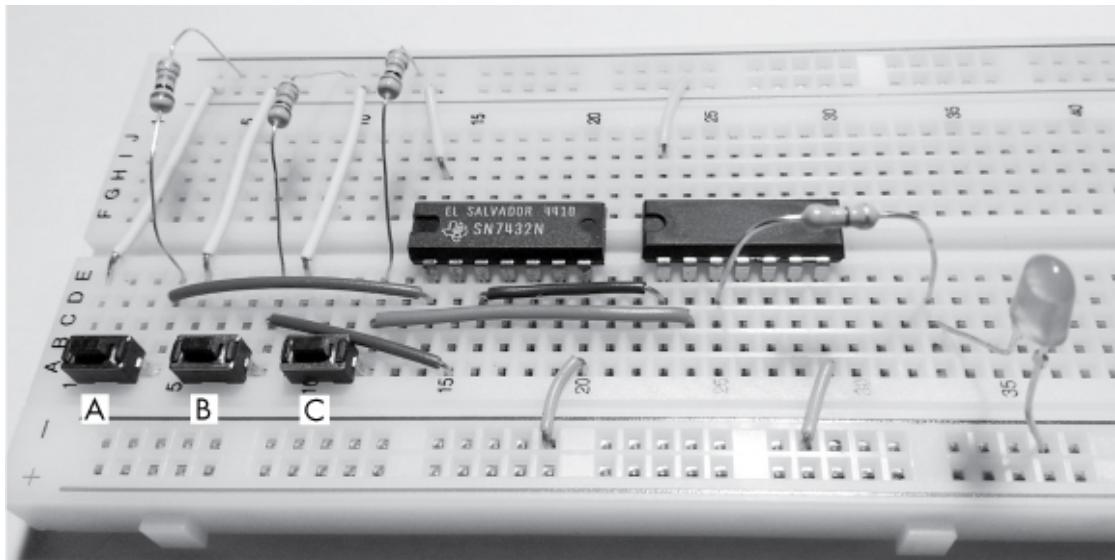


Figure 4-24: Breadboard implementation of (A OR B) AND C

Once you've built your circuit as shown in Figure 4-24, be sure to check various combinations of inputs and see if it matches the truth table for (A OR B) AND C in Table 4-4. If your circuit isn't working as expected, check out "Troubleshooting Circuits" on page 340.

5

MATH WITH DIGITAL CIRCUITS



In the previous chapter we covered logic gates and digital circuits, which enable us to implement logical expressions in hardware. Earlier in this book we defined computers as electronic devices that can be programmed to carry out a set of instructions. In this chapter, I begin to bridge those concepts by showing you how simple logic gates pave the way for the operations a computer executes. We cover a specific operation that all computers are able to perform—addition. First, we go over the basics of addition in binary. Then we use logic gates to build hardware that adds, demonstrating how simple gates work together in a computer to perform useful operations. Finally, we cover the representation of integers as signed and unsigned numbers in a computer.

Binary Addition

Let's look at the basics of how addition works in binary. The underlying principles of addition are the same for all place-value systems, so you have a head start since you already know how to add in decimal! Rather

than deal in abstract concepts, let's take a concrete example: adding two specific 4-bit numbers, 0010 and 0011, as shown in Figure 5-1.

$$\begin{array}{r} 0010 \\ + 0011 \\ \hline \text{????} \end{array}$$

Figure 5-1: Adding two binary numbers

Just as we do in decimal, we begin with the rightmost place, known as the *least significant bit*, and add the two values together (Figure 5-2). Here, $0 + 1$ is 1.

$$\begin{array}{r} 001\lceil 0 \\ + 001\lceil 1 \\ \hline \text{???}\lceil 1 \end{array}$$

Figure 5-2: Adding the least significant bit of two binary numbers

Now let's move one bit to the left and add those values together, as shown in Figure 5-3.

$$\begin{array}{r} 1 \quad \leftarrow \text{Carry} \\ 001\lceil 0 \\ + 001\lceil 1 \\ \hline \text{??}\lceil 01 \end{array}$$

Figure 5-3: Adding the twos place

As you can see in Figure 5-3, this place requires us to add $1 + 1$, which presents us with an interesting twist. In decimal, we represent $1 + 1$ with the symbol 2, but in binary we only have two symbols, 0 and 1. In binary, $1 + 1$ is 10 (see Chapter 1 for an explanation), which requires two bits to represent. We can only put one bit in this place, so 0 goes into the current place, and we carry the 1 to the next place, as indicated in Figure 5-3. We can now move to the next place (see Figure 5-4), and when we add these bits, we must include the carried bit from the previous place. This gives us $1 + 0 + 0 = 1$.

$$\begin{array}{r}
 1 \quad \leftarrow \text{Carry} \\
 0010 \\
 + 0011 \\
 \hline
 ?101
 \end{array}$$

Figure 5-4: Adding the fours place

Finally, we add the *most significant bit*, as shown in Figure 5-5.

$$\begin{array}{r}
 0010 \\
 + 0011 \\
 \hline
 0101
 \end{array}$$

Figure 5-5: Adding the eights place

Once we add all the places, the complete result in binary is 0101. One way to sanity-check our work is to simply convert everything to decimal, as shown in Figure 5-6.

Binary	Decimal
0010	2
+ 0011	+ 3
<hr/>	<hr/>
0101	5

Figure 5-6: Adding two binary numbers, then converting to decimal

As you can see in Figure 5-6, our answer in binary (0101) matches what we'd expect in decimal (5). Simple enough!

EXERCISE 5-1: PRACTICE BINARY ADDITION

You can now practice what you've just learned. Try the following addition problems:

$$0001 + 0010 = \underline{\hspace{2cm}}$$

$$0011 + 0001 = \underline{\hspace{2cm}}$$

$$0101 + 0011 = \underline{\hspace{2cm}}$$

$$0111 + 0011 = \underline{\hspace{2cm}}$$

See Appendix A for answers.

Fortunately, addition works the same way no matter what base you're working in. The only difference between bases is how many symbols are available to you. Binary makes addition particularly straightforward, since the addition of each place always results in exactly two output bits, each with only two possible values:

Output 1 A *sum* bit (S) of 0 or 1, representing the least significant bit of the result of the addition operation

Output 2 A *carry-out* bit (C_{out}) of 0 or 1

Half Adders

Now let's say we want to construct a digital circuit that adds a *single* place of two binary numbers. We initially focus on the least significant bit. Adding the least significant bits of two numbers only requires two binary inputs (let's call them A and B), and the binary outputs are a sum bit (S) and a carry-out bit (C_{out}). We call such a circuit a *half adder*.

Figure 5-7 shows the symbol for a half adder.

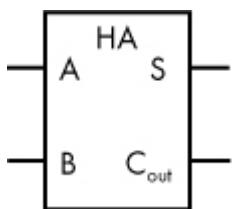


Figure 5-7: Symbol for a half adder

To clarify how the half adder fits in with our earlier example of adding two binary numbers, Figure 5-8 relates the two concepts.

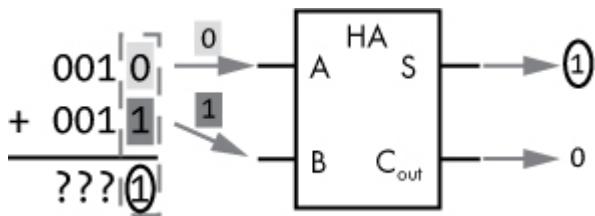


Figure 5-8: Half adder in action

As you can see in Figure 5-8, the least significant bit from the first number is input A and the least significant bit from the second number is input B. The sum is an output, S, and the carry-out is also an output.

Internally, the half adder can be implemented as a combinational logic circuit, so we can also describe it with a truth table, as shown in Table 5-1. Note that A and B are inputs, while S and C_{out} are outputs.

Table 5-1: Truth Table for a Half Adder

Inputs		Outputs	
A	B	S	C _{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Let's walk through the truth table in Table 5-1. Adding 0 and 0 results in 0 with no carry. Adding 0 and 1 (or the reverse) results in 1 with no carry. Adding 1 and 1 gives us 0 with a carry of 1.

Now, how do we go about implementing this with digital logic gates? The solution is straightforward if we examine one output at a time, as shown in Figure 5-9.

Inputs		Outputs	
A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

XOR AND

Figure 5-9: Truth table for a half adder; outputs match XOR, AND

Looking at only output S in Figure 5-9, we can see that it exactly matches the truth table for an XOR gate (see Chapter 4). Looking at C_{out} only, we can observe that it matches the output of an AND gate. Therefore, we can implement a half adder using only two gates: XOR and AND, as shown in Figure 5-10.

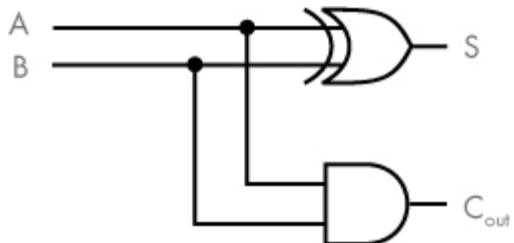


Figure 5-10: Half adder implemented with two logic gates, XOR and AND

As you can see in Figure 5-10, digital inputs A and B act as the inputs for both the XOR gate and the AND gate. The gates then produce the desired outputs, S and C_{out} .

NOTE

Please see Project #5 on page 89, where you can build a circuit for a half adder.

Full Adders

A half adder can handle the logic needed to perform addition for the least significant bits of two binary numbers. However, each subsequent bit requires an additional input: the carry-in bit, C_{in} . This is because every bit place, except the least significant bit, needs to handle the situation where addition of the previous bit place resulted in a carry-out, which in turn becomes the carry-in for the current bit place. Adding a C_{in} input to our adder component requires a new circuit design, and we call this circuit a *full adder*. The symbol for a full adder, shown in Figure 5-11, is similar to the symbol for a half adder, differing only by an extra input, C_{in} .

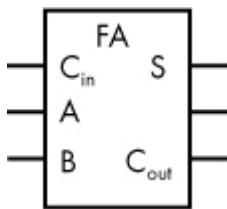


Figure 5-11: Symbol for a full adder

In Figure 5-12, we see an example of the relationship between binary addition of a single place and the full adder.

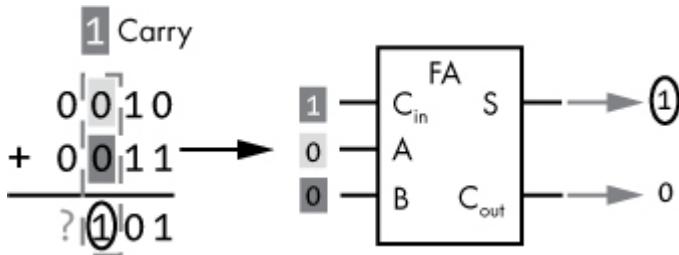


Figure 5-12: Full adder in action

The full adder handles the addition of a single place, including the carry-in bit. In the example shown in Figure 5-12, we add the bits in the 4s place. Since the bits in the previous place were 1 and 1, there's a carry-in of 1. The full adder takes all three inputs ($A = 0$, $B = 0$, and $C_{in} = 1$) and produces an output of $S = 1$ and $C_{out} = 0$.

For a complete picture of the possible inputs and outputs of a full adder, we can use a truth table, shown in Table 5-2. This table has three

inputs (A , B , C_{in}) and two outputs (S , C_{out}). Take a moment to consider the outputs from the various input combinations.

Table 5-2: Full Adder Truth Table

Inputs		Outputs		
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

How do we go about implementing a full adder? As the name implies, a full adder can be realized by combining two half adders (Figure 5-13).

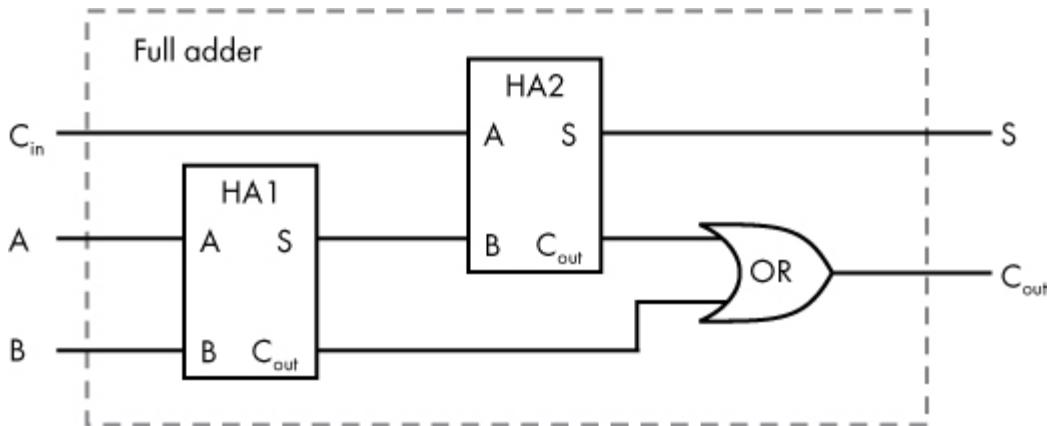


Figure 5-13: A full adder circuit realized with two half adders and an OR gate

A full adder's sum output (S) should be the sum of A and B (which we can calculate using one half adder—HA1) plus C_{in} (which we can calculate with a second half adder—HA2), as shown in Figure 5-13.

We also need our full adder to output a carry-out bit. This turns out to be simple to implement, because the full adder's C_{out} value is 1 if the carry-out from either half adder is 1. We can therefore use an OR gate to complete the full adder circuit, as shown in Figure 5-13.

Here we see another example of encapsulation. Once this circuit is constructed, the functionality of a full adder can be used without knowledge of the specific implementation details. In the next section, let's see how we can use full and half adders together to add a number with multiple bits.

A 4-bit Adder

A full adder allows us to add two 1-bit numbers, plus a carry-in bit. This gives us a building block for creating a circuit that can add binary numbers with more than one place. Let's now combine several 1-bit adder circuits to create a 4-bit adder. Let's use a half adder for the least significant bit (since it doesn't require a carry-in) and full adders for the other bits. The idea is to string the adders together so the carry-out from each adder flows into the carry-in of the subsequent adder, as shown in Figure 5-14.

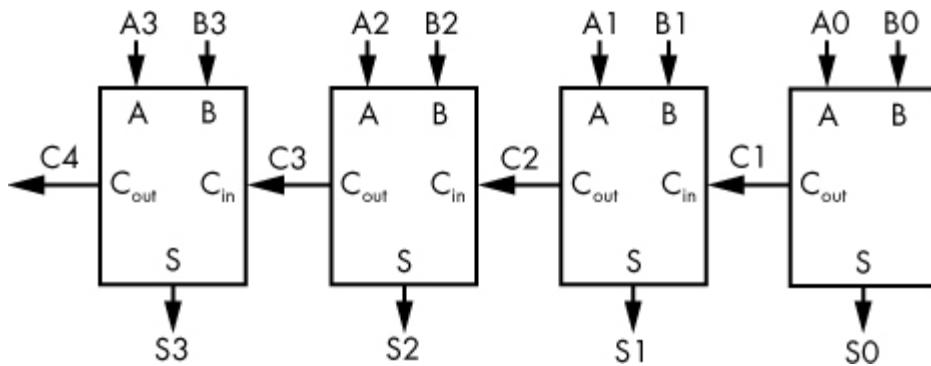


Figure 5-14: A 4-bit adder

For consistency with the way people write numbers, I've arranged Figure 5-14 with the least significant bit on the right, and the diagram's flow progressing from right to left. This means that our adder block

diagrams will have inputs and outputs positioned differently than previously shown; don't let that confuse you!

In Figure 5-15, I applied our earlier example of two (0010) plus three (0011) to this 4-bit adder.

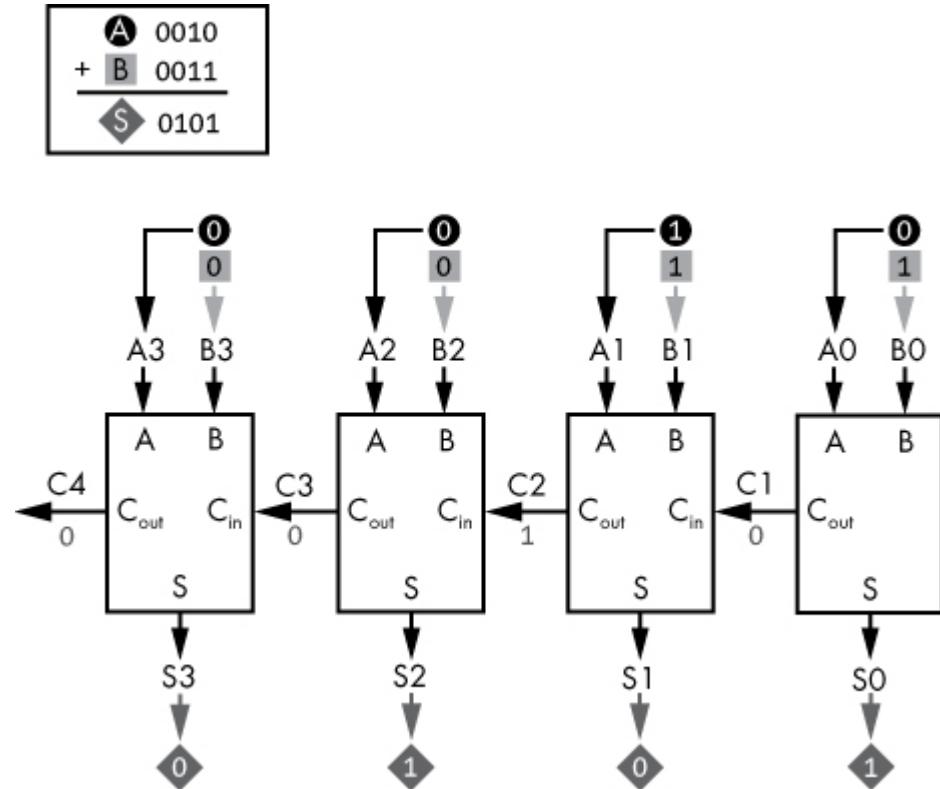


Figure 5-15: A 4-bit adder in action

In Figure 5-15, we can see how each bit from input A (0010) and input B (0011) is fed into each subsequent adder unit, starting with the least significant bit on the right, moving to the most significant bit on the left. You can follow the flow of the diagram by reading from right to left. Add the rightmost bits first, 0 (A0) and 1 (B1); the result is 1 (S0) with a carry of 0.

The output carry bit from the rightmost adder flows into the next adder as C1, where 1 (A1) and 1 (B1) are added, along with the carry of 0. This results in 0 (S1) with a carry of 1 (C2). The process continues until the leftmost adder completes. The final result is a set of the output bits, 0101 (S3 to S0), and a carry of 0 (C4). If we need to handle a larger

number of bits, we can extend the design in Figure 5-15 by simply incorporating more full adders.

This type of adder requires the carry bits to propagate, or ripple, through the circuit. For this reason, we call this circuit a *ripple carry adder*. Each carry bit that ripples to the next full adder introduces a small delay, so extending the design to handle more bits makes the circuit slower. The output of the circuit will be inaccurate until all the carry bits have time to propagate.

Several versions of 4-bit adders are available in the 7400 series of ICs. If you need a 4-bit adder in a project, you can use such an IC rather than construct the adder from individual logic gates.

Let's pause here and consider the broader implications of what we've just covered. Yes, you learned how to build a 4-bit adder, but how does this relate to computing? Recall that computers are electronic devices that can be programmed to carry out a set of logical instructions. Those instructions include mathematical operations, and we just saw how logic gates, built from transistors, can be combined to perform one of those operations—addition. We covered addition as a concrete example of a computer operation, and although we don't go into the details in this book, you can also implement other fundamental computer operations with logic gates. This is how computers work—simple logic gates work together to perform complex tasks.

Signed Numbers

Thus far in this chapter we've only concerned ourselves with positive integers, but what if we want to be able to handle negative integers as well? First, we need to consider how a negative number can be represented in a digital system like a computer. As you know, all data in a computer is represented as sequences of 0s and 1s. A negative sign is neither a 0 nor a 1, so we need to adopt a convention for representing negative values in a digital system. In computing, a *signed number* is a

sequence of bits that can be used to represent a negative or positive number, depending on the specific values of those bits.

A digital system's design must define how many bits are used to represent an integer. Typically, we represent integers using 8, 16, 32, or 64 bits. One of those bits can be assigned to represent a negative sign. We can, for example, say that if the most significant bit is 0, then the number is positive, and if the most significant bit is 1, then the number is negative. The remaining bits are then used to represent the absolute value of the number. This approach is known as *signed magnitude representation*. This works, but it requires extra complexity in a system's design to account for the bit that has a special meaning. For example, the adder circuits we built earlier would need to be modified to account for the sign bit.

A better way to represent negative numbers in a computer is known as *two's complement*. In this context, the two's complement of a number represents the negative of that number. The simplest way to find the two's complement of a number is to replace every 1 with a 0 and every 0 with a 1 (in other words, flip the bits), and then add 1. Hang tight with me here; this is going to seem overly complicated at first, but if you follow the details, it will make sense.

Let's take a 4-bit example, the number 5, or 0101 in binary. Figure 5-16 shows the process of finding the two's complement of this number.

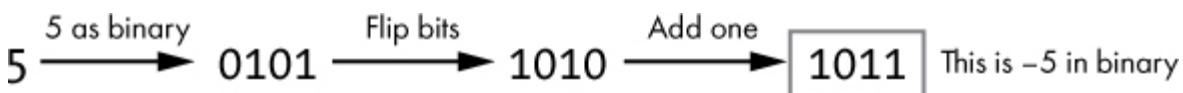


Figure 5-16: Finding the two's complement of 0101

First, we flip the bits, then we add one, giving us 1011 binary. So, in this system, 5 is represented as 0101 and -5 is represented as 1011. Keep in mind that 1011 only represents -5 in the context of a 4-bit signed number. That binary sequence could be interpreted in other ways in a different context, as we will see later. What if we want to go the other way, starting with the negative value? The process is the same, as shown in Figure 5-17.

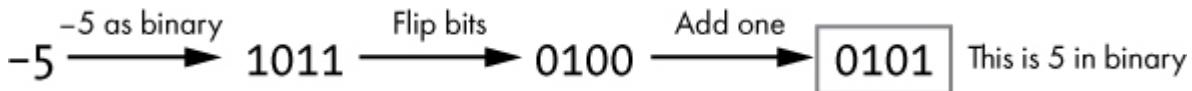


Figure 5-17: Finding the two's complement of 1011

As you can see in Figure 5-17, taking the two's complement of -5 gets us back to the original value of 5. This makes sense, given that the negative of -5 is 5.

EXERCISE 5-2: FIND THE TWO'S COMPLEMENT

Find the 4-bit two's complement of 6. See Appendix A for the answer.

Now we know how to represent a number as a positive value or a negative value using two's complement, but how is this useful? I think the easiest way to see the benefits of this system is to just try it. Let's say we want to add 7 and -3 (that is, subtract 3 from 7). We expect the result to be positive 4. Let's first determine what our inputs are in binary, shown in Figure 5-18.

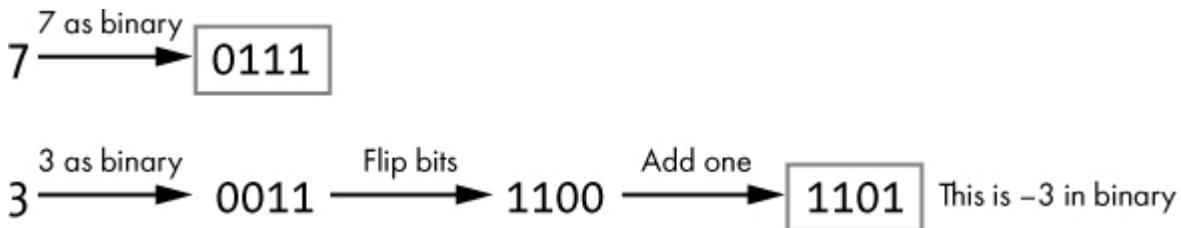


Figure 5-18: Find the 4-bit two's complement form of 7 and -3 .

Our two binary inputs will be 0111 and 1101. Now, forget for a moment that we're dealing with positive and negative values. Just add the two binary numbers. Don't worry about what the bits represent, just add them, and prepare to be amazed! Look at Figure 5-19 once you've done the binary math.

Binary	Signed decimal
0111	7
+ 1101	- 3
<hr style="border: 0.5px solid black; margin: 5px 0;"/>	<hr style="border: 0.5px solid black; margin: 5px 0;"/>
1 0100	4

Figure 5-19: Addition of two binary numbers, interpreted as signed decimal

As you can see in Figure 5-19, this addition results in a carry-out bit beyond what a 4-bit number can represent. I'll explain this in more detail later, but for now, we can ignore that carry-out bit. This gives us a 4-bit result of 0100, which is positive 4, our expected number! That's the beauty of two's complement notation. We don't have to do anything special during the addition or subtraction operation; it just works.

Let's pause here and reflect on the implications of this. Remember those adder circuits we built earlier? They will work for negative values too! Any circuit designed to handle binary addition can use the two's complement as a means of handling negative numbers or subtraction. The detailed mathematical explanation for why all of this works is outside the scope of this book; if you are curious, there are good explanations available online.

TWO'S COMPLEMENT TERMINOLOGY

The term *two's complement* actually refers to two related concepts. Two's complement is a form of *notation* for representing positive and negative integers. For example, the number 5, represented in 4-bit two's complement notation is 0101, whereas -5 is represented as 1011. At the same time, two's complement is also an *operation* used to negate an integer stored in two's complement format. For example, taking the two's complement of 0101 gives us 1011.

Here's another way to look at two's complement notation: the most significant place has a weight equal to the negative value of that place, and all other places have weights equal to the positive values of those

places. So, for a 4-bit number, the places have the weights shown in Figure 5-20.

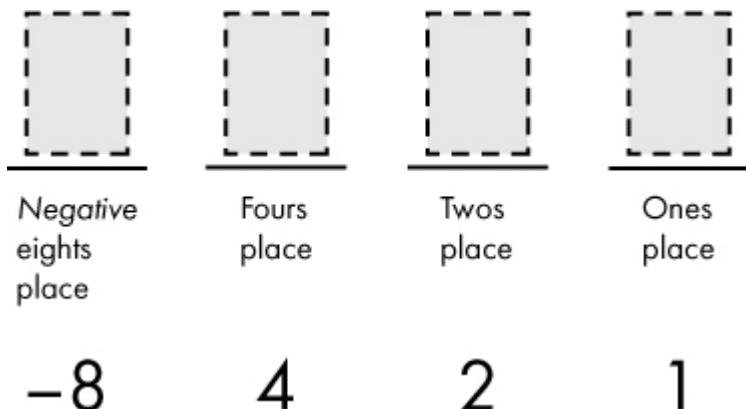


Figure 5-20: Place-value weights of a signed 4-bit number using two's complement notation

If we then apply this approach to the two's complement representation of -3 (1101), we can calculate the decimal value as shown in Figure 5-21.

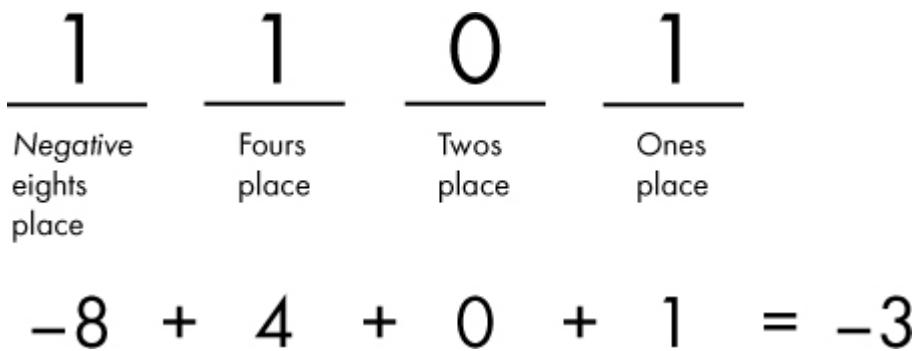


Figure 5-21: Find the signed decimal value of 1101 using two's complement place value.

When dealing with two's complement, I find that looking at the most significant place's weight as equal to the negative value of that place is a convenient mental shortcut. Now that we have covered the weights of all the places in a 4-bit signed number, we can examine the full range of values that can be represented with such a number, as shown in Table 5-3.

Table 5-3: All Possible Values of a 4-bit Signed Number

Binary	Signed decimal
--------	----------------

Binary	Signed decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Given Table 5-3, we can observe that for a signed 4-bit number, our maximum value is 7 and our most negative value is -8, for a total of 16 possible values. Note that anytime the most significant bit is 1, the value will be negative. We can generalize as follows for an n -bit signed number:

- Maximum value: $(2^{n-1}) - 1$
- Minimum value: $-(2^{n-1})$
- Count of unique values: 2^n

So, for an 8-bit signed number (as an example), we find that

- Maximum value = 127
- Minimum value = -128
- Count of unique values = 256

Unsigned Numbers

Signed integers that use two's complement to represent negative values are a convenient way to handle negatives without requiring specialized adder hardware. The adder we covered earlier works just as well with negative values as it does with positive values. However, there are scenarios in computing in which negative values simply aren't needed, and treating our numbers as signed simply wastes about half the range of values (all the negative values go unused), while also capping the maximum possible value to about half of what could otherwise be represented. Because of this, in such scenarios we want to treat numbers as *unsigned*, meaning the sequence of bits always represents a positive value or zero, but never a negative value.

Looking again at a 4-bit number, Table 5-4 shows what each 4-bit binary value represents if we interpret it as signed or unsigned.

Table 5-4: All Possible Values of a 4-bit Number, Signed or Unsigned

Binary	Signed decimal	Unsigned decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

We can generalize as follows for an n -bit unsigned number:

- Maximum value: $(2^n) - 1$
- Minimum value: 0
- Count of unique values: 2^n

So, let's take an example 4-bit value, say 1011. Looking at Table 5-4, what does it represent? Does it represent -5 or does it represent 11? The answer is “it depends!” It can represent either -5 or 11, depending on the context. From an adder circuit's point of view, it doesn't matter. As far as the adder is concerned, the 4-bit value is just 1011. Any addition operation is performed the same way regardless; the only difference is how we interpret the result. Let's look at an example. In Figure 5-22, we add two binary numbers: 1011 and 0010.

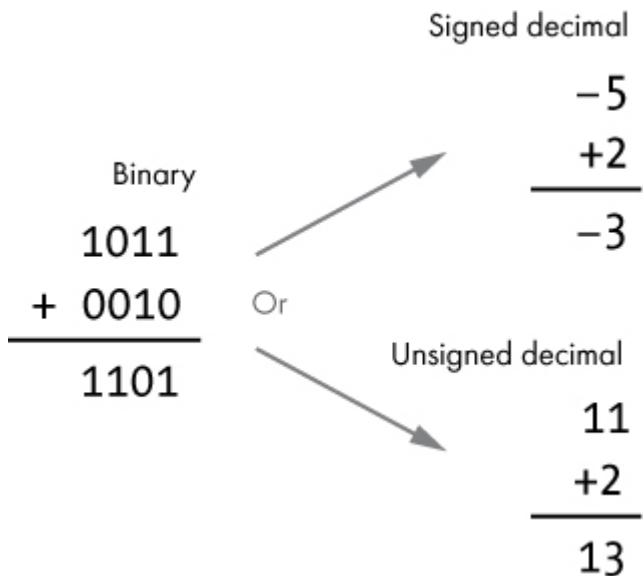


Figure 5-22: Adding two binary numbers, interpreted as signed or unsigned

As you can see in Figure 5-22, adding those two binary numbers results in 1101, regardless of whether we're working with signed or unsigned numbers. After the calculation is complete, we get to decide how to interpret that result. Either we just added -5 and 2 and the result was -3 , or we added 11 and 2 and the result was 13. In either case the math works out; it's just a matter of interpretation! In the context of

computing, it is the program running on a computer that is responsible for correctly interpreting the result of an addition operation as signed or unsigned.

EXERCISE 5-3: ADD TWO BINARY NUMBERS AND INTERPRET AS SIGNED AND UNSIGNED

Add 1000 and 0110. Interpret your work as signed numbers. Then interpret it as unsigned. Do the results make sense? See Appendix A for the answer.

So far, we've mostly ignored the most significant carry-out bit, but it has a meaning that should be understood. For unsigned numbers, a carry-out of 1 means that an *integer overflow* has occurred. In other words, the result is too large to be represented by the number of bits assigned to represent an integer. For signed numbers, if the most significant carry-in bit is not equal to the most significant carry-out bit, then an overflow has occurred. Also for signed numbers, if the most significant carry-in is equal to the most significant carry-out, then no overflow has occurred, and the carry-out bit can be ignored.

Integer overflows are a source of errors in computer programs. If a program does not check if an overflow has occurred, then the result of an addition operation may be incorrectly interpreted, leading to unexpected behavior. A famous example of an integer overflow error is found in the arcade game Pac-Man. When the player reaches level 256, the right side of the screen is filled with garbled graphics. This happens because the level number is stored as an 8-bit unsigned integer, and adding 1 to its maximum value of 255 results in an overflow. The game's logic doesn't account for this condition, leading to the glitch.

Summary

In this chapter, we used addition as an example of how computers build upon logic gates to perform complex tasks. You learned how to perform

addition in binary and how to construct hardware that can add binary numbers from logic gates. You saw how a half adder can add 2 bits and produce a sum and a carry-out bit, whereas a full adder can add 2 bits plus a carry-in bit. We covered how single-bit adders can be combined to perform multi-bit addition. You also learned how integers are represented in a computer using signed and unsigned numbers.

In the next chapter, we'll move beyond combinational logic circuits and learn about sequential logic. With sequential logic, hardware can have memory, allowing for the storage and retrieval of data. You'll see how memory circuits can be built. We'll also cover clock signals, a method of synchronizing the state of multiple components in a computer system.

PROJECT #5: BUILD A HALF ADDER

In this project, you'll construct a half adder using an XOR gate and an AND gate. The inputs will be controlled with switches or pushbuttons. The outputs should be connected to LEDs to easily observe their states. For this project, you'll need the following components:

- Breadboard
- Two LEDs
- Two current-limiting resistors to use with your LEDs (approximately 220Ω)
- Jumper wires
- 7408 IC (contains four AND gates)
- 7486 IC (contains four XOR gates)
- Two pushbuttons or switches that will fit a breadboard
- Two 470Ω resistors
- 5-volt power supply

As a reminder, see the sections “Buying Electronic Components” on page 333 and “Powering Digital Circuits” on page 336 if you need help on those topics. For a reminder of how the pins are numbered on the 7408 IC, see Figure 4-14. The 7486 IC wasn't covered previously, so I'm including its pinout diagram here in Figure 5-23.

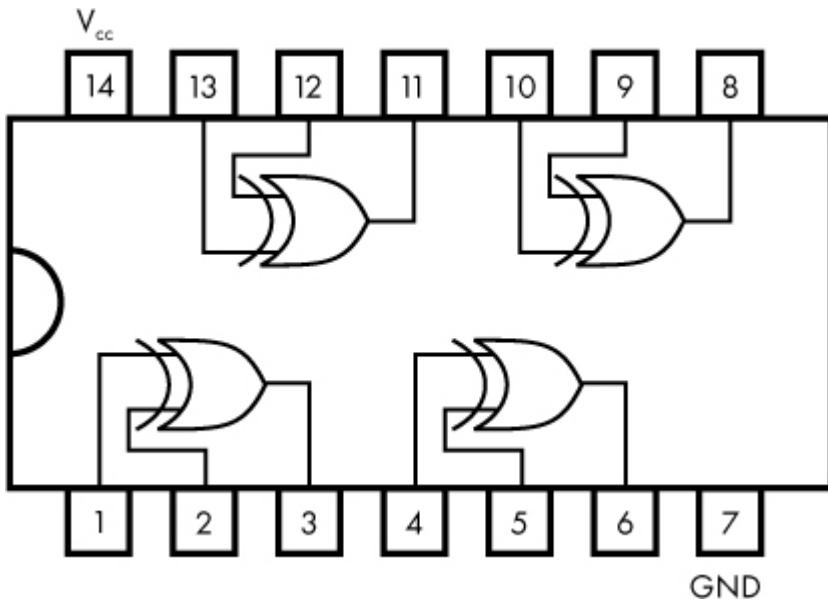


Figure 5-23: Pinout diagram for the 7486 XOR integrated circuit

Figure 5-24 provides the wiring diagram for a half adder. Keep reading past the figure for more details on how to build this circuit.

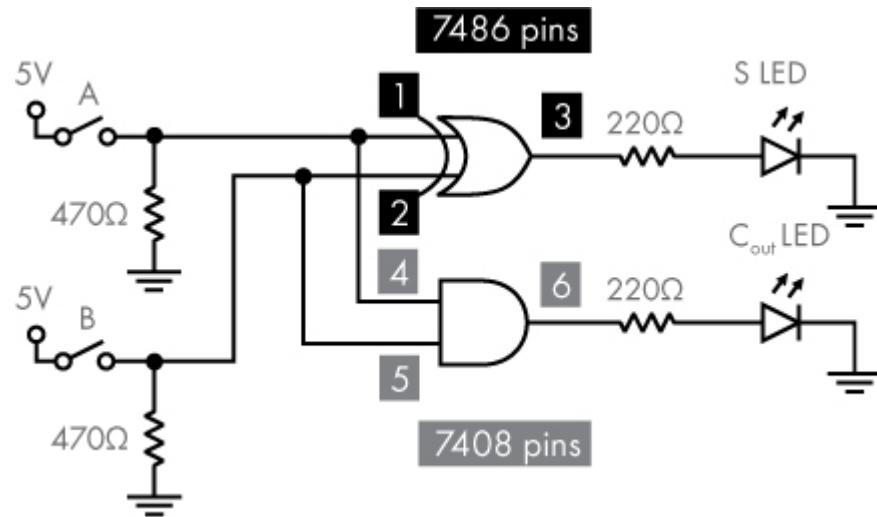


Figure 5-24: Half adder built from XOR and AND gates

Figure 5-24 shows the connections for switches, with pull-down resistors, and for LEDs, with current-limiting resistors. Also note the pin numbers on the 7486 and 7408 ICs, shown in boxes. Note the black dots found on the wires connecting A and B to the resistors and ICs. The dots represent a connection point—for example, switch A, the 470Ω resistor, pin 1 on the 7486 IC, and pin 4 on the 7408 IC are all connected. Don't forget to connect the 7486 and 7408 ICs to 5V and ground via pins 14 and 7 (not shown in Figure 5-24), respectively.

Figure 5-25 shows how this circuit could look when implemented on a breadboard.

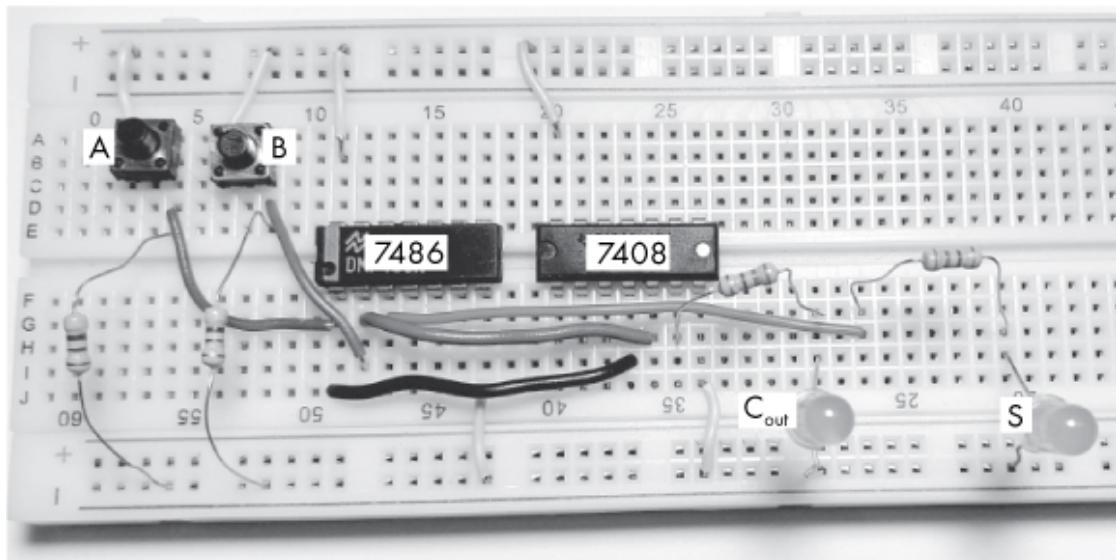


Figure 5-25: Half adder built from XOR and AND gates

Once you've constructed this circuit, try all combinations of inputs A and B to confirm that the outputs match the expected values as shown in the half adder truth table (Table 5-1).

6

MEMORY AND CLOCK SIGNALS



In the previous chapters we saw how digital logic gates can be combined to produce useful combinational logic circuits where the output is a function of the inputs. In this chapter, we look at sequential logic circuits. These circuits have memory, the ability to store a record of the past. We cover some specific kinds of memory devices: latches and flip-flops. We also learn about clock signals, which are a way to synchronize multiple circuit components.

Sequential Logic Circuits and Memory

Let's now examine a type of digital circuit known as a *sequential logic circuit*. A sequential logic circuit's output depends not only on its present set of inputs, but also on past inputs to the circuit. In other words, a sequential logic circuit has some knowledge of its own previous history or state. Digital devices store a record of past state in what is known as *memory*, a component that allows for storage and retrieval of binary data.

Let's consider a simple example of sequential logic: a coin-operated vending machine. A vending machine has at least two inputs: a coin slot

and a vend button. For simplicity, let's assume that the vending machine only vends one type of item and that item costs one coin. The vend button doesn't do anything unless a coin has been inserted. If the vending machine were based on *combinational logic*, where the state is determined by present inputs only, then a coin would have to be inserted at the same instant that the vend button is pressed.

Fortunately, that's not how vending machines work! They have memory that tracks whether a coin has been inserted. When we press the vend button, the sequential logic in the vending machine checks its memory to see if a coin was previously inserted. If so, the machine dispenses an item. We'll build on this sequential logic example later in the chapter.

Sequential logic is possible because of memory. Memory stores binary data, and its capacity for storage is measured in bits or bytes. Modern computing devices such as smartphones usually have at least 1GB of memory. That's over 8 billion bits! Let's begin with something a little simpler: a memory device with 1 bit of memory.

The SR Latch

A *latch* is a type of memory device that remembers one bit. The *SR latch* has two inputs: S (for set) and R (for reset), and an output called Q, the single bit that's “remembered.” When S is set to 1, output Q becomes 1 too. When S goes to 0, Q remains equal to 1, because the latch remembers this previous input. This is the essence of memory—the component remembers a previous input, even if that input changes. When R is set to 1, this is an indicator to reset/clear the memory bit, so output Q becomes 0. Q will remain 0 even if R goes back to 0.

We summarize the behavior of an SR latch in Table 6-1.

Table 6-1: Operation of an SR Latch

S	R	Q (output)	Operation
0	0	0	Initial state
1	0	1	Set Q = 1
0	1	0	Reset Q = 0
1	1	0	Reset Q = 0 (overrides Set)

S	R	Q (output)	Operation
0	0	Maintain previous value	Hold
0	1	0	Reset
1	0	1	Set
1	1	X	Invalid

By design, setting S to 1 and R to 1 at the same time is an invalid input, and the value of Q in this scenario is undefined. In practice, attempting this causes Q to go to 1 or 0, but we can't reliably say which. Besides, it doesn't make sense to try to set and reset the latch at the same time. The circuit diagram symbol for an SR latch is shown in Figure 6-1.

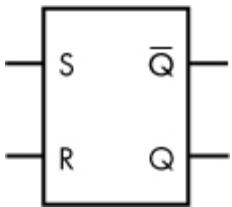


Figure 6-1: The circuit diagram symbol for an SR latch

In Figure 6-1 there's an additional output: \bar{Q} . Read this as “complement of Q,” “NOT Q,” or “inversion of Q.” It's simply the opposite of Q. When Q is 1, \bar{Q} is 0, and vice-versa. It can be useful to have both Q and \bar{Q} available, and as you'll see, the design of such a circuit lends itself to including this output without extra effort.

We can implement an SR latch fairly simply using only two NOR gates and some wires. That said, understanding how the design works takes some thought. Consider the circuit shown in Figure 6-2, which is an implementation of an SR latch.

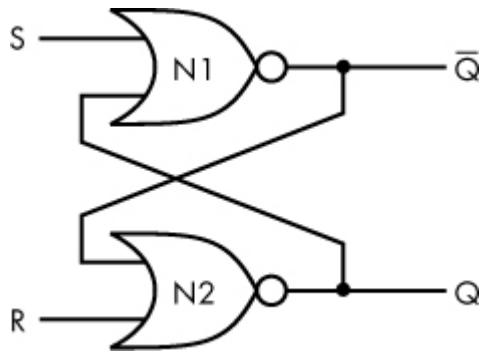


Figure 6-2: SR latch implemented with cross-coupled NOR gates

In Figure 6-2, we have two NOR gates in what is known as a *cross-coupled configuration*. As a reminder, a NOR gate only outputs a 1 if both inputs are 0; otherwise, it outputs a 0. The output from N1 feeds into N2's input, and the output from N2 feeds into N1's input. The inputs are S and R. The outputs are Q and Q̄. Let's examine how the circuit works by activating and clearing various inputs, examining the outputs as we go. Assume that initially S is 0, and R is 1.

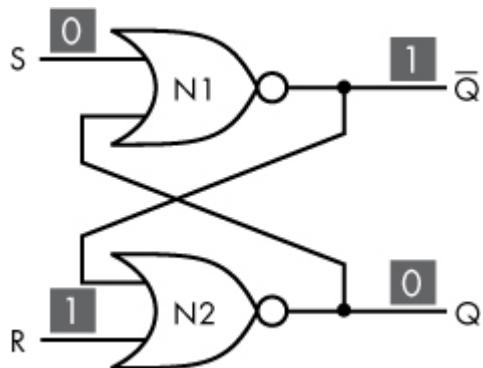


Figure 6-3: SR latch, initial state

Initial state ($S = 0, R = 1$)

1. $R = 1$, so the output of N2 is 0.
2. The output of N2 is fed into N1.
3. $S = 0$, so the output of N1 is 1.
4. Initially $Q = 0$.

Summary: when R goes high, the output goes low (see Figure 6-3).

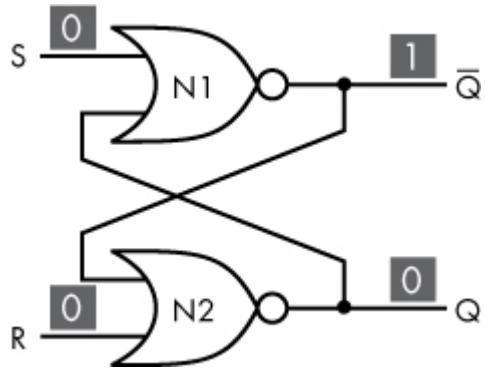


Figure 6-4: SR latch, inputs low

Next, clear all inputs ($S = 0, R = 0$)

1. R goes to 0.
2. The other input to N2 is still 1, so the output of N2 is still 0.
3. Therefore, Q still equals 0.

Summary: the circuit remembered the previous output state (see Figure 6-4).

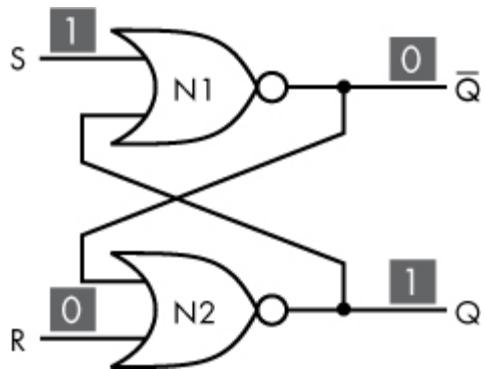


Figure 6-5: SR latch, S goes high

Next, activate the S input ($S = 1, R = 0$)

1. S goes to 1.
2. This causes the output of N1 to go to 0.
3. The inputs to N2 are now 0 and 0, so the output of N2 is 1.
4. Therefore, Q now equals 1.

Summary: setting S high causes the output to go high (see Figure 6-5).

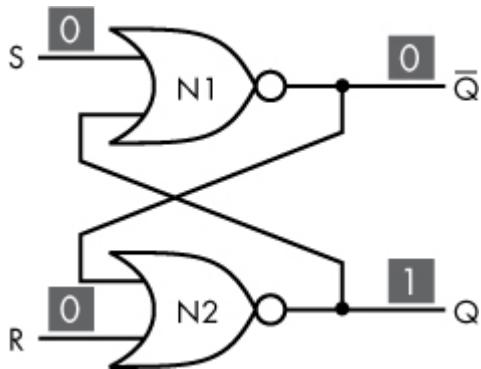


Figure 6-6: SR latch, S goes low

Finally, clear all inputs again ($S = 0, R = 0$)

1. S goes to 0.
2. The other input to N1 is still 1, so the output of N1 is still 0.
3. The inputs to N2 are unchanged.
4. Therefore, Q still equals 1.

Summary: the circuit remembered the previous output state (see Figure 6-6).

Putting all of that together, we've just described the desired behavior of an SR latch, as previously summarized in Table 6-1. When S (set) is 1, the output, Q, goes to 1 and stays at 1 even when S goes back to 0. When R (reset) is 1, the output, Q, goes to 0 and stays at 0 even when R goes back to 0. In this way, the circuit remembers either a 1 or 0, so we have a device with 1 bit of memory! Even though there are two outputs (Q and \bar{Q}), both are just different representations of the same saved bit. Remember, setting both S = 1 and R = 1 at the same time is an invalid input.

To understand the behavior of the SR latch, we've looked at how the circuit behaves when the inputs are held high and then set low. However, S and R typically just need to be “pulsed.” When the circuit is

at rest, both S and R are low. When we want to change its state, we have no reason to hold S or R high for long; we just need to quickly set it high and then back to low—a simple pulse of the input.

UNIVERSAL LOGIC GATES

We just demonstrated how an SR latch can be constructed with NOR gates. In fact, NOR gates can be used to create any other logic circuit, not just the SR latch. The NOR gate is known as a *universal logic gate*; it can be used to implement any logical function. The same is true of NAND.

Now that we've investigated the internal design of an SR latch, we can optionally go back to using the symbol in Figure 6-1 to represent an SR latch. When we do this, we no longer need to concern ourselves with the internals of a latch. This is another example of encapsulation! We take a design and put it in a “black box,” which makes it easier to use that design without worrying about the internal details. I find it helpful to think of the SR latch in simple terms: it's a 1-bit memory device that has a state Q of either 1 or 0. The S input sets Q to 1, and the R input resets Q to 0.

NOTE

Please see Project #6 on page 104, where you can build an SR latch.

Using the SR Latch in a Circuit

Now that we have a basic memory device, the SR latch, let's use it in an example circuit. Let's return to our vending machine example and design a vending machine circuit that uses a latch. The circuit has the following requirements:

- The circuit has two inputs: a COIN button and a VEND button. Pressing COIN represents inserting a coin. Pressing VEND causes

the machine to vend an item (the circuit will just turn on an LED to represent vending an item).

- The circuit has two LED outputs: COIN LED and VEND LED. COIN LED lights when a coin has been inserted. VEND LED lights to indicate that an item is being vended.
- The machine won't vend an item unless a coin has been first inserted.
- For simplicity, assume only one coin can be inserted. Inserting additional coins does not change the state of the circuit.
- Normally after a vending operation occurs, we'd expect the circuit to reset itself and go back to the "no coin" state. However, for simplicity of design, we'll skip the automatic reset in favor of a manual reset.

At a conceptual level, our vending circuit will be implemented as shown in Figure 6-7.

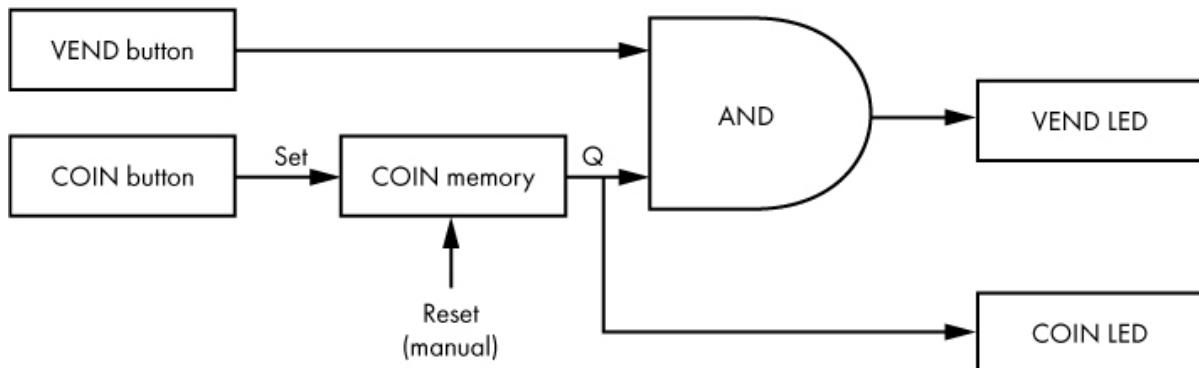


Figure 6-7: Conceptual vending machine circuit with manual reset

Let's walk through Figure 6-7. When you press the COIN button, the COIN memory device (an SR latch) stores the fact that a coin was inserted. The memory device then outputs a 1, indicating that a coin has been inserted, and the COIN LED lights up. When you press the VEND button, if a coin was previously inserted, the AND gate outputs 1, and the VEND LED lights. On the other hand, if you press the VEND button without previously inserting a coin, nothing happens. To

clear the COIN LED and reset the device, you must manually set the Reset input to 1.

NOTE

Please see Project #7 on page 105, where you can build the vending machine circuit just described.

This basic vending machine circuit demonstrates a practical use of memory in a circuit. Since our circuit design includes a memory element, the VEND button can behave differently based on whether a coin was inserted in the past. However, once the COIN bit is set in memory, it stays set until the circuit is manually reset. That's not ideal, so let's update our circuit so that it resets automatically after a vend operation occurs.

Once the machine vends an item, we expect the COIN bit to be set back to 0, since the action of vending "uses" the coin. In other words, vending should also cause the coin memory to reset. To implement this logic, we can connect the output of the AND gate to the memory reset, as shown in Figure 6-8. That way, when the VEND LED turns on, the COIN memory resets.

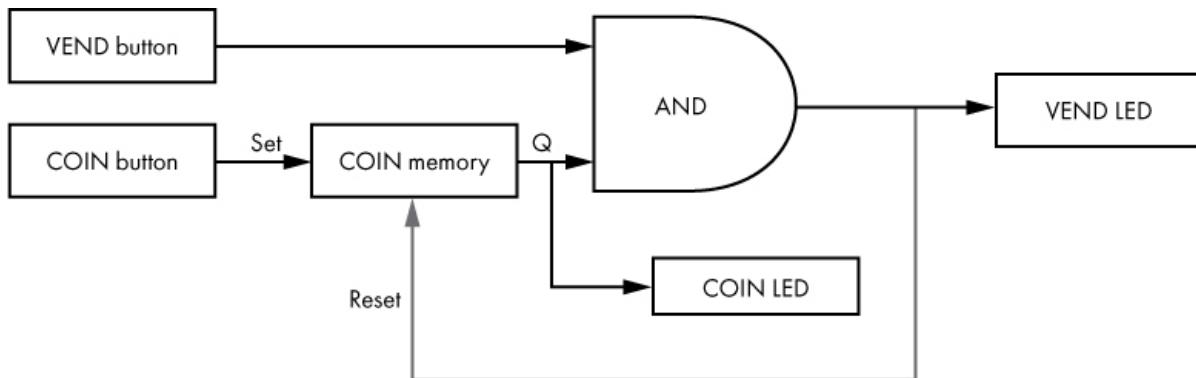


Figure 6-8: Conceptual vending machine circuit with automatic reset

The system shown in Figure 6-8 will reset the circuit during vending, but there's a problem with this design. Can you spot it? The problem may not be obvious. If you completed the last project, you may want to try this type of reset on the circuit you just built. Connect a wire from the output of the AND gate to the R input in the SR latch,

press COIN, then press VEND. Spoilers ahead, so don't read on until you've given this a try, either mentally or on a breadboard!

The problem is that although the reset works as expected, it happens so quickly that the VEND LED immediately turns off, or more likely, the VEND LED never comes on. Here we have an example of a design that technically works but works so quickly that the user of the device can't see what happened. This is a fairly common problem in user interface design. The devices and programs we build often operate so quickly that we must deliberately slow things down a bit so that the user can keep up. In this case, a solution would be to introduce a delay on the reset line so that the VEND LED has time to light up for a second or two before the reset occurs. This is shown in Figure 6-9.

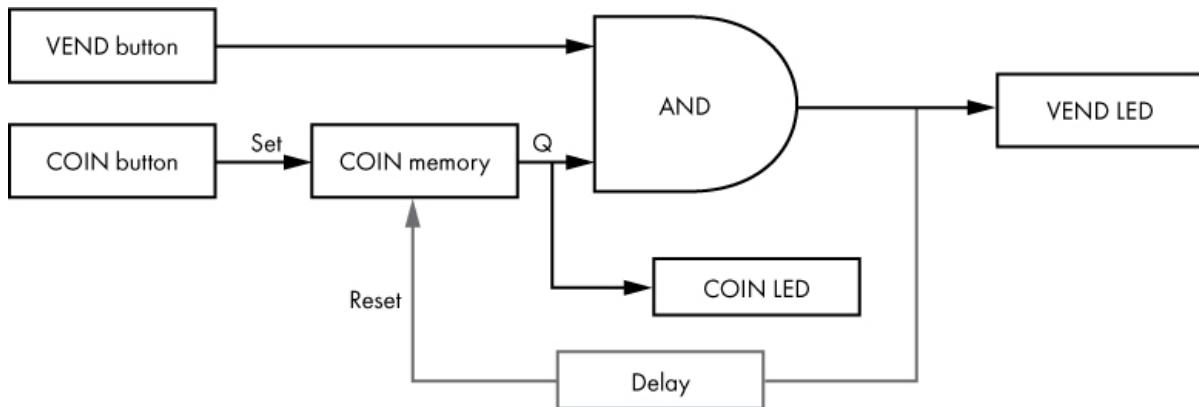


Figure 6-9: Conceptual vending machine circuit with automatic delayed reset

How can we go about adding a delay? One approach is to use a capacitor. A *capacitor* is an electrical component that stores energy. It has two terminals. When current flows to the capacitor, the capacitor charges. The measure of a capacitor's ability to store electric charge is called *capacitance*, which is measured in *farads*. One farad is a very large value, so we typically rate capacitors in *microfarads*, abbreviated μF .

When the capacitor is not charged, it acts like a short circuit. Once the capacitor is charged, it acts like an open circuit. The time it takes to charge or discharge a capacitor is controlled by the capacitor's capacitance value and resistance in the circuit. Larger capacitance and resistance values result in a capacitor taking longer to charge. So we can

use a capacitor and resistor to introduce a delay in our circuit caused by the time it takes the capacitor to charge.

NOTE

Please see Project #8 on page 107, where you can add a delayed reset to your vending machine circuit.

So far in this chapter we've restricted our exploration of memory to single-bit devices. Although 1 bit of memory has limited applicability, in Chapter 7 we'll see how we can use sets of single-bit memory cells together to represent larger amounts of data.

Clock Signals

As circuits become more complex, we often need to keep the various elements synchronized so that they all change state at the same time. We may have to do this for circuits with multiple memory devices, where we'd like to ensure that all the stored bits can be set at the same time. This is especially true when we need to consider sets of bits together. We can synchronize multiple circuit components with a clock signal. A *clock signal*, or just a *clock*, alternates its voltage level between high and low. Typically, the signal alternates on a regular cadence, where the signal is high half the time and low the other half. We call this type of signal a *square wave*. Figure 6-10 shows a 5V square wave clock signal graphed over time.

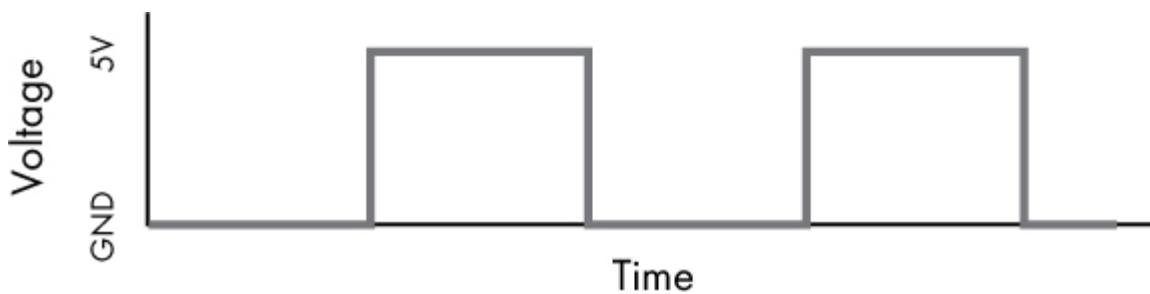


Figure 6-10: A 5V square wave clock signal

A single iteration of the voltage rising and falling is a *pulse*. A complete oscillation from low to high and back to low (or the reverse) is

a *cycle*. We measure the *frequency* of the clock signal in cycles per second, or *hertz (Hz)*. In Figure 6-11, the frequency of the clock signal shown is 2Hz, because the signal completes two full oscillations in one second.

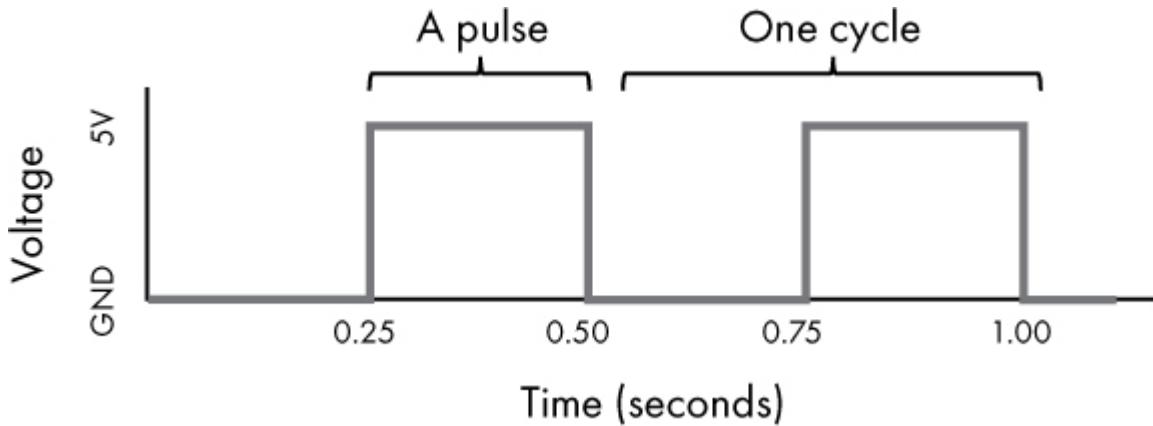


Figure 6-11: A 2Hz clock signal

When a circuit uses a clock, all components that need to be synchronized are connected to the clock. Each component is designed to allow state changes only when a clock pulse occurs. Clock-driven components typically trigger state changes on either the rising edge or the falling edge of the pulse. A component that changes state on the rising pulse edge is known as *positive edge-triggered*, and a component that changes state on the falling pulse edge is known as *negative edge-triggered*. Figure 6-12 provides an example of a rising and falling edge.



Figure 6-12: Pulse edges illustrated

The graphics in this book illustrate pulse edges as vertical lines; this implies an instantaneous change from low to high or vice versa. In practice, however, it takes time to change states, but for the purposes of our discussion, let's imagine the state change happening instantaneously.

NOTE

Please see Project #9 on page 109, where you can use your SR latch as a manual clock.

JK Flip-Flops

A 1-bit memory device that uses a clock is a *flip-flop*. There's some overlap in usage of the terms *latch* and *flip-flop*, but here we use *latch* to mean memory devices without a clock, and *flip-flop* to mean clocked memory devices. You may see the terms used interchangeably or with different connotations elsewhere.

Let's examine a specific clocked memory device, the *JK flip-flop*. The JK flip-flop is a conceptual extension of the SR latch, so let's compare the two. The SR latch has input S to set the memory bit and input R to reset the memory bit; similarly, the JK flip-flop has input J to set and input K to reset. The SR latch immediately changes state when S or R is set high, but the JK flip-flop only changes state on a clock pulse. The JK flip-flop also adds an additional feature: when both J and K are set high, the output toggles a single time from low to high or high to low. This is summarized in Table 6-2.

Table 6-2: Comparison of SR Latch and JK Flip-Flop

	SR latch	JK flip-flop
Changes state	Immediately when S or R goes high	Only on clock pulse if J or K are high
Set	S = 1	J = 1
Reset	R = 1	K = 1
Toggle	Not applicable	J = 1 and K = 1

When representing a JK flip-flop in a diagram, the symbols shown in Figure 6-13 can be used.

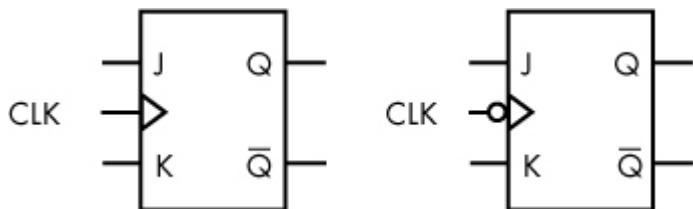


Figure 6-13: JK flip-flops, positive edge-triggered (left), negative edge-triggered (right)

Figure 6-13 shows two versions of the JK flip-flop. The one on the left is positive edge-triggered, meaning it changes state on the rising edge of the clock pulse. On the right, we have the symbol for a negative edge-triggered JK flip-flop (note the circle on the CLK input); it changes state on the falling edge of the clock pulse. The two devices behave identically otherwise.

So a JK flip-flop is a 1-bit memory device that only changes state when it receives a clock pulse. It's quite similar to an SR latch, except that a clock controls its state changes, and it has the ability to toggle its value. Table 6-3 summarizes the behavior of the JK flip-flop.

Table 6-3: Summary of the Functionality of a JK Flip-Flop

J	K	Clock	Q (output)	Operation
0	0	Pulse	Maintain previous value	Hold
0	1	Pulse	0	Reset
1	0	Pulse	1	Set
1	1	Pulse	Inverse of previous value	Toggle

We won't go through a step-by-step walkthrough of the JK flip-flop as we did for the SR latch. Instead, the best way to understand a JK flip-flop is to work with one directly.

NOTE

Please see Project #10 on page 111, where you can go hands-on with a JK flip-flop.

T Flip-Flops

Connecting J and K and treating them as a single input creates a flip-flop that only does one of two things on a clock pulse: it either toggles or maintains its value. To see why this is the case, review Table 6-3 and note the behavior when both J and K are 0 or both J and K are 1.

Connecting J and K is a commonly used technique, and a flip-flop that behaves in this way is a *T flip-flop*. Figure 6-14 shows the symbol for a T flip-flop on the right.

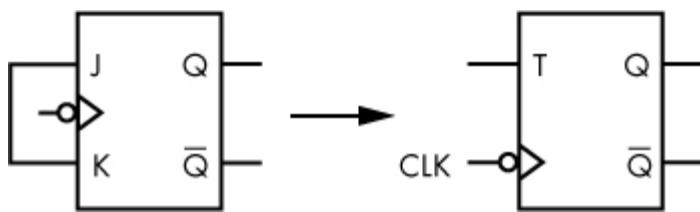


Figure 6-14: A JK flip-flop with J and K connected is known as a T flip-flop.

So a T flip-flop simply toggles its value on clock pulse, when T is 1. Table 6-4 summarizes the behavior of the T flip-flop.

Table 6-4: Summary of the Functionality of a T Flip-Flop

T	Clock	Q	Operation
0	Pulse	Maintain previous value	Hold
1	Pulse	Inverse of previous value	Toggle

Using a Clock in a 3-Bit Counter

To illustrate the use of a clock in a circuit, let's build a 3-bit counter—a circuit that counts from 0 to 7 in binary. This circuit has three memory elements, each representing one bit of a 3-bit number. The circuit takes a clock input, and when a clock pulse occurs, the 3-bit number increments (increases by 1). Since all the bits represent a single number,

it's important that we synchronize their state changes with a clock. Let's use T flip-flops to accomplish this.

First, see Table 6-5 as a review of counting in binary using a 3-bit number.

Table 6-5: Counting in Binary with 3 Bits

Binary	Decimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Table 6-5 presents our 3-bit number as a single value on each row. Let's now assign each of the bits to memory elements labeled Q0, Q1, and Q2. Q0 is the least significant bit and Q2 is the most significant bit, as shown in Table 6-6.

Table 6-6: Counting in Binary, Each Bit Assigned to a Separate Memory Element

All 3 bits	Q2	Q1	Q0	Decimal
000	0	0	0	0
001	0	0	1	1
010	0	1	0	2
011	0	1	1	3
100	1	0	0	4
101	1	0	1	5
110	1	1	0	6
111	1	1	1	7

If we look at the Q columns in Table 6-6 individually, we can see a pattern emerge. As we count, Q0 toggles every time. Q1 toggles when

Q0 was previously 1. Q2 toggles when both Q1 and Q0 were previously 1. In other words, apart from Q0, each bit toggles on the next count when all the preceding bits are 1. T flip-flops are perfect for implementing this counter, since toggling is what they do! Let's look at how we can build a circuit to do this, shown in Figure 6-15.

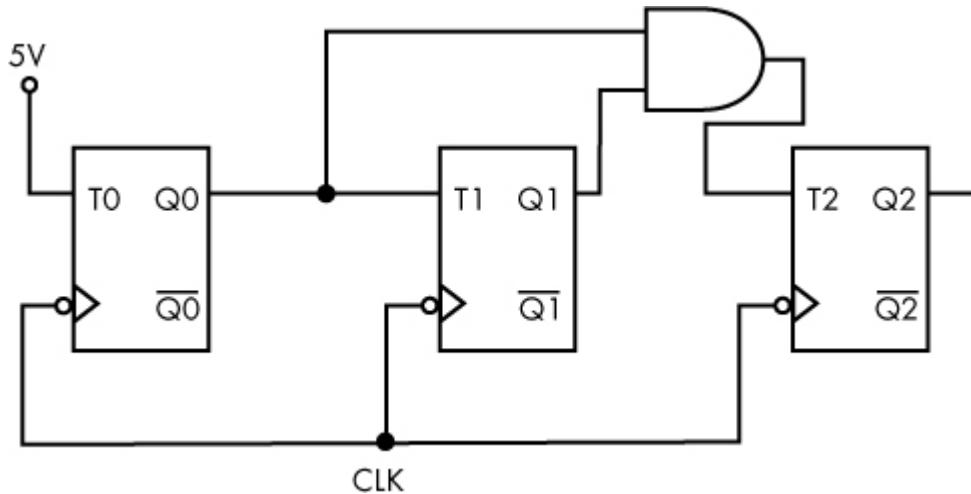


Figure 6-15: A 3-bit counter built from T flip-flops

In Figure 6-15, all three T flip-flops use the same clock signal, so they are synchronized. T0 is connected to 5V, so Q0 toggles every time the clock pulses. T1 is connected to Q0, so a clock pulse causes Q1 to toggle only when Q0 is high. T2 is connected to Q0 AND Q1, so Q2 only toggles on a clock pulse when Q0 and Q1 are both high.

NOTE

Please see Project #11 on page 113, where you can build your very own 3-bit counter.

Consider how we might use such a counter in conjunction with the vending machine circuit we designed earlier. Instead of simply tracking whether a coin is inserted or not, we can track the count of coins inserted, at least up to seven coins! For a vending machine counter to be useful, it also needs to be able to count down, since vending an item should decrease the coin count. I won't cover the specifics of how to add a counter to the vending circuit here, but feel free to experiment on

your own. Designs for counter circuits that count up and down are available online, or you can use an up/down counter IC like the 74191.

We've constructed a counter from T flip-flops, which was built from JK flip-flops, which are digital logic circuits based on transistors! This again demonstrates how encapsulation allows us to build complex systems, hiding the details along the way.

Summary

In this chapter, we covered sequential logic circuits and clock signals. You learned that unlike combinational logic circuits, sequential circuits have memory, a record of past state. You learned about the SR latch, a simple single-bit memory device. We saw how synchronizing multiple circuit components, including memory devices, can be accomplished with a clock signal, an electrical signal that alternates its voltage level between high and low. A clocked single-bit memory device is known as a flip-flop, which allows for state changes to only occur in synchronization with the clock signal. You learned how JK flip-flops work, how T flip-flops can be constructed from JK flip-flops, and finally how a clock and T flip-flops can be used together to create a 3-bit counter.

Memory and clocks are key components of modern computing devices, and in the next chapter, we'll see how they play a role in today's computers. There you'll learn about computer hardware—memory, processor, and I/O.

PROJECT #6: CONSTRUCT AN SR LATCH USING NOR

GATES

In this project, you'll build an SR latch on a breadboard. You'll connect output Q to an LED to easily observe the state. You should test setting S and R high and low and observe the output.

For this project, you'll need the following components:

- Breadboard
- LED
- Current-limiting resistor to use with your LED (approximately 220Ω)
- Jumper wires
- 7402 IC (contains four NOR gates)
- 5-volt power supply
- Two 470Ω resistors
- Two switches or pushbuttons that fit a breadboard
- Optional: An additional 220Ω resistor and another LED

As a reminder, see the sections “Buying Electronic Components” on page 333 and “Powering Digital Circuits” on page 336 if you need help on those topics. Also, review Project #4 on page 68 for a reminder about how to use buttons/switches with pull-down resistors. Connect your components as shown in Figure 6-16 to build an SR latch. Note that the NOR gates are arranged differently within the 7402 IC as compared to the layout of gates in other ICs like the 7408 (AND gates) and the 7432 (OR gates), so be sure to use the right pins for inputs and outputs.

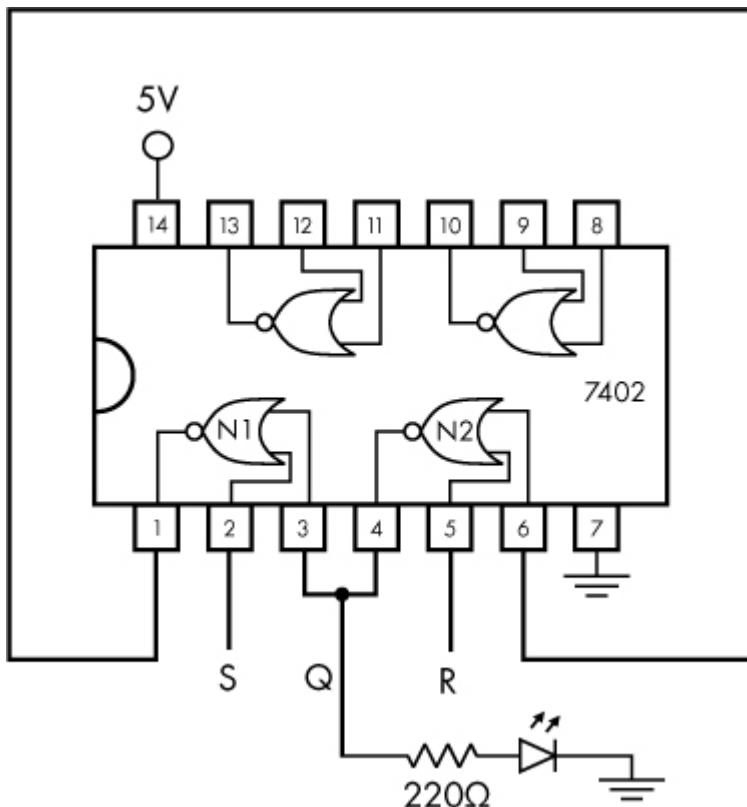


Figure 6-16: Wiring diagram for an SR latch built from a 7402 IC

Once you've constructed the SR latch circuit as shown in Figure 6-16, connect S and R to buttons (or switches) with pull-down resistors, as shown in Figure 6-17. This allows you to easily set the value of S or R just by pressing a button.

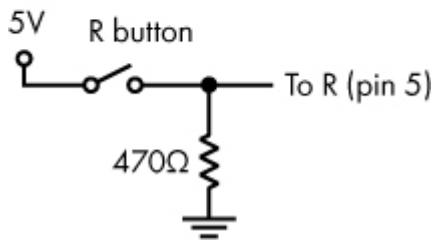
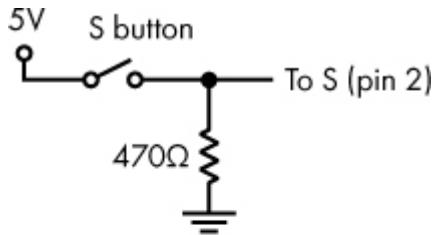


Figure 6-17: Using buttons and pull-down resistors to control inputs S and R

Once you have connected your buttons to the SR latch, try setting S or R to high or low by pressing and releasing the button. Observe the results. Does Q turn on when you press S and stay on even after you release S? Does Q turn off when you press R and stay off even after you release R? If you want to also see the value of \bar{Q} , which should always be the opposite of Q, just connect another 220Ω resistor and another LED to pins 1 and 6 of the IC.

When you initially apply power, the output will be in an unpredictable state. That is, the circuit may start up with either $Q = 0$ or $Q = 1$. Or maybe your circuit reliably starts up with Q as a certain value. The reason for this unpredictability is that this design leads to a *race condition*. If $S = 0$ and $R = 0$ when power is applied, both N1 and N2 try to output a 1. One of them does this slightly faster (thus, a *race*). If N1 outputs a 1 first, N2 goes low and Q is 0. If N2 outputs a 1 first, N1 goes low and Q is 1. This can be addressed by holding the R button down during startup (to force $Q = 0$) and then releasing the R button after startup.

Keep this circuit around, we'll use it in the next project.

PROJECT #7: CONSTRUCT A BASIC VENDING MACHINE

CIRCUIT

In this project, you'll build the vending machine circuit described earlier in this chapter. You can reuse your SR latch from the last project as the memory unit. Be sure to use current limiting resistors on your LEDs and pull-down resistors for your button inputs. Test the circuit to make sure it works as expected. To reset the circuit, press the R button on the SR latch.

For this project, you'll need the following components:

- The 7402 SR latch on a breadboard you constructed in Project #6
- An additional LED
- An additional current-limiting resistor to use with your LED (approximately 220Ω)
- Jumper wires
- 7408 IC (contains four AND gates)
- An additional pushbutton or switch that will fit a breadboard
- An additional pull-down resistor to use with your button (approximately 470Ω)

As a reminder, see the sections “Buying Electronic Components” on page 333 and “Powering Digital Circuits” on page 336 if you need help on these topics.

In the circuit diagram shown in Figure 6-18, the IC pin numbers are indicated in boxes. Although they are not shown in the diagram, be sure to connect both the 7402 and the 7408 chips to 5V and ground (pins 14 and 7, respectively).

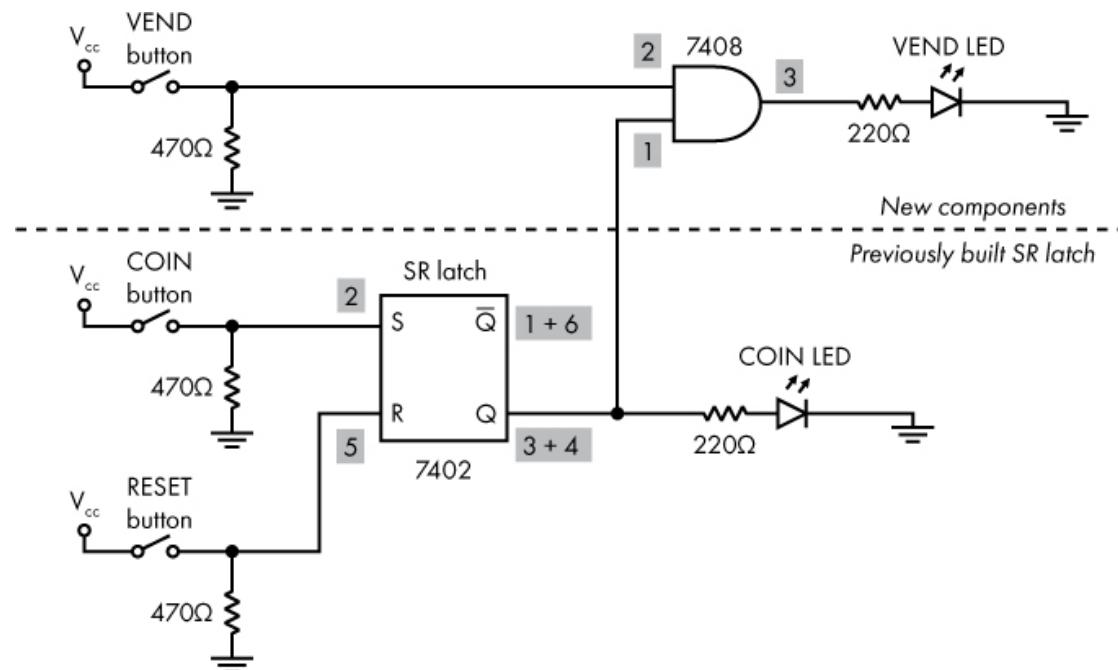


Figure 6-18: Wiring diagram for a basic vending machine circuit

The bottom portion of Figure 6-18 is the circuit you built in the previous project. The only difference is that now the S button represents the COIN button and the output Q LED now represents the COIN indicator LED. To build the full circuit, you only need to add the top portion of the circuit and connect the two parts together as shown.

Once your circuit is built, you should see that when you press the COIN button, the COIN LED lights. Pressing the VEND button should cause the VEND button to light, but only if the COIN LED is already lit. Press the RESET button to reset the circuit.

Keep this circuit around, we'll use it in the next project.

PROJECT #8: ADD A DELAYED RESET TO THE VENDING MACHINE CIRCUIT

In this project, you'll add a delayed reset to the vending machine circuit from Project #7. You'll need the following components:

- The vending machine circuit you constructed in Project #7
- $4.7\text{k}\Omega$ resistor
- $220\mu\text{F}$ electrolytic capacitor
- Jumper wires

There are multiple types of capacitors; a discussion of the various types is outside the scope of this book. For this project, you'll use an *electrolytic capacitor* (Figure 6-19). When connecting your capacitor, note that electrolytic capacitors are polarized, meaning one pin is negative and one is positive. Look for a negative sign or arrow indicating the negative terminal. Sometimes the negative terminal is shorter. In Figure 6-21, the negative terminal should connect to ground.

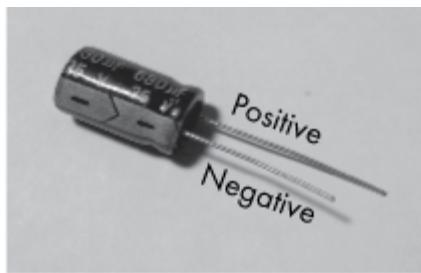


Figure 6-19: An electrolytic capacitor. The shorter pin with a stripe/arrow is the negative pin.

Figure 6-20 shows circuit diagram symbols for capacitors. On the left is the symbol for a nonpolarized capacitor. In the middle and on the right are symbols used to represent polarized capacitors. Both polarized symbols provide a means of identifying the positive and negative terminals of the capacitor.

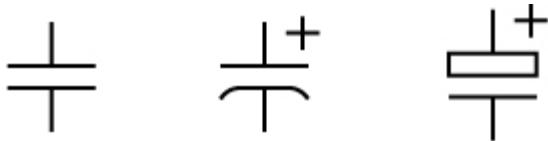


Figure 6-20: Circuit diagram symbols for capacitors

Figure 6-21 shows how you can add the capacitor-based delayed reset to the vending machine circuit, replacing the manual reset. Keep reading past the figure for more details on how to build this circuit.

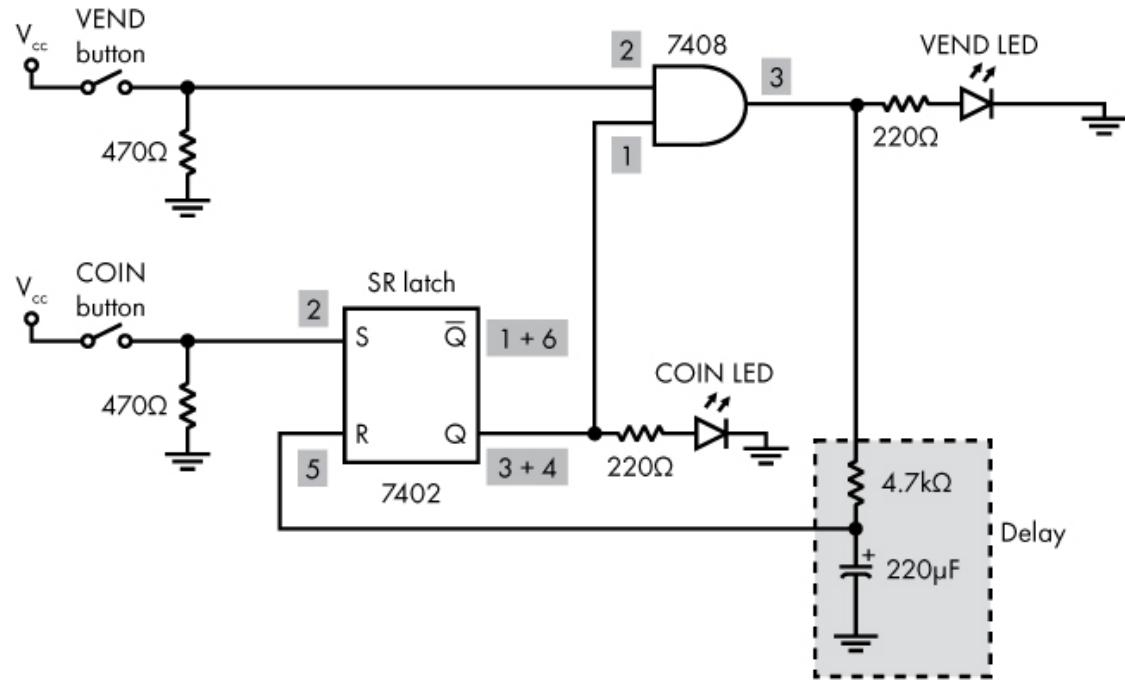


Figure 6-21: Wiring diagram for a vending machine circuit with delayed reset

If you still have a manual reset switch or button connected to R (pin 5 on the 7402 chip), be sure to disconnect it, as its presence will interfere with the delayed reset operation. In Figure 6-21, note how the VEND output of our circuit (pin 3 on the 7408 chip), which goes high when vending occurs, is connected to the latch's reset input through a new delay component. This new component consists of a resistor and capacitor that, together, introduce a delay of about 1 second to the reset. Let's walk through what happens here:

1. When a vend operation occurs, the output from the 7408 AND gate goes high.

2. The uncharged capacitor initially acts like a short circuit to ground, and the reset R to the latch is kept low, so no reset occurs at first.
3. Since no reset has happened yet, the VEND LED has an opportunity to light.
4. If the VEND button is held down, the AND output stays high, and the capacitor begins to charge.
5. After about 1 second, the capacitor is sufficiently charged and acts like an open circuit, effectively removing the connection to ground.
6. The reset input R to the latch goes high, and a reset occurs.

A few things to note about this design:

- The VEND button must be held down to give the capacitor time to charge.
- The circuit still may start with the COIN LED already on. Just hold VEND to reset. This could be addressed with a power-on reset circuit, but that's outside the scope of this project.
- If adding the reset component causes the entire vending circuit to do nothing, the R input is likely stuck at a high voltage. Check the voltage on pin 5 of the 7402 to see if it is high (anything above 0.8V) when it should be low. If you run into this problem, double-check the values of the $4.7\text{k}\Omega$ resistor and the $220\mu\text{F}$ capacitor. Also check your wiring; a loose connection or a jumper wire in the wrong row can throw things off.
- I chose the capacitance and resistance values because they produce a delay of about 1 second. You could use other values. However, changing these values runs the risk of causing the voltage at the R input to be too high when it should be low, as just noted.

Your completed circuit should look something like the circuit shown in Figure 6-22, although your specific layout will probably vary.

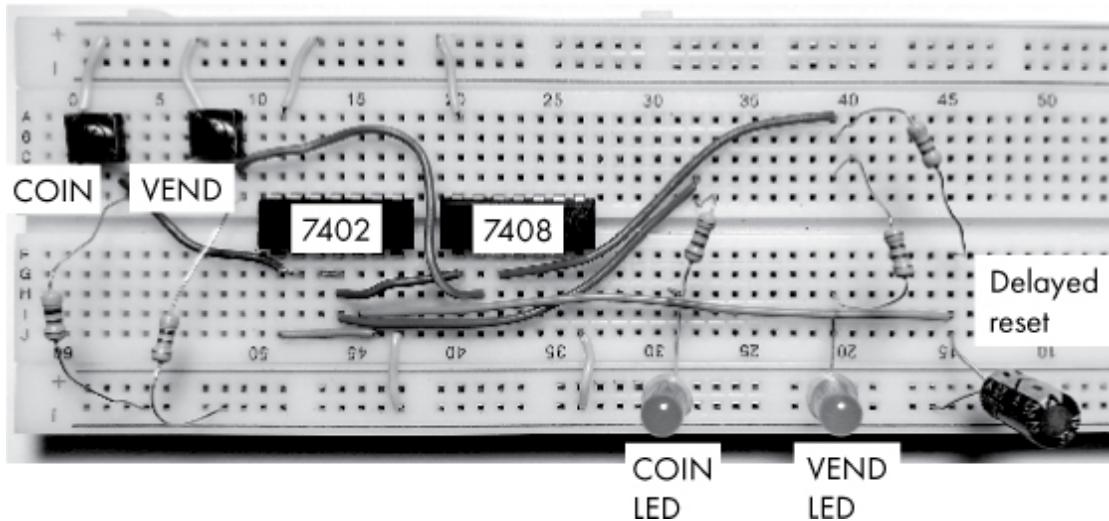


Figure 6-22: Vending machine circuit with delayed reset on a breadboard

I recommend that you keep the SR latch part of your circuit intact, as you'll use it again in the following projects. You can remove the other components from the board, but keep the portion from Project #6. Or you can just build another SR latch when you need to.

PROJECT #9: USING A LATCH AS A MANUAL CLOCK

You're going to need a clock signal for the projects later in the chapter. In this project, you'll configure your previously constructed SR latch as a manual clock.

As you learned earlier, a clock input needs to alternate between high voltage and low voltage. You could try to implement a clock by moving a wire between ground and 5V. That would certainly cause the voltage to alternate, but not in the way you want. When you were moving the wire, some of the time the wire wouldn't be connected to anything. During those moments, the voltage on the input clock pin would "float," and you'd get unpredictable behavior in your circuit. That's not a good option.

Or you could add an oscillator that would automatically generate pulses on a regular cadence, say one pulse every second. That's how clocks typically work in the real world. A common IC is designed for this purpose: the 555 timer. However, for the upcoming exercises you need to be able to carefully observe state changes in your circuits, so what you really need is a *manual* clock, that is, a clock that only goes high or low when you tell it to. In a sense, such a manual clock isn't even really a clock, because it won't alternate states on a regular cadence. That said, whether it's technically a clock or not isn't terribly relevant—we need a device you can use to manually trigger state changes.

You may be tempted to try using a regular pushbutton and a pull-down resistor as a clock, as shown in Figure 6-23. After all, pressing the button makes the voltage go high, and releasing the button makes the voltage go low.

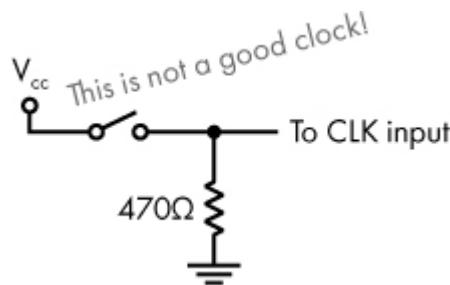


Figure 6-23: Simple switch with pull-down resistor as a CLK input (this won't work very well)

Unfortunately, the design in Figure 6-23 actually makes a very poor manual clock. The problem is that mechanical buttons and switches tend to “bounce.” Internally the switch has metal contacts that connect when the switch is closed. The act of closing the switch results in an initial connection between the contacts, but then the contacts separate and come back together, sometimes multiple times, before the switch finally settles into a closed state. The same thing happens when the switch is opened, except in reverse. A simple button press or flip of a switch results in the voltage jumping high and low multiple times. This is called *switch bounce*, illustrated in Figure 6-24.

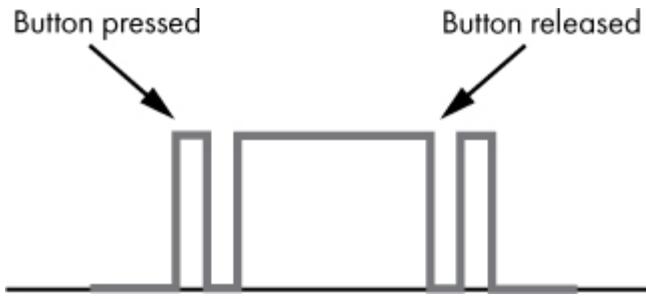


Figure 6-24: Switch bounce, not what we want in a clock

Debounce circuits are hardware options for removing bounce. One such debounce circuit is based on an SR latch, which conveniently, you have already constructed! If you connect S and R to switches, those inputs to the latch still bounce, but the output of the latch (Q) holds its value, as shown in Figure 6-25. This is an effective way of removing switch bounce.

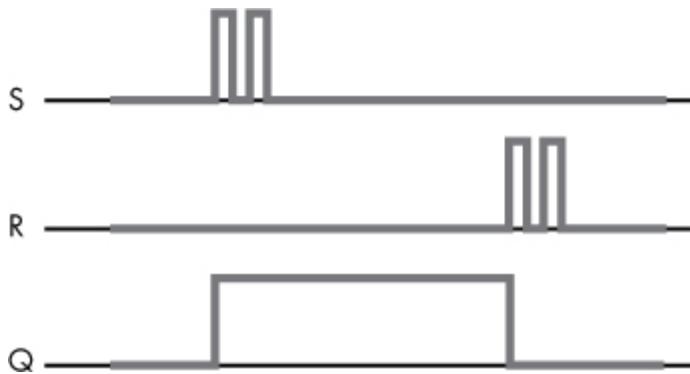


Figure 6-25: An SR latch produces a clean output even when its inputs bounce.

To use an SR latch as a clock, press S to set the clock signal high, and then press R to set the clock signal low. Just don’t press both buttons at the same time! You can use the SR latch you constructed in Project #6 as a clock. If you previously removed the reset button/switch from pin 5 as part of Project #8, connect it again. The complete SR latch as a manual clock should be wired as shown in Figure 6-26.

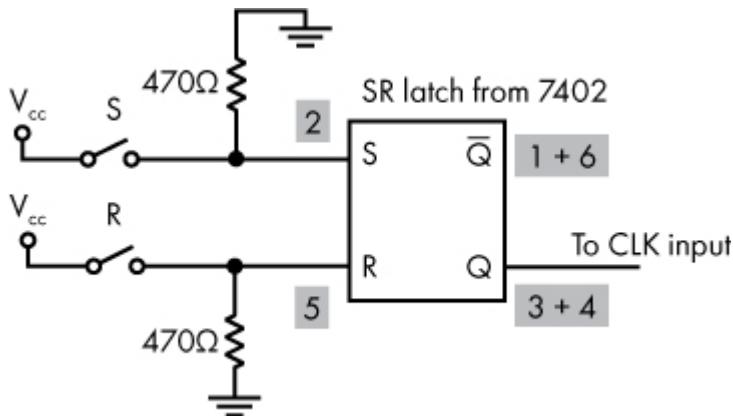


Figure 6-26: A debounced manual clock created from two buttons/switches and an SR latch

Press S to set the clock pulse high, and press R to set the clock pulse low. Now you have a manual clock you can use in the following projects.

PROJECT #10: TEST A JK FLIP-FLOP

Although you could build a JK flip-flop from other gates, it is conveniently sold as an integrated circuit, so you can save yourself some trouble. The 7473 chip contains two negative edge-triggered JK flip-flops. In this project, you'll use this integrated circuit to test the functionality of a single JK flip-flop. You'll try setting J and K either high or low, and then send a clock pulse through the circuit. Connect an LED to the output Q to easily see the state change.

For this project, you'll need the following components:

- The SR latch configured as a clock (discussed in Project #9)
- 7473 IC (contains two JK flip-flops)
- Jumper wires
- LED
- Current-limiting resistor to use with your LED (approximately 220Ω)

As a reminder, see “Buying Electronic Components” on page 333 and “Powering Digital Circuits” on page 336 if you need help on those topics. Figure 6-27 shows the pinout diagram for the 7473 IC.

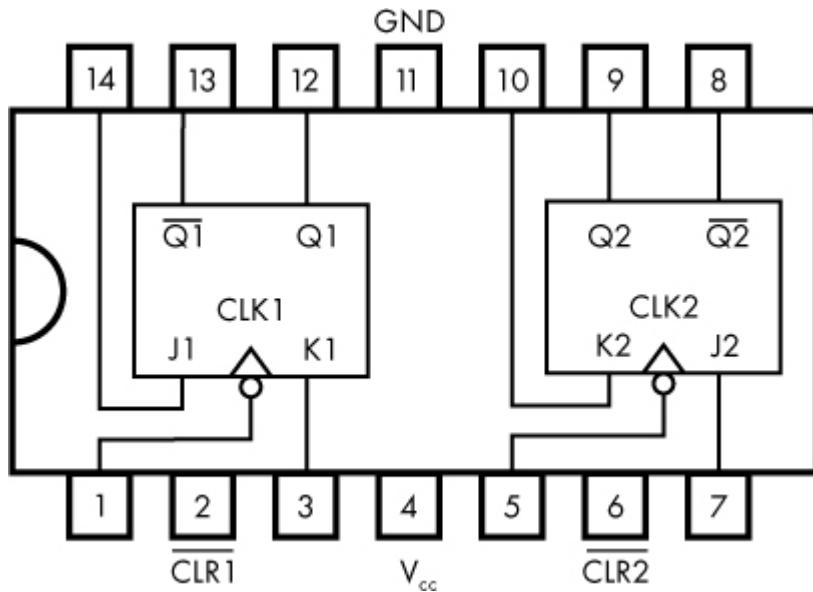


Figure 6-27: Pinout for the 7473 IC

The 7473 IC contains two JK flip-flops as shown in Figure 6-27. Note that the voltage and ground connections aren't in the "usual" locations and instead are pins 4 and 11, respectively. Also, note that the CLK (clock) inputs are marked with a circle, indicating that this circuit is negative edge-triggered; you should expect the state to change when the clock pulse falls. Since you're using an SR latch for your manual clock, this means you'll see the JK state change when you press the SR latch's R input button.

An additional input for each JK flip-flop wasn't mentioned in the chapter: CLR. When this pin is set low, the flip-flop clears the saved bit ($Q = 0$). CLR is asynchronous, meaning it doesn't wait on the clock pulse. The line shown above CLR means it is *active low*, meaning the saved bit is cleared when the input is set low. CLR is also sometimes called Reset or R, not to be confused with the R input of our SR latch. Connect the JK flip-flop's CLR input (pin 2) to 5V to keep your flip-flop from resetting. For testing a single flip-flop, you can connect the chip as shown in Figure 6-28.

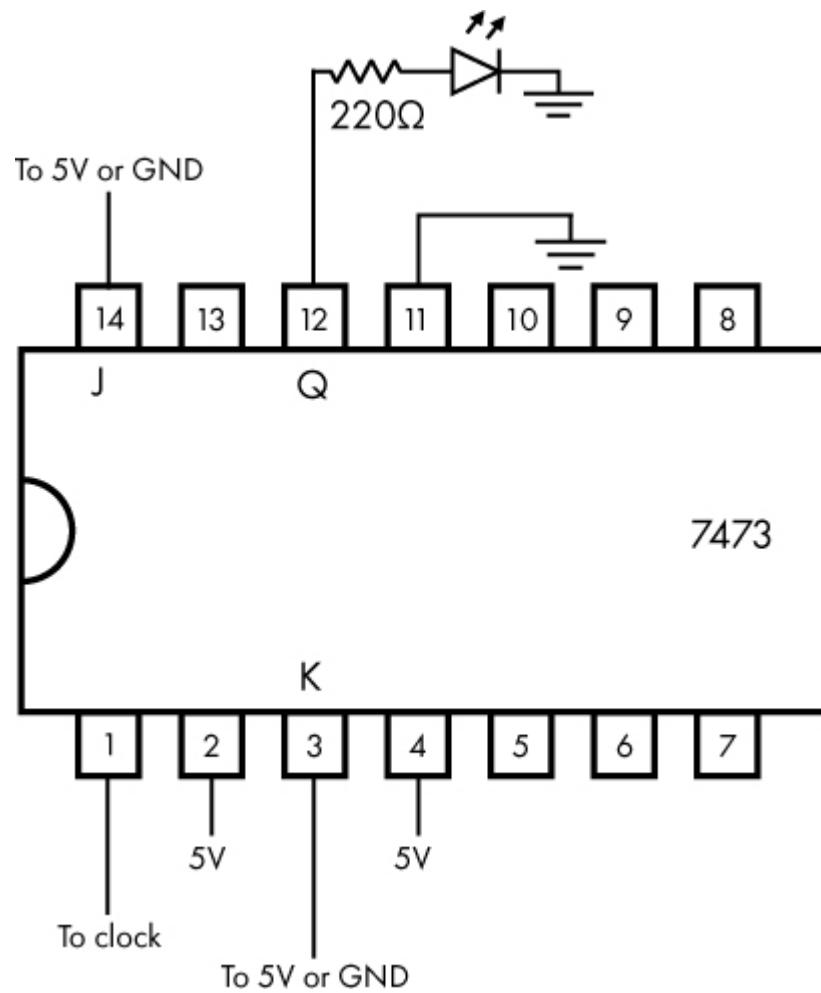


Figure 6-28: A simple JK test circuit

Using your previously built SR latch as a clock, connect the SR latch's output Q (pins 3 and 4 on the 7402) to the clock input of the 7473 (pin 1). Now, try setting inputs J (pin 14) and K (pin 3) on the 7473 to 5V or ground. You should see that doing so has no effect on the JK flip-flop's output LED until the clock transitions from high to low. Reminder: pulse the SR latch clock by first pressing S and then pressing R to set the clock signal high and then low. Flip back to Table 6-3 to see the expected functionality of a JK flip-flop and ensure that your circuit works as expected.

Keep this circuit intact for the next project.

PROJECT #11: CONSTRUCT A 3-BIT COUNTER

In this project, you'll build the 3-bit counter described earlier in this chapter. Connect the Q outputs to LEDs to easily observe the output.

For this project, you'll need the following components:

- The circuit constructed in Project #10 (including the manual clock from Project #9)
- An additional 7473 IC
- 7408 IC (contains four AND gates)
- 47k Ω resistor
- 10 μ F electrolytic capacitor
- An additional button or switch
- Jumper wires
- Two additional LEDs
- Two additional current-limiting resistors to use with your LEDs (approximately 220 Ω each)

Connect everything as shown in Figure 6-29. Pin numbers on ICs are shown in boxes.

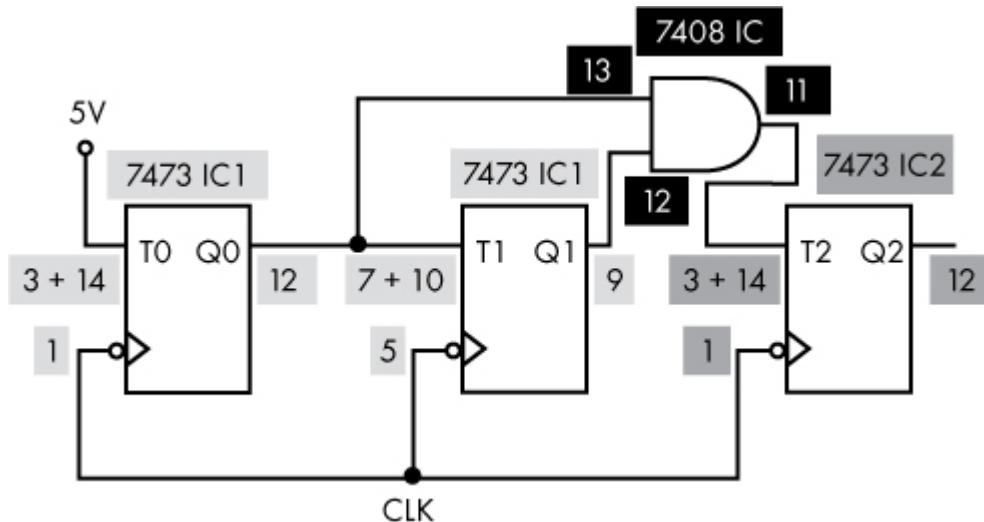


Figure 6-29: A 3-bit counter built from T flip-flops, pin numbers shown

In addition to the pin connections shown in Figure 6-29, be sure to make the following connections:

- Both 7473 ICs need pins 4 and 11 connected to 5V and ground, respectively.
- The 7408 should have pin 7 connected to ground and pin 14 to 5V.
- Q0, Q1, and Q2 should connect to LEDs through 220 Ω resistors so you can see the bits update.

- The manual clock output (pins 3 and 4 on the 7402) should be connected to CLK on all three flip-flops (pins 1 and 5 on the first 7473, and pin 1 on the second 7473).

This circuit starts up in an unpredictable state. You can correct this manually by resetting all three flip-flops, but that's tedious. Instead, add a *power-on reset* circuit that ensures the flip-flops all start with their output = 0. Each flip-flop in the 7473 package has a CLR input, which when held low, resets the flip-flop, regardless of the state of the clock. You want CLR to go low on startup for a brief time and then go high and stay there. This ensures that the counter starts at zero when powered on. For good measure, you can also add a COUNTER RESET button that manually resets the counter when pressed. This reset capability is shown in Figure 6-30.

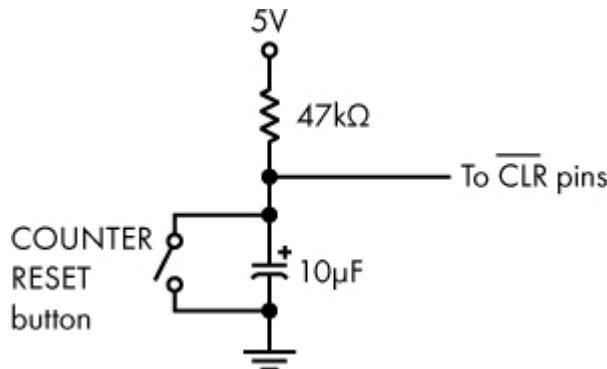


Figure 6-30: Power-on reset circuit for 3-bit binary counter

When power is initially applied to the circuit shown in Figure 6-30, the capacitor acts like a short circuit, and CLR is held low, setting the circuit to its initial state. Once the capacitor is charged, it acts like an open circuit, CLR goes high, and the circuit is ready to use. The COUNTER RESET button or switch, when pushed, also causes CLR to go low and reset the circuit. This circuit needs to be connected to CLR inputs: pins 2 and 6 on the first 7473 chip, and pin 2 and the second 7473 chip. In Project #10, pin 2 on the first 7473 was connected to 5V; be sure to disconnect it before you hook up the power-on reset circuit. Remember to orient the terminals of your electrolytic capacitor correctly—the negative terminal should connect to ground.

With the power-on reset in place, the circuit should start with your counter at 000. Sending a clock pulse to the circuit should cause the counter to increment by 1 when the clock edge falls. Reminder: pulse the SR latch clock by pressing S to set the clock signal high and pressing R to set the clock signal low. Test counting from 000 to 111 and ensure the counter works as expected.

7

COMPUTER HARDWARE



The preceding chapters covered the foundational elements of computing—binary, digital circuits, memory. Let’s now examine how these elements come together in a computer, a device that’s more than the sum of its parts. In this chapter, I first provide an overview of computer hardware. Then we dive deeper into three parts of a computer: main memory, the processor, and input/output.

Computer Hardware Overview

Let’s begin with an overview of what makes a computer different from other electronic devices. Previously, we’ve seen how we can use logic circuits and memory devices to build circuits that perform useful tasks. The circuits we’ve built with logic gates have a set of features hard-wired into their design. If we want to add or modify a feature, we have to change the physical design of our circuit. On a breadboard that’s possible, but for a device that has been manufactured and sent to customers, changing hardware isn’t usually an option. Defining the features of a device in hardware alone limits our ability to quickly innovate and improve a design.

The circuits we've built so far give us a glimpse into how computers work, but we're missing one critical element of computers: *programmability*. That is, a computer must be able to perform new tasks *without* changing hardware. To accomplish such a feat, a computer must be able to accept a set of instructions (a *program*) and perform the actions specified in those instructions. It must therefore have hardware that can perform a variety of operations in the order specified in a program. Programmability is a key differentiator between a device that is a computer and one that is not. In this chapter we cover computer *hardware*, the physical elements of a computer. This is in contrast to *software*, the instructions that tell a computer what to do, which we'll cover in the next chapter.

The ability to run software distinguishes a computer from a fixed-purpose device. That said, software still needs hardware, so what kind of hardware do we need to implement a general-purpose computer? First, we need memory. We've already covered single-bit memory devices such as latches and flip-flops; the type of memory used in a computer is a conceptual extension of those simple memory devices. The primary memory used in a computer is known as *main memory*, but often it's referred to as just memory or *random access memory (RAM)*. It's *volatile*, meaning it only retains data while powered. The "random access" part of RAM means that any arbitrary memory location can be accessed in roughly the same amount of time as any other location.

The second key component we need is a *central processing unit*, or *CPU*. Often simply called a *processor*, this component carries out the instructions specified in software. The CPU can directly access main memory. Most processors today are *microprocessors*, CPUs on a single integrated circuit. A processor built on a single integrated circuit has the benefits of lower cost, improved reliability, and increased performance. A CPU is a conceptual extension of the digital logic circuits we covered previously.

Although main memory and a CPU are the minimum hardware requirements for a computer, in practice most computing devices need

to interact with the outside world, and they do so through input/output (I/O) devices. In this chapter, we cover main memory, the CPU, and I/O in more detail. These three elements are illustrated in Figure 7-1.

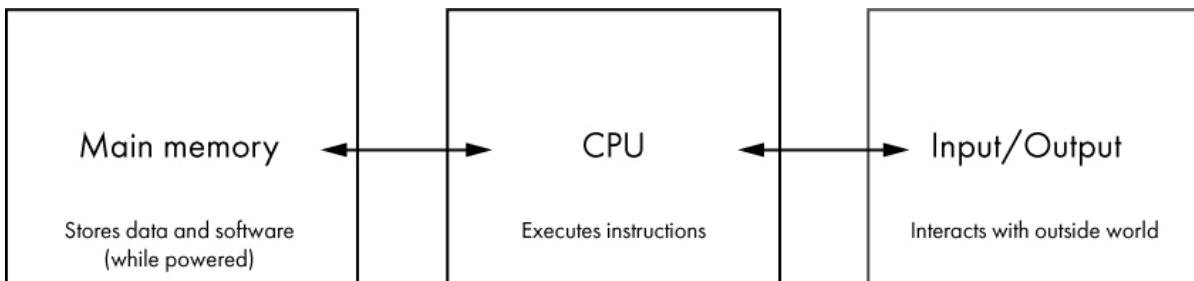


Figure 7-1: The hardware elements of a computer

Main Memory

While executing a program, a computer needs a place to store the program's instructions and related data. For example, when a computer runs a word processor for editing documents, the computer needs a place to hold the program itself, the contents of the document, and the state of editing—what part of the document is visible, the location of the cursor, and so forth. All of this data is ultimately a series of bits that the CPU needs to be able to access. Main memory handles the task of storing these 1s and 0s.

Let's explore how main memory works in a computer. There are two common types of computer memory: *static random access memory (SRAM)* and *dynamic random access memory (DRAM)*. In both types, the basic unit of memory storage is a *memory cell*, a circuit that can store a single bit. In SRAM, memory cells are a type of flip-flop. SRAM is static because its flip-flop memory cells retain their bit values while power is applied. On the other hand, DRAM memory cells are implemented using a transistor and a capacitor. The capacitor's charge leaks over time, so data must be periodically rewritten to the cells. This refreshing of the memory cells is what makes DRAM dynamic. Today, DRAM is commonly used for main memory due to its relatively low price. SRAM is faster but more expensive, so it's used in scenarios where speed is

critical, such as in cache memory, which we'll cover later. An example "stick" of RAM is shown in Figure 7-2.

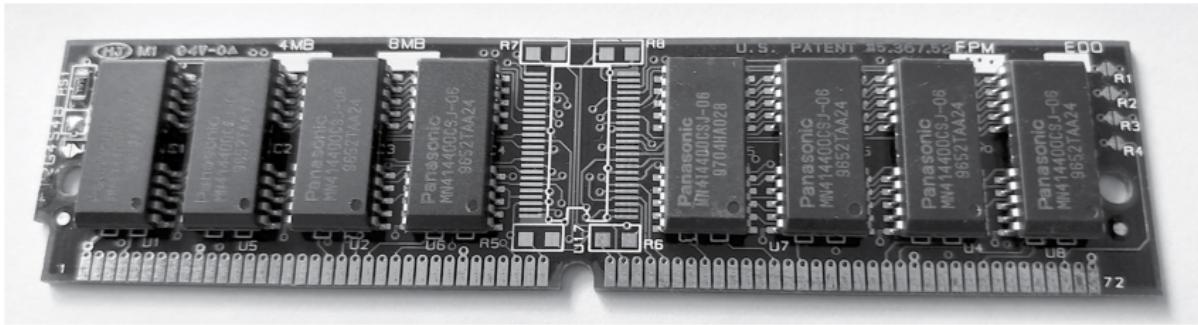


Figure 7-2: Random access memory

As a generalization, you can think of the internals of RAM as grids of memory cells. Each single-bit cell in a grid can be identified using two-dimensional coordinates, the location of that cell in its grid. Accessing a single bit at a time isn't very efficient, so RAM accesses multiple grids of 1-bit memory cells in parallel, allowing for reads or writes of multiple bits at once—a whole byte, for example. The location of a set of bits in memory is known as a *memory address*, a numeric value that identifies a memory location. It's common for memory to be *byte-addressable*, meaning a single memory address refers to 8 bits of data. The internal details of the arrangement of memory or the implementation of the memory cells aren't required knowledge for a CPU (or a programmer!). The main thing to understand is that computers assign numeric addresses to bytes of memory, and the CPU can read or write to those addresses, as illustrated in Figure 7-3.

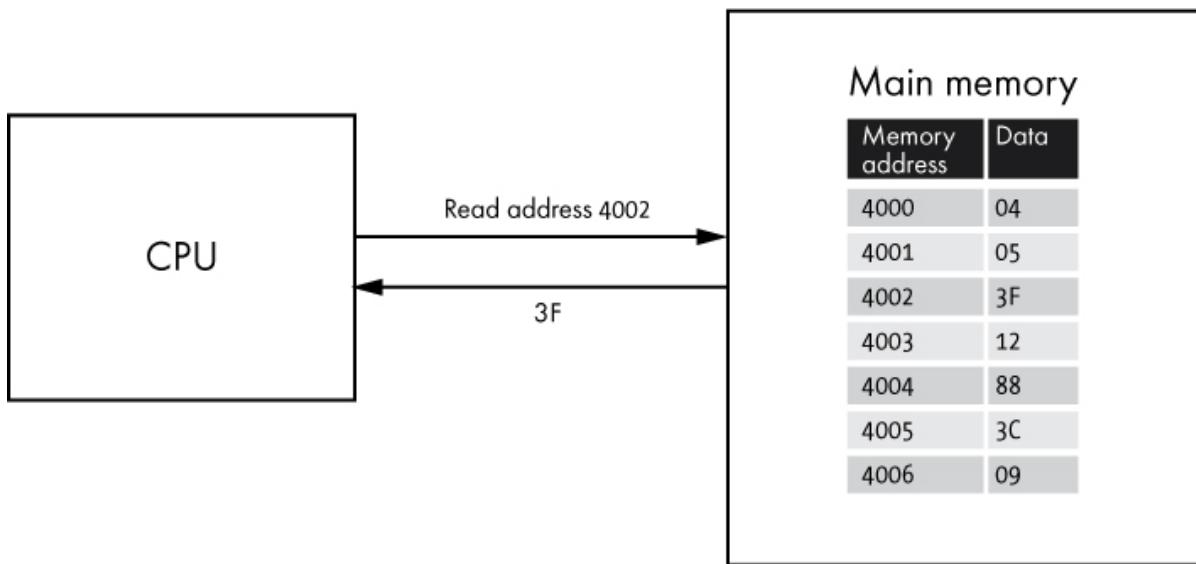


Figure 7-3: A CPU reads a byte from a memory address.

Let's consider a fictitious computer system that can address up to 64KB of memory. By today's standards, that's a tiny amount of memory for a computer, but it's still useful for us as an example. Let's also imagine that our fictitious computer's memory is byte-addressable; each memory address represents a single byte. That means that we need one unique address for each byte of memory, and since 64KB is $64 \times 1024 = 65,536$ bytes, we need 65,536 unique addresses. Each address is just a number, and memory addresses usually start with 0, so our range of addresses is 0 to 65,535 (or 0xFFFF).

Since our fictitious 64KB computer is a digital device, memory addresses are ultimately represented in binary. How many bits do we need to represent a memory address on this system? The number of unique values that can be represented by a binary number with n bits is equal to 2^n . So we want to know the value of n for $2^n = 65,536$. The inverse of raising 2 to a certain power is the base-2 logarithm. Therefore $\log_2(2^n) = n$ and $\log_2(65,536) = 16$. Stated another way, $2^{16} = 65,536$. Therefore, a 16-bit memory address is needed to address 65,536 bytes.

Or, more simply, since we already know that our highest numbered memory address is 0xFFFF, and we know that each hexadecimal symbol

represents 4 bits, we can see that 16 bits are required (4 hex symbols \times 4 bits per symbol). Again, our fictitious computer is able to address 65,536 bytes, and each byte is assigned a 16-bit memory address. Table 7-1 shows a 16-bit memory layout with some example data.

Table 7-1: A 16-Bit Memory Address Layout, Skipping Middle Addresses, with Example Data

Memory address (as binary)	Memory address (as hex)	Example data
0000000000000000	0000	23
0000000000000001	0001	51
0000000000000010	0002	4A
-----	----	--
1111111111111101	FFFD	03
1111111111111110	FFFE	94
1111111111111111	FFFF	82

Why does the number of bits matter? The number of bits used to represent a memory address is a key part of a computer system’s design. It limits the amount of memory that a computer can access, and it impacts how programs deal with memory at a low level.

Let’s now imagine that our fictitious computer has stored the ASCII string “Hello” starting at memory address 0x0002. Since ASCII characters each require 1 byte, storing “Hello” requires 5 bytes. When examining memory, it’s common to use hexadecimal to represent both memory addresses and the contents of those memory addresses. Table 7-2 provides a visual look at “Hello” stored in memory, starting at address 0x0002.

Table 7-2: “Hello” Stored in Memory

Memory address	Data byte	Data as ASCII
0000	00	
0001	00	
0002	48	H
0003	65	e

Memory address	Data byte	Data as ASCII
0004	6C	l
0005	6C	l
0006	6F	o
0007	00	
----	--	
FFFF	00	

Using this format makes it clear that each address only stores 1 byte, so storing all 5 ASCII characters requires addresses 0x0002 through 0x0006. Note that the table shows a value of 00 for other memory addresses, but in practice, it isn't safe to assume that a random address will hold 0; it could be anything. That said, in some programming languages it's standard practice to end a text string with a null terminator (a byte equal to 0), and in that case, we'd actually expect to see 00 at address 0x0007.

Applications that allow inspection of computer memory commonly represent the contents of memory in a format similar to what is shown in Figure 7-4.

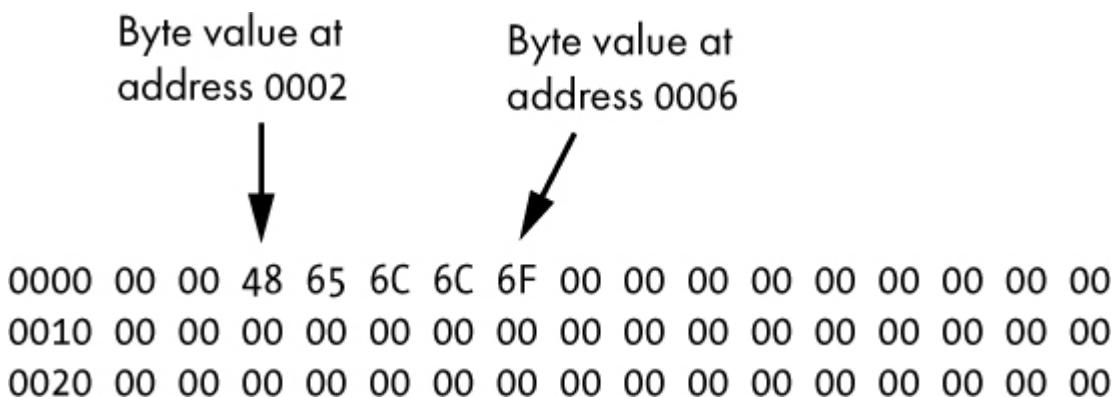


Figure 7-4: A typical view of memory bytes

The leftmost column in Figure 7-4 is a memory address in hexadecimal, and the following 16 values represent the bytes at that address and the 15 subsequent addresses. This approach is more compact than Table 7-2, but it means that each address isn't uniquely

called out. In this figure, we again see the ASCII string “Hello” stored starting at address 0x0002.

Our hypothetical computer with 64KB of RAM is useful as an example, but modern computing devices tend to have a much larger amount of memory. As of 2020, smartphones commonly have at least 1GB of memory, and laptop computers usually have at least 4GB.

EXERCISE 7-1: CALCULATE THE REQUIRED NUMBER OF BITS

Using the techniques just described, determine the number of bits required for addressing 4GB of memory. You’ll want to look back at Table 1-3 for a reference on SI prefixes. Remember that each byte is assigned a unique address, which is just a number. The answer is in Appendix A.

Central Processing Unit (CPU)

Memory gives the computer a place to store data and program instructions, but it’s the CPU, or processor, that carries out those instructions. It’s the processor that allows a computer to have the flexibility to run programs that weren’t even conceived of at the time the processor was designed. A processor implements a set of instructions that programmers can then use to construct meaningful software. Each instruction is simple, but these basic instructions are the building blocks for all software.

Here are some examples of types of instructions that CPUs support:

Memory access read, write (to memory)

Arithmetic add, subtract, multiply, divide, increment

Logic AND, OR, NOT

Program flow jump (to a specific part of a program), call (a subroutine)

We'll go into specific CPU instructions in Chapter 8, but for now, it's important to understand that CPU instructions are just operations that the processor can perform. They are fairly simple (add two numbers, read from a memory address, perform a logical AND, and so forth). Programs consist of ordered sets of these instructions. To use a cooking analogy, the CPU is the cook, a program is a recipe, and each instruction in the program is a step of the recipe that the cook knows how to perform.

Program instructions reside in memory. The CPU reads these instructions so it can run the program. Figure 7-5 illustrates a simple program that's read from memory by the CPU.

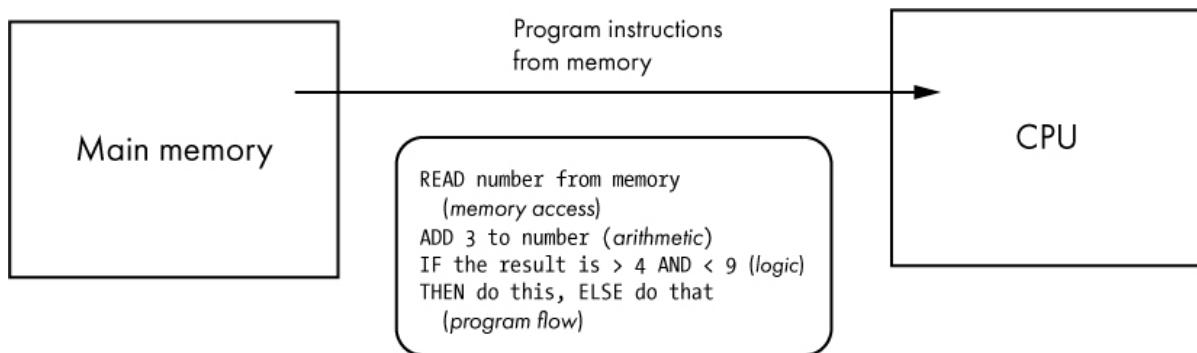


Figure 7-5: An example program is read from memory and runs on the CPU.

The example program in Figure 7-5 is written in *pseudocode*, a human-readable description of a program that's not written in a real programming language. The steps in the program fall into the categories just described (memory access, arithmetic, logic, and program flow). In the first step, the program reads a number stored at a certain address in memory. The program then adds 3 to that number. It then performs a logical AND of two conditions. If the logical result is true, then the program does “this”; otherwise, it does “that.” Believe it or not, all programs are, in essence, simply various combinations of these types of fundamental operations.

Instruction Set Architectures

Although all CPUs implement these types of instructions, the specific instructions available on different processors vary. Some instructions that exist for one type of CPU simply don't exist on other types of CPUs. Even instructions that do exist on nearly all CPUs aren't implemented in the same way. For example, the specific binary sequence used to mean "add two numbers" is not the same across processor types. A family of CPUs that use the same instructions are said to share an *instruction set architecture (ISA)*, or just *architecture*, a model of how a CPU works. Software that's built for a certain ISA works on any CPU that implements that ISA. It's possible for multiple processor models, even those from different manufacturers, to implement the same architecture. Such processors may work very differently internally, but by adhering to the same ISA, they can run the same software. Today, there are two prevalent instruction set architectures: x86 and ARM.

The majority of desktop computers, laptops, and servers use x86 CPUs. The name comes from Intel Corporation's naming convention for its processors (each ending in 86), beginning with the 8086 released in 1978, and continuing with the 80186, 80286, 80386, and 80486. After the 80486 (or more simply the 486), Intel began branding its CPUs with names such as Pentium and Celeron; these processors are still x86 CPUs despite the name change. Other companies besides Intel also produce x86 processors, notably Advanced Micro Devices, Inc. (AMD).

The term *x86* refers to a set of related architectures. Over time, new instructions have been added to the x86 architecture, but each generation has tried to retain backward compatibility. This generally means that software developed for an older x86 CPU runs on a newer x86 CPU, but software built for a newer x86 CPU that takes advantage of new x86 instructions won't be able to run on older x86 CPUs that don't understand the new instructions.

The x86 architecture includes three major generations of processors: 16-bit, 32-bit, and 64-bit. Let's pause to examine what we mean when we say that a CPU is a 16-bit, 32-bit, or 64-bit processor. The number of bits associated with a processor, also known as its *bitness* or *word size*,

refers to the number of bits it can deal with at a time. So a 32-bit CPU can operate on values that are 32 bits in length. More specifically, this means that the computer architecture has 32-bit registers, a 32-bit address bus, or a 32-bit data bus. Or all three may be 32-bit. We'll cover more details on registers, data buses, and address buses later.

Going back to x86 and its generations of processors, the original 8086 processor, released in 1978, was a 16-bit processor. Encouraged by the success of the 8086, Intel continued producing compatible processors. Intel's subsequent x86 processors were also 16-bit until the 80386 processor was released in 1985, bringing with it a new 32-bit version of the x86 architecture. This 32-bit version of x86 is sometimes called IA-32. Thanks to backward compatibility, modern x86 processors still fully support IA-32. An example x86 processor is shown in Figure 7-6.

Interestingly, it was AMD, and not Intel, that brought x86 into the 64-bit era. In the late 1990s, Intel's 64-bit focus was on a new CPU architecture called IA-64 or Itanium, which was *not* an x86 ISA, and ended up as a niche product for servers. With Intel focused on Itanium, AMD seized the opportunity to extend the x86 architecture. In 2003, AMD released the Opteron processor, the first 64-bit x86 CPU. AMD's architecture was originally known as *AMD64*, and later Intel adopted this architecture and called its implementation *Intel 64*. The two implementations are mostly functionally identical, and today 64-bit x86 is generally referred to as *x64* or *x86-64*.

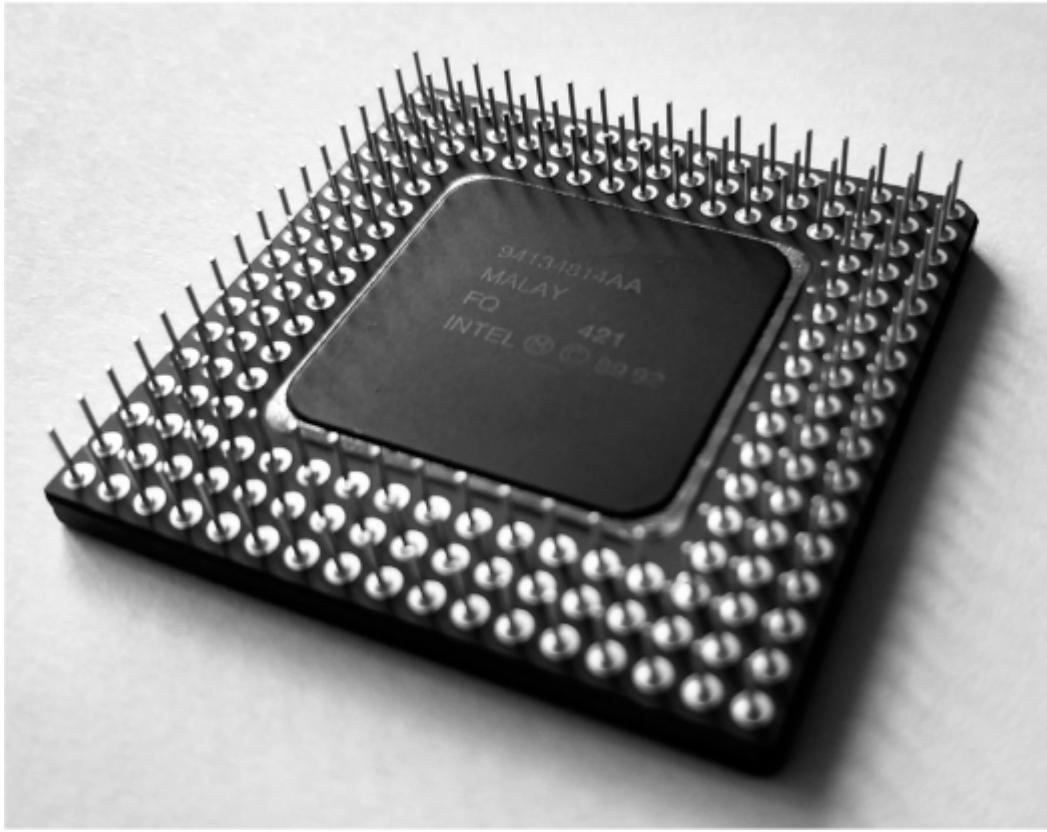


Figure 7-6: An Intel 486 SX, a 32-bit x86 processor

Although x86 rules the personal computer and server world, ARM processors command the realm of mobile devices like smartphones and tablets. Multiple companies manufacture ARM processors. A company called ARM Holdings develops the ARM architecture and licenses their designs to other companies to implement. It's common for ARM CPUs to be used in *system-on-chip (SoC)* designs, where a single integrated circuit contains not only a CPU, but also memory and other hardware. The ARM architecture originated in the 1980s as a 32-bit ISA. A 64-bit version of the ARM architecture was introduced in 2011. ARM processors are favored in mobile devices due to their reduced power consumption and lower cost as compared to x86 processors. ARM processors can be used in PCs as well, but that market largely remains focused on x86, to retain backward compatibility with existing x86 PC software. However, in 2020, Apple announced their intention to move macOS computers from x86 to ARM CPUs.

CPU Internals

Internally, a CPU consists of multiple components that work together to execute instructions. We'll focus on three fundamental components: the processor registers, the arithmetic logic unit, and the control unit. *Processor registers* are locations within the CPU that hold data during processing. The *arithmetic logic unit (ALU)* performs logical and mathematical operations. The processor *control unit* directs the CPU, communicating with the processor registers, the ALU, and main memory. Figure 7-7 shows a simplified view of CPU architecture.

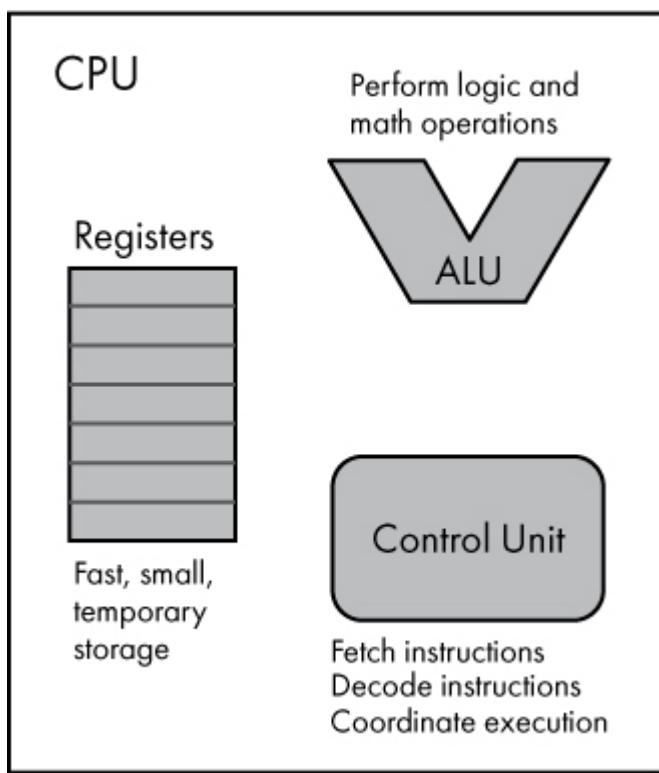


Figure 7-7: A greatly simplified view of CPU architecture

Let's look at processor registers. Main memory holds data for an executing program. However, when a program needs to operate on a piece of data, the CPU needs a temporary place to store the data within the processor hardware. To accomplish this, CPUs have small internal storage locations known as processor registers, or just registers.

Compared to accessing main memory, accessing registers is a very fast operation for a CPU, but registers can only hold very small amounts of

data. We measure the size of an individual register in bits, not bytes, because registers are so small. As an example, a 32-bit CPU usually has registers that are 32 bits “wide,” meaning each register can hold 32 bits of data. The registers are implemented in a component known as the *register file* (not to be confused with a data file, such as a document or photo). The memory cells used in the register file are typically a type of SRAM.

The ALU handles logic and math operations within the CPU. We previously covered combinational logic circuits, circuits in which the output is a function of the input. A processor’s ALU is just a complex combinational logic circuit. The ALU’s inputs are values called *operands*, and a code indicating what operation to perform on those operands. The ALU outputs the result of the operation along with a status that provides more detail on execution of the operation.

The control unit acts as the coordinator of the CPU. It works on a repeating cycle: fetch an instruction from memory, decode it, and execute it. Since a running program is stored in memory, the control unit needs to know which memory address to read in order to fetch the next instruction. The control unit determines this by looking at a register known as the *program counter (PC)*, also known as the *instruction pointer* on x86. The program counter holds the memory address of the next instruction to execute. The control unit reads the instruction from the specified memory address, stores the instruction in a register called the *instruction register*, and updates the program counter to point to the next instruction. The control unit then decodes the current instruction, making sense of the 1s and 0s that represent an instruction. Once decoded, the control unit executes the instruction, which may require coordinating with other components in the CPU. For example, execution of an addition operation requires the control unit to instruct the ALU to perform the needed math. Once an instruction has completed, the control unit repeats the cycle: fetch, decode, execute.

Clock, Cores, and Cache

Since CPUs execute ordered sets of instructions, you may wonder what causes a CPU to progress from one instruction to the next. We previously demonstrated how a clock signal can be used to move a circuit from one state to another, such as in a counter circuit. The same principle applies here. A CPU takes an input clock signal, as illustrated in Figure 7-8, and a clock pulse acts as a signal to the CPU to transition between states.

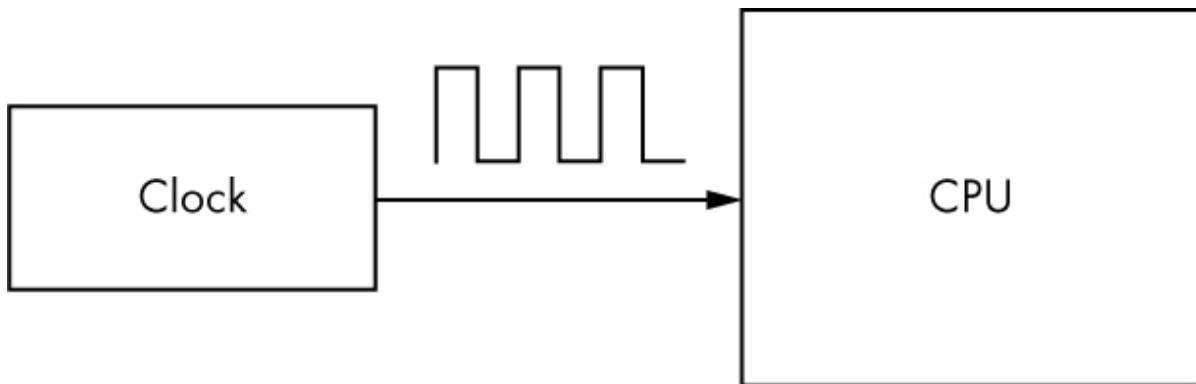


Figure 7-8: A clock provides an oscillating signal to the CPU.

It's an oversimplification to think that a CPU executes exactly one instruction per clock cycle. Some instructions take multiple cycles to complete. Also, modern CPUs use an approach called *pipelining* to divide instructions into smaller steps so that portions of multiple instructions can be run in parallel by a single processor. For example, one instruction can be fetched while another is decoded and yet another is executed. Still, it can be helpful to think of each pulse of the clock as a signal to the CPU to move forward with executing a program. Modern CPUs have clock speeds measured in *gigahertz (GHz)*. For example, a 2GHz CPU has a clock that oscillates at 2 *billion* cycles per second!

Increasing the frequency of the clock allows a CPU to perform more instructions per second. Unfortunately, we can't just run a CPU at an arbitrarily high clock rate. CPUs have a practical upper limit on their input clock frequency, and pushing a CPU beyond that limit leads to excessive heat generation. Also, the CPU's logic gates may not be able to keep up, causing unexpected errors and crashes. For many years, the

computer industry saw steady increases in the upper limit of clock rates for CPUs. This clock rate increase was largely due to regular improvements in manufacturing processes that led to increased transistor density, which allowed for CPUs with higher clock rates but roughly the same power consumption. In 1978, the Intel 8086 ran at 5MHz, and by 1999, the Intel Pentium III had a 500MHz clock, a 100x increase in only about 20 years! CPU clock rates continued to increase rapidly until the 3GHz threshold was crossed in the early 2000s. Since then, despite continued growth in transistor count, physical limitations associated with diminutive transistor sizes have made significant increases to clock rate impractical.

With clock rates stagnant, the processor industry turned to a new approach for getting more work out of a CPU. Rather than focusing on increasing clock frequency, CPU design began to focus on execution of multiple instructions in parallel. The idea of a *multicore* CPU was introduced, a CPU with multiple processing units called *cores*. A *CPU core* is effectively an independent processor that resides alongside other independent processors in a single CPU package, as illustrated in Figure 7-9.

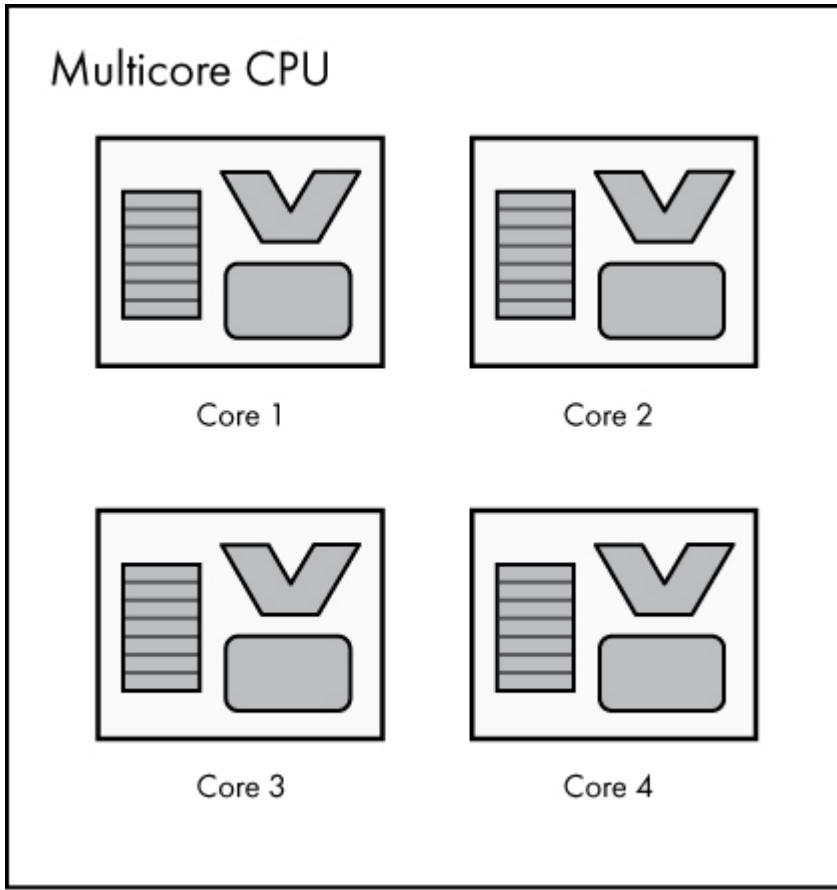


Figure 7-9: A four-core CPU—each core has its own registers, ALU, and control unit

Note that multiple cores running in parallel is not the same as pipelining. The parallelism of multicore means that each core works on a different task, a separate set of instructions. In contrast, pipelining happens *within* each core, allowing portions of multiple instructions to be executed in parallel by a single core.

Each core added to a processor opens the door to a computer running additional instructions in parallel. That said, adding multiple cores to a computer's CPU doesn't mean all applications benefit immediately or equally. Software must be written to take advantage of parallel processing of instructions to get the maximum benefit of multicore hardware. However, even if individual programs aren't designed with parallelism in mind, a computer system as a whole can benefit, since modern operating systems run multiple programs at once.

I've previously described how CPUs load data from main memory into registers for processing and then store that data back from registers to memory for later use. It turns out that programs tend to access the same memory locations over and over. As you might expect, going back to main memory multiple times to access the same data is inefficient! To avoid this inefficiency, a small amount of memory resides within the CPU that holds a copy of data frequently accessed from main memory. This memory is known as a *CPU cache*.

The processor checks the cache to see if data it wishes to access is there. If so, the processor can speed things up by reading or writing to the cache rather than to main memory. When needed data is not in the cache, the processor can move that data into cache once it has been read from main memory. It's common for processors to have multiple cache levels, often three. We refer to these cache levels as L1 cache, L2 cache, and L3 cache. A CPU first checks L1 for the needed data, then L2, then L3, before finally going to main memory, as illustrated in Figure 7-10. L1 cache is the fastest to access, but it's also the smallest. L2 is slower and larger, and L3 is slower and larger still. Remember that even with these progressively slower levels of cache, it is still slower to access main memory than any level of cache.

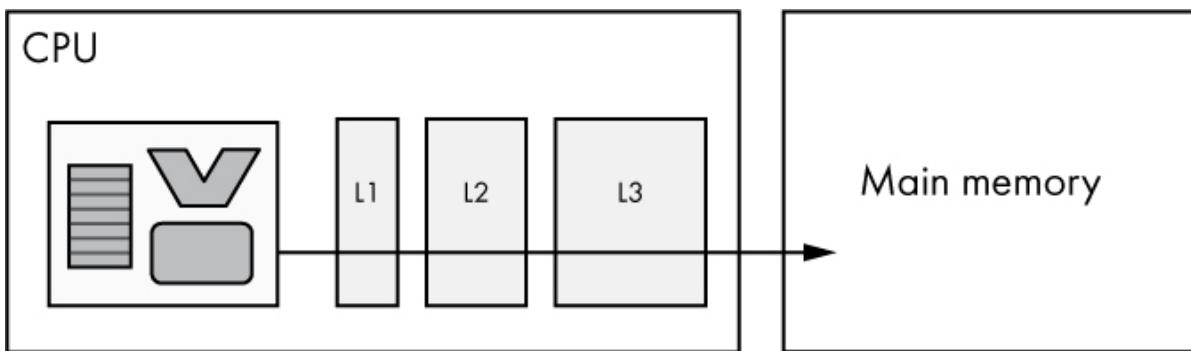


Figure 7-10: A single-core CPU with three levels of cache

In multicore CPUs, some caches are specific to each core, whereas others are shared among the cores. For example, each core may have its own L1 cache, whereas the L2 and L3 caches are shared, as shown in Figure 7-11.

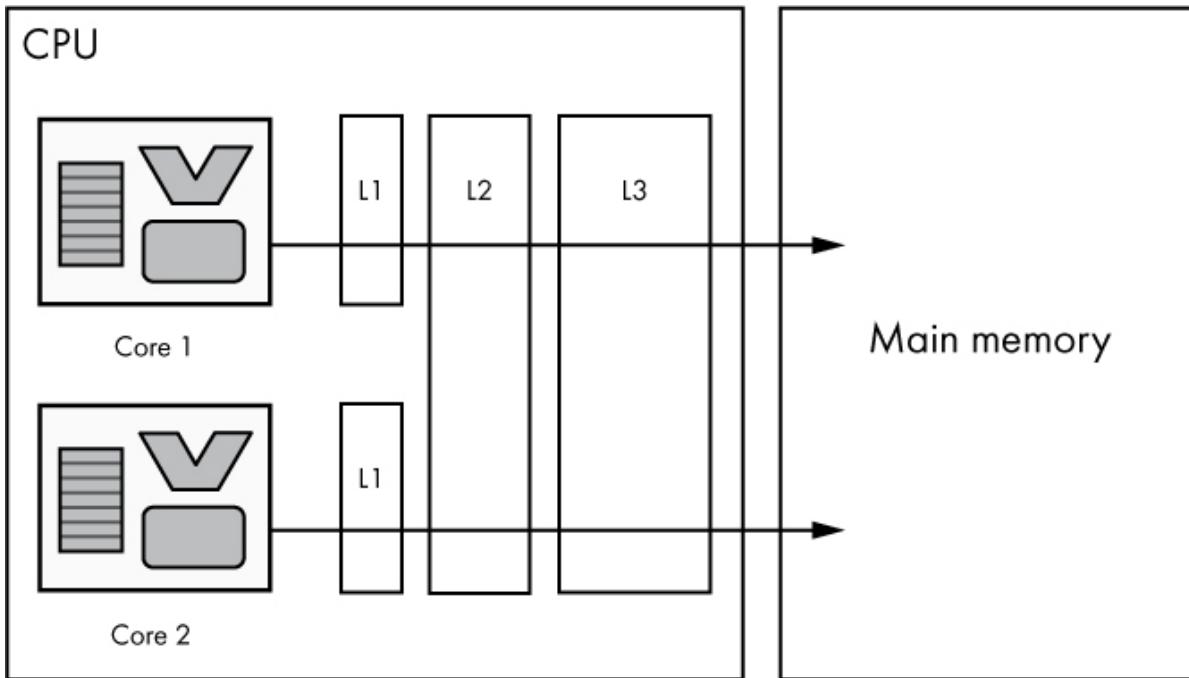


Figure 7-11: A two-core CPU with cache. Each core has its own L1 cache, whereas L2 and L3 caches are shared.

Beyond Memory and Processor

I have outlined the two fundamental components required for a computer: memory and a processor. However, a device that consists of only memory and a processor has a couple of gaps that need to be filled if we want a useful device. The first gap is that both memory and CPUs are volatile; they lose state when power is removed. The second gap is that a computer with only memory and a processor has no way of interacting with the outside world. Let's now see how secondary storage and I/O devices fill these gaps.

Secondary Storage

If a computer only included memory and a processor, then every time that device was powered down, it would lose all its data! To emphasize that point, *data* here means not only a user's files and settings, but also any installed applications, and even the operating system itself. This rather inconvenient computer would require someone to load the OS

and any applications every time it was powered on. That might discourage users from ever turning it off. Believe it or not, computers in previous generations did work this way, but fortunately that isn't the case today.

To address this problem, computers have secondary storage. *Secondary storage* is nonvolatile and therefore remembers data even when the system is powered down. Unlike RAM, secondary storage is not directly addressable by the CPU. Such storage is usually much cheaper per byte than RAM, allowing for a large capacity of storage as compared to main memory. However, secondary storage is also considerably slower than RAM; it isn't a suitable replacement for main memory.

In modern computing devices, hard disk drives and solid-state drives are the most common secondary storage devices. A *hard disk drive (HDD)* stores data using magnetism on a rapidly spinning platter, whereas a *solid-state drive (SSD)* stores data using electrical charges in nonvolatile memory cells. Compared to HDDs, SSDs are faster, quieter, and more resistant to mechanical failure, since SSDs have no moving parts. Figure 7-12 is a photo of a couple of secondary storage devices.

With a secondary storage device in place, a computer can load data on demand. When a computer is powered on, the operating system loads from secondary storage into main memory; any applications that are set to run at startup also load. After startup, when an application is launched, program code loads from secondary storage into main memory. The same goes for any user data (documents, music, settings, and so on) stored locally; it must load from secondary storage into main memory before it can be used. In common usage, secondary storage is often referred to simply as storage, while primary storage/main memory is just called memory or RAM.



Figure 7-12: A 4GB hard disk drive from 1997 beside a modern 32GB microSD card, a type of solid-state storage

Input/Output

Even with secondary storage in place, our hypothetical computer still has a problem. A computer consisting of a processor, memory, and storage doesn't have any way of interacting with the outside world! This is where input/output devices come in. An *input/output (I/O) device* is a component that allows a computer to receive input from the outside world (keyboard, mouse), send data to the outside world (monitor, printer), or both (touchscreen). Human interaction with a computer requires going through I/O. Computer-to-computer interaction also requires going through I/O, often in the form of a computer network, such as the internet. Secondary storage devices are actually a type of I/O device. You may not think of accessing internal storage as I/O, but from the perspective of the CPU, reading or writing to storage is just another I/O operation. Reading from the storage device is input, while writing

to the storage device is output. Figure 7-13 provides some examples of input and output.

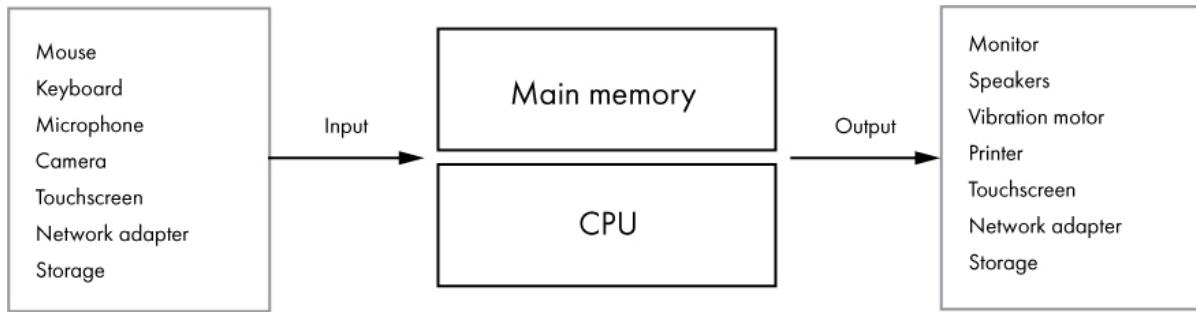


Figure 7-13: Common types of input and output

So how does a CPU go about communicating with I/O devices? A computer can have a wide variety of I/O devices attached to it, and the CPU needs a standard way to communicate with any such device. To understand this, we need to first discuss *physical address space*, the range of hardware memory addresses available to a computer. Earlier in this chapter, in the section entitled “Main Memory” on page 119, we covered how bytes of memory are assigned an address. All memory addresses on a given computer system will be represented with a certain number of bits. That number of bits determines not only the size of each memory address, but also the range of addresses available for the computer hardware to use—the physical address space. Address space is often larger than the amount of RAM installed on a computer, leaving some physical memory addresses unused.

To give an example, in the case of a computer with a 32-bit physical address space, the physical address range is from 0x00000000 to 0xFFFFFFFF (the largest address that can be represented with a 32-bit number). That’s approximately 4 billion addresses, each representing a single byte, or 4GB of address space. Let’s say that this computer has 3GB of RAM, so 75 percent of the available physical memory addresses are assigned to bytes of RAM.

Now let’s return to the question of how CPUs communicate with I/O devices. Addresses in physical address space don’t always refer to bytes of memory; they can also refer to an I/O device. When physical

address space is mapped to an I/O device, the CPU can communicate with that device just by reading or writing to its assigned memory address(es); this is called *memory-mapped I/O (MMIO)* and is illustrated in Figure 7-14. When a computer treats the memory of I/O devices just like main memory, its CPU does not need any special instructions for I/O operations.

However, some CPU families, notably x86, do include special instructions for accessing I/O devices. When computers use this approach, rather than mapping I/O devices to a physical memory address, devices are assigned an *I/O port*. A port is like a memory address, but instead of referring to a location in memory, the port number refers to an I/O device. You can think of the set of I/O ports as just another address space, distinct from memory addresses. This means that port 0x378 does not refer to the same thing as physical memory address 0x378. Accessing I/O devices through a separate port address space is known as *port-mapped I/O (PMIO)*. Today's x86 CPUs support both port-mapped and memory-mapped I/O.

I/O ports and memory-mapped I/O addresses generally refer to a device controller rather than directly to data stored on the device. For example, in the case of a hard disk drive, the bytes of the disk aren't directly mapped into address space. Instead, a hard drive controller presents an interface, accessible through I/O ports or memory-mapped I/O addresses, that allows the CPU to request read or write operations to locations on the disk.

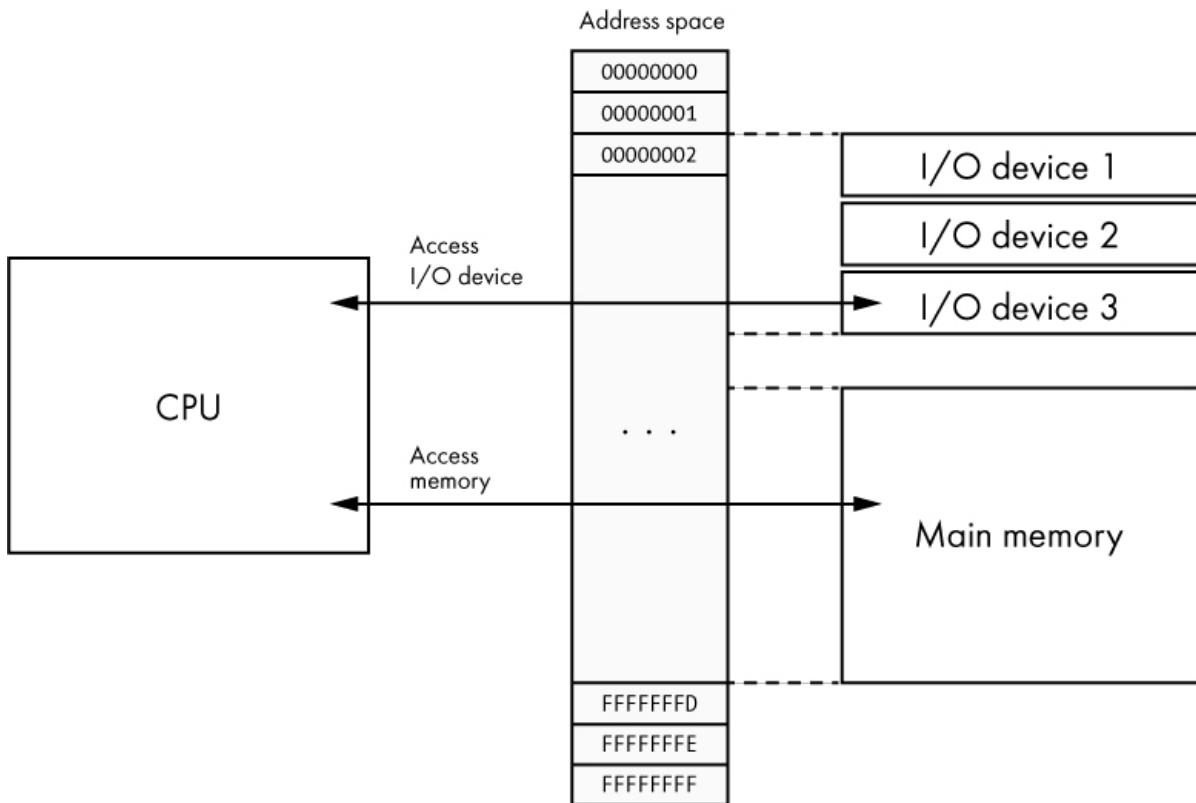


Figure 7-14: Memory-mapped I/O

EXERCISE 7-2: GET TO KNOW THE HARDWARE DEVICES IN YOUR LIFE

Choose a couple of computing devices that you own or use—say a laptop, smartphone, or game console. Answer the following questions about each device. You may be able to find the answers by looking at the settings on the device itself, or you may have to do some research online.

- What kind of CPU does the device have?
- Is the CPU 32-bit or 64-bit (or something else)?
- What's the CPU clock frequency?
- Does the CPU have L1, L2, or L3 cache? If so, how much?
- Which instruction set architecture does the CPU use?
- How many cores does the CPU have?

- How much and what kind of main memory does the device have?
- How much and what kind of secondary storage does the device have?
- What I/O devices does the device have?

Bus Communication

At this point, we've covered the roles of memory, the CPU, and I/O devices in a computer. We've also touched on the CPU's communication with both memory and I/O devices through memory address space. Let's take a closer look at how the CPU communicates with memory and I/O devices.

A *bus* is a hardware communication system used by computer components. There are multiple bus implementations, but in the early days of computers, a bus was simply a set of parallel wires, each carrying an electrical signal. This allowed multiple bits of data to be transferred in parallel; the voltage on each wire represented a single bit. Today's bus designs aren't always that simple, but the intent is similar.

There are three common bus types used in communication between the CPU, memory, and I/O devices. An *address bus* acts as a selector for the memory address that the CPU wishes to access. For example, if a program wishes to write to address 0x2FE, the CPU writes 0x2FE to the address bus. The *data bus* transmits a value read from memory or a value to be written to memory. So if the CPU wishes to write the value 25 to memory, then 25 is written to the data bus. Or if the CPU is reading data from memory, the CPU reads the value from the data bus. Finally, a *control bus* manages the operations happening over the other two buses. As examples, the CPU uses the control bus to indicate that a write operation is about to happen, or the control bus can carry a signal

indicating the status of an operation. Figure 7-15 illustrates how a CPU uses the address bus, data bus, and control bus to read memory.

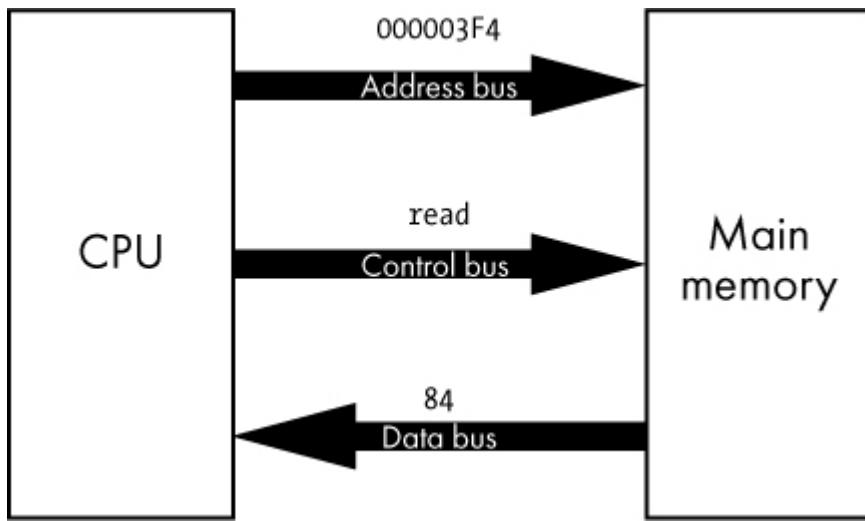


Figure 7-15: The CPU requests a read of the value at address 3F4, and the value of 84 is returned.

In the example shown in Figure 7-15, the CPU needs to read the value stored at memory address 000003F4. To do this, the CPU writes 000003F4 to the address bus. The CPU also sets a certain value on the control bus, indicating that it wishes to perform a read operation. These bus updates act as inputs to the memory controller (the circuit that manages interactions with main memory), telling it that the CPU wishes to read the value stored at address 000003F4 in main memory. In response, the memory controller retrieves the value stored at address 000003F4 (84 in this example) and writes it to the data bus, which the CPU can then read.

Summary

In this chapter, we covered computer hardware: a central processing unit (CPU) to execute instructions, random access memory (RAM) that stores instructions and data while powered, and input/output (I/O) devices that interact with the outside world. You learned that memory is composed of single-bit memory cells, implemented with a type of flip-

flop in SRAM, and with a transistor and capacitor in DRAM. We covered how memory addressing works, where each address refers to a byte of memory. You learned about CPU architectures, including x86 and ARM. We explored how CPUs work internally, looking at registers, the ALU, and the control unit. We covered secondary storage and other types of I/O, and finally, we looked at bus communication.

In the next chapter, we'll move beyond hardware to the thing that makes computers unique among devices—software. We'll examine the low-level instructions that processors execute, and we'll see how those instructions can be combined to perform useful operations. You'll have the opportunity to write software in assembly language and use a debugger to explore machine code.

8

MACHINE CODE AND ASSEMBLY LANGUAGE



We've covered the physical parts of a computer: the CPU, main memory, and I/O devices. Understanding the hardware of a computer is important, but hardware is only half the story. The magic of computers is in software. It's software that moves a computer from being a fixed-purpose device to a highly flexible device that can easily take on new abilities! In this chapter we cover low-level software—machine code and assembly language. I've found that these topics are best understood using an interactive approach, so the bulk of this chapter's content is in the projects.

Software Terms Defined

To discuss software, I need to first introduce several terms. Instructions that tell a computer what to do are known as *software*; this contrasts with hardware, the physical elements of a computer. An ordered set of software instructions that accomplishes a task is called a *program*, and *programming* is the act of writing such programs.

The term *application* is sometimes used synonymously with program, although *application* tends to refer to a program that interacts directly

with humans, rather than programs that interact with software or hardware. An application may also consist of multiple programs working together. The word *app* came into popular use around 2008, and tends to carry other connotations that I will cover in Chapter 13.

Another name for a set of software instructions is *computer code*, or just *code*. CPUs execute *machine code*, whereas software developers typically write their source code in a higher-level programming language. The term *source code* refers to the text of a program as originally written by developers. Such code usually isn't written in a form that CPUs understand directly, so additional steps must be taken before it can run on a computer. I will cover more details on source code and high-level programming languages in Chapter 9, but now let's look at the foundation of software: machine code.

Machine code is software in the form of binary *machine language* instructions. As described in Chapter 7, a CPU's architecture determines which instructions that particular CPU understands. In the same way that a human language is formed from a vocabulary, a machine language is formed from a list of instructions known to a CPU family. Vocabulary words arranged into sentences convey meaning, and CPU instructions arranged into programs do the same.

No matter how a program was originally written (and there are lots of ways to write programs), it eventually needs to execute on a CPU as a series of machine language instructions. As you might expect, CPU instructions boil down to a series of 0s and 1s, just like everything else a computer deals with. This is worth repeating: no matter how a program was originally written, no matter what programming language was used, no matter what technologies were involved, in the end, that program becomes a series of 0s and 1s, representing instructions that a CPU can execute.

Some years ago, I had a job that involved diagnosing software failures. Often, the problems I analyzed occurred in software written by other companies. I didn't have the source code for this software, nor did I have much information about how the software was supposed to work,

and yet my job was to determine why the software was failing! I had a coworker who took this in stride, and he regularly reminded me that “it’s just code.” In other words, the failing software was just a bunch of 1s and 0s that a CPU interpreted as instructions. If a CPU can make sense of the code, so can you.

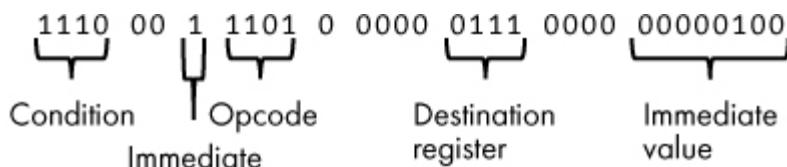
An Example Machine Instruction

I think the simplest way to jump into the topic of machine code is to look at an example. Let’s take a look at a specific machine instruction understood by the ARM family of processors. As you may recall, ARM processors are found in most smartphones, so this instruction is probably something your phone would understand.

Our example instruction tells the processor to move the number 4 into the r_7 register, one of several general-purpose registers on ARM processors. Recall from our previous discussion of computer hardware that a register is a small storage location within the CPU. The ARM instruction for doing that looks like this in binary:

11100011101000000111000000000100

Let’s examine how an ARM CPU would go about making sense of this instruction, as shown in Figure 8-1. Note that we’re skipping some of the bits that aren’t relevant to our discussion.



Details:

- Condition = 1110 = always execute (unconditional)
- Immediate = 1 = value is in the last 8 bits of instruction
- Opcode = 1101 = move value, usually represented as "mov"
- Destination register = 0111 = r_7
- Immediate value = 0000 0100 = 4 decimal

Figure 8-1: Decoding an ARM instruction

The *condition* section specifies the conditions under which the instruction should be executed. `1110` means the instruction is not conditional, so the CPU should always execute it. Although this is not the case in this example, some instructions only need to run under specific conditions. The next two bits, `00` in this example, aren't relevant to our discussion, so we'll skip them. The *immediate bit* tells us whether we're accessing a value in a register or accessing a value specified in the instruction itself (known as an *immediate value*). In this case, the immediate bit is `1`, so we're using a number specified within the instruction. If the immediate bit were `0`, the register that should be accessed would be specified elsewhere in the instruction's bits. The *opcode* represents the operation that the CPU is to perform. In this case, it's `mov`, meaning the CPU has to move some data. The *destination register* of `0111` tells us we're moving a value into register `r7` (`0111` is binary for seven). Finally, the immediate value of `00000100` is `4` in decimal, which is the number we want to move into register `r7`. To recap, this binary sequence tells an ARM CPU to move the number `4` into the `r7` register.

A CPU always deals with everything in binary, but most people have a hard time with all those 0s and 1s. Let's represent the same instruction in hexadecimal to make it somewhat easier to read:

`e3a07004`

Now isn't that better? Well, maybe not. It's more compact and easier to distinguish than binary, but its meaning still isn't obvious. Fortunately for us, there's yet another way to represent this instruction: assembly language. *Assembly language* (or *assembler language*) is a programming language in which each statement directly represents a machine language instruction. Each type of machine language has a corresponding assembly language—x86 assembly, ARM assembly, and so forth. An assembly language statement consists of a *mnemonic* that represents a CPU opcode, plus any required operands (such as a register or numeric value). A mnemonic is a human-readable form of an opcode,

allowing assembly language programmers to use `mov` instead of `1101` in their code. The same ARM instruction discussed earlier can also be represented using the following assembly language statement:

```
mov r7, #4
```

Compared to the corresponding binary and hexadecimal representations, this statement is certainly a better way to say, “move 4 into the `r7` register”! At least it’s easier to read for humans. That said, remember that the assembly language statement is just a convenience for people. A CPU never executes instructions in a text format, it only deals with the binary form of an instruction. If a programmer writes a program in assembly language, the assembly instructions must still be turned into machine code before a computer runs the program. This is accomplished using an *assembler*, a program that translates assembly language statements to machine code. An assembly language text file is fed into an assembler, and the output is a binary object file containing machine code, as illustrated in Figure 8-2.

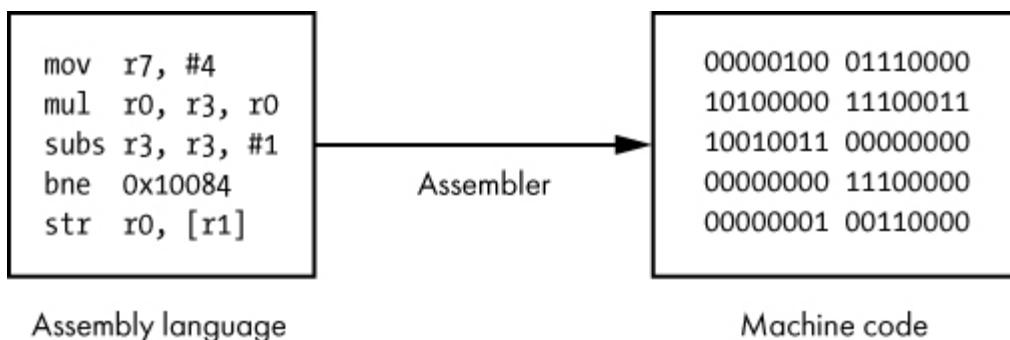


Figure 8-2: An assembler turns assembly language into machine code.

Calculating a Factorial in Machine Code

Now that we’ve examined a single ARM instruction, let’s see how multiple instructions can be combined to perform a useful task. Let’s look at some ARM machine code that calculates the factorial of an integer. As you may remember from math class, the factorial of n

(written as $n!$) is the product of the positive integers less than or equal to n . So as an example, the factorial of 4 is

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

Now that we have a definition for factorial, let's look at an implementation of a factorial calculation in ARM machine code. To keep things simple, we won't examine the full program code, just the part that does the factorial algorithm. We assume that initially the value of n is stored in the r0 register, and that when the code completes, the result of the calculation is also stored in r0.

Machine code, like any other data a computer deals with, must be loaded into memory before the CPU can access it. The following is a view of our machine code as 32-bit (4-byte) hexadecimal values, along with the memory address of each value.

Address	Data
0001007c	e2503001
00010080	da000002
00010084	e0000093
00010088	e2533001
0001008c	1afffffc

When our code loads into memory, the factorial logic starts at address `0001007c`. Let's examine the contents of memory starting at that address. Note that `0001007c` isn't a magic address; it just happens to be where the code loaded in this example. Note, also, that the memory address values increase by 4 because each data value requires 4 bytes of storage. Each ARM instruction is 4 bytes in length, so this data represents five ARM instructions.

Looking at these instructions as hexadecimal values doesn't give us much insight to their meaning, so let's decode the instructions so we can make sense of this program. In the following listing, I've converted the hexadecimal data values to their corresponding assembly language mnemonics. In case you're wondering, manually translating machine language to assembly language isn't something you need to know how to do! We have software for that called a *disassembler*. For now, this book

acts as your disassembler. Here's each instruction paired with its assembly statement:

Address	Data	Assembly
0001007c	e2503001	subs r3, r0, #1
00010080	da000002	ble 0x10090
00010084	e0000093	mul r0, r3, r0
00010088	e2533001	subs r3, r3, #1
0001008c	1afffffc	bne 0x10084
00010090	---	

The CPU executes these instructions sequentially until it hits a branching instruction (such as `ble` or `bne`), which may cause it to jump to another part of the program. Address `00010090` marks the end of the factorial logic. Once that address is reached, the factorial result has been stored in `r0`. At that point, the CPU executes whatever instruction happens to be at address `00010090`.

You may be wondering how these instructions represent a calculation of a factorial. For most people, a cursory look at such instructions isn't sufficient to understand the underlying intent. Taking a step-by-step approach and tracking the values of the registers as each instruction is executed can help you understand the program. I'll provide you with some needed background information, and then you can try evaluating how this program works.

To make sense of this program, you first need a description of each instruction used. In Table 8-1, I've given you an explanation of each instruction in this program. In this table, I've used placeholder names for registers such as `Rd` and `Rn`. When you review assembly code, you'll see actual register names used instead, such as `r0` or `r3`. The order of the operands listed in the code corresponds to the order of the operands in Table 8-1. For example, `subs r3, r0, #1` means subtract 1 from the value stored in `r0` and store the result in `r3`.

Table 8-1: Explanation of a Few ARM Instructions

Instruction	Details
subs Rd, Rn, #imm	Subtract the value in register <code>Rn</code> from the value in register <code>Rd</code> , and store the result in <code>Rd</code> . The immediate value <code>imm</code> is typically 1 or -1.
mul Rd, Rn, Rm	Multiply the values in registers <code>Rn</code> and <code>Rm</code> , and store the result in register <code>Rd</code> .
ble Rn, Rd, label	Branch if less than or equal to zero. If the value in register <code>Rn</code> is less than or equal to the value in register <code>Rd</code> , branch to the label <code>label</code> .
bne Rn, Rd, label	Branch if not equal. If the value in register <code>Rn</code> is not equal to the value in register <code>Rd</code> , branch to the label <code>label</code> .

Instruction	Details
<code>subs Rd, Rn, #Const</code>	<p>Subtract</p> <p>Subtracts constant value <code>const</code> from the value stored in register <code>Rn</code> and stores the result in register <code>Rd</code>.</p> <p>In other words, $Rd = Rn - \text{Const}$</p>
<code>mul Rd, Rn, Rm</code>	<p>Multiply</p> <p>Multiplies the value stored in register <code>Rn</code> and the value stored in register <code>Rm</code> and stores the result in register <code>Rd</code>.</p> <p>In other words, $Rd = Rn \times Rm$</p>
<code>ble Addr</code>	<p>Branch if less than or equal</p> <p>If the previous operation's result was less than or equal to 0, then jump to the instruction at address <code>Addr</code>. Otherwise, continue to the next instruction.</p>
<code>bne Addr</code>	<p>Branch if not equal</p> <p>If the previous operation's result was not 0, then jump to the instruction at address <code>Addr</code>. Otherwise, continue to the next instruction.</p>

BRANCHING AND THE STATUS REGISTER

The branch instructions don't actually look at the numeric result of the previous instruction. ARM processors, like most CPUs, have a register dedicated to tracking status. This status register has 32 bits, and each bit corresponds to a certain status flag. For example, bit 31 is the N flag, and it is set to 1 when an instruction results in a negative number. Only certain instructions affect the state of these flags. For example, the `subs` instruction alters the state of the flags. If a certain subtraction operation results in a negative result, the N flag is set; otherwise, it is cleared. Other instructions, including branching instructions, then look at the status flags to determine what to do. This may seem like a roundabout approach, but really, it simplifies things for instructions like `bne` —the processor can branch (or not) based on the value of a single bit.

We've reached the end of the explanatory content on this topic; the rest of the chapter is made up of an exercise and two projects. In Exercise 8-1, you'll walk through the example factorial program using the details found in Table 8-1 to understand how each instruction works.

EXERCISE 8-1: USE YOUR BRAIN AS A CPU

Try running the following ARM assembly program in your mind, or use pencil and paper:

Address	Assembly
0001007c	subs r3, r0, #1
00010080	ble 0x10090
00010084	mul r0, r3, r0
00010088	subs r3, r3, #1
0001008c	bne 0x10084
00010090	---

Assume an input value of $n = 4$ is initially stored in $r0$. When the program gets to the instruction at **00010090**, you've reached the end of the code, and $r0$ should be the expected output value of 24. I recommend that for each instruction, you keep track of the values of $r0$ and $r3$ before and after the instruction completes. Work through the instructions until you reach the instruction at **00010090** and see if you got the expected result. If things worked correctly, you should have looped through the same instructions several times; that's intentional. The answer is in Appendix A.

Walking through assembly language on paper is a great start, but trying out assembly language on a computer is even better.

NOTE

Please see Project #12 on page 145, where you can assemble the factorial code and examine it while it runs. Also, see Project #13 on page 155, where you can learn some additional approaches for examining machine code.

Summary

In this chapter we covered machine code, a series of CPU-specific instructions represented as bytes in memory. You learned how an example ARM processor instruction is encoded, and you saw how that instruction can be represented in assembly language. You learned that assembly language is a kind of source code, specifically a human-readable form of machine code. We saw how multiple assembly language statements can be combined to perform useful operations.

In the next chapter, we'll cover high-level programming languages. Such languages provide an abstraction from a CPU's instruction set, allowing developers to write source code that's easier to understand and portable across different computer hardware platforms.

PROJECT #12: FACTORIAL IN ASSEMBLY

Prerequisite: A Raspberry Pi, running Raspberry Pi OS. I recommend that you flip to Appendix B and read the entire “Raspberry Pi” section on page 341. That gets you set up and walks you through using Raspberry Pi OS, including how to work with files, which you do extensively in this chapter’s projects.

In this project, you’ll build a factorial program in assembly language, like the one we covered earlier in this chapter. You’ll then examine the generated machine code. The factorial program includes some additional code beyond what was included in the chapter. Specifically, the program also reads the initial value of *n* from memory, writes the result back to memory, and hands control back to the operating system at the end.

ASSEMBLY INSTRUCTIONS AND DIRECTIVES

Since you’re including this additional code, I’ve provided Table 8-2 to explain the various instructions used in the code. You saw some of these instructions already in Table 8-1, but I’m including everything here for easy reference.

Table 8-2: ARM Instructions Used in Project #12

Instruction	Details
<code>ldr Rd, Addr</code>	Load register from memory Reads the value at address <i>Addr</i> and puts it in register <i>Rd</i> .

Instruction	Details
<code>str Rd, Addr</code>	Store register to memory Writes the value in register <i>Rd</i> to address <i>Addr</i> .
<code>mov Rd, #Const</code>	Move constant to register Moves constant value <i>Const</i> to register <i>Rd</i> .
<code>svc</code>	Make system call Makes a request of the operating system.
<code>subs Rd, Rn, #Const</code>	Subtract Subtracts constant value <i>Const</i> from the value stored in register <i>Rn</i> and stores the result in register <i>Rd</i> . In other words, $Rd = Rn - Const$
<code>mul Rd, Rn, Rm</code>	Multiply Multiplies the value stored in register <i>Rn</i> and the value stored in register <i>Rm</i> and stores the result in register <i>Rd</i> . In other words, $Rd = Rn \times Rm$
<code>ble Addr</code>	Branch if less than or equal If the previous operation's result was less than or equal to 0, then jump to the instruction at address <i>Addr</i> . Otherwise, continue to the next instruction.
<code>bne Addr</code>	Branch if not equal If the previous operation's result was not 0, then jump to the instruction at address <i>Addr</i> . Otherwise, continue to the next instruction.

When writing code in assembly language, developers also use *assembler directives*. These aren't ARM instructions, but commands to the assembler. These directives start with a period, so they're easy to distinguish from instructions. In the following code, you also see text followed by a colon—these are *labels*, names given to a memory address. Since we don't know where instructions will be located in memory when we write the code, we refer to memory locations by labels instead of by memory addresses. One more thing to note: the @ sign indicates that the text following it (on the same line) is a comment. I've included comments to explain the program, but you can skip entering them if you prefer.

ENTER AND REVIEW THE CODE

That's enough background information; this is a project after all! Time to enter the code. Use the text editor of your choice to create a new file called *fac.s* in the root of your home folder. Detailed steps for using text editors on Raspberry Pi OS are included in the “Working with Files and Folders” section of the Raspberry Pi documentation on page 346. Enter the following ARM assembly code into your text editor (you don’t have to preserve indentation and empty lines, but be sure to maintain line breaks, although extra line breaks won’t hurt). Don’t worry if you don’t yet understand all of this code; I explain what you need to know following the code.

```
.global _start❶

.text❷
_start:❸
    ldr r1, =n      @ set r1 = address of n❹
    ldr r0, [r1]    @ set r0 = the value of n
    subs r3, r0, #1 @ set r3 = r0 - 1
    ble end        @ jump to end if r3 <= 0
loop:
    mul r0, r3, r0  @ set r0 = r3 x r0
    subs r3, r3, #1 @ decrement r3
    bne loop        @ jump to loop if r3 > 0
end:
    ldr r1, =result @ set r1 = address of result❺
    str r0, [r1]    @ store r0 at result

@ Exit the program
    mov r0, #0❻
    mov r7, #1
    svc 0

.data❾
n: .word 5❽
result: .word 0
```

After entering in your code, save it in the text editor as *fac.s* in the root of your home folder. Let’s walk through this code, starting with the directives and labels.

As mentioned earlier, text followed by a colon, like `_start:`, is a label for a memory location **❸**. The `_start` label marks the point at which the program begins execution. The `.global` directive makes the `_start` label visible to the linker **❶** (we’ll get to the linker in a minute) so that it can be set as the entry point for the program. The `.text` directive tells the assembler that the lines following it are instructions **❷**.

At the end of the code, the `.data` directive tells the assembler that the lines following it are data **❾**. In the data section, the program is storing two 32-bit values, each indicated by the `.word` directive **❽**. The first is the value of `n`, initially set to a value of 5. The

second is `result`, initially set to a value of 0. In this context, “word” means 4 bytes, or 32 bits.

Now let’s look at the functional additions to the code beyond what was in the chapter. We now have code that loads `n` from memory, saves the factorial result to memory, and exits the program. The first two instructions in `_start` load the value of `n` from a location in memory ④. The `ldr` instruction loads a register with a value. We reference the address of `n` with `=n`. On the next line, `[r1]` is in brackets because the program is accessing the value stored at the address in `r1`.

The two instructions following `end` save the result to a location in memory ⑤. The first instruction moves the address of the memory location named `result` into the `r1` register. After that, the code stores the value in the `r0` register (which happens to be the calculated factorial) to the `result` memory address, referenced by `r1`.

The last three instructions in the `.text` section are used to cleanly exit the program ⑥. This requires the help of the operating system, so I’ll skip over the details of these instructions until we cover operating systems in Chapter 10.

ASSEMBLE, LINK, AND RUN

You now have a text file with assembly language instructions, but this isn’t a format that a computer can run. You need to turn the assembly language instructions into bytes of machine code through a two-step process. First, you need to convert (assemble) the instructions to machine code bytes using an *assembler*. The result of this process is an *object file*, a file that contains the bytes of your program but still isn’t in the final format needed to run the program. Next, you need to use a program called a *linker* to turn your object file into an executable file that the operating system can run. This process is illustrated in Figure 8-3.

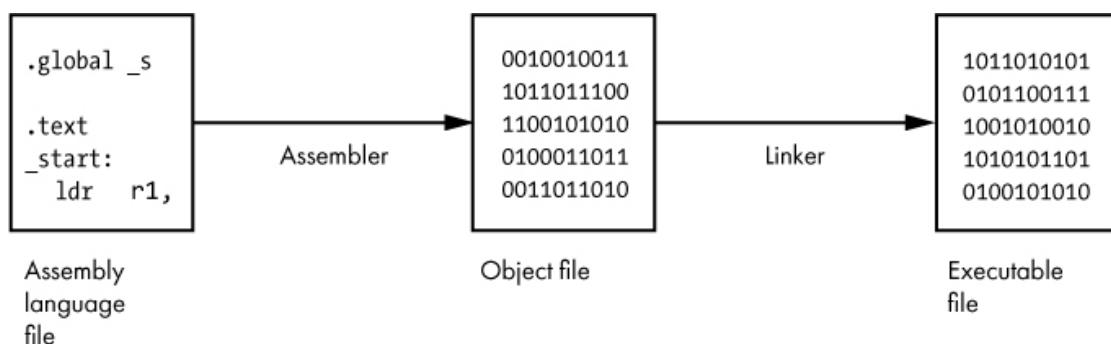


Figure 8-3: Assembling and linking produces an executable file

You may wonder why this two-step process is needed. If you’re assembling multiple source files that all work together as one program, each source file assembles into an object file. The linker then combines the various object files into one executable file. This allows for object files that were created previously to be linked as needed. In this

case, you have just one object file, and the linker simply turns it into a format that's ready to execute.

Now, assemble your code:

```
$ as -o fac.o fac.s
```

The `as` tool is the GNU Assembler, which turns your assembly language statements into machine code. This command writes the generated machine code to a file called `fac.o`, an object file. The assembler may give you a warning if your `fac.s` file doesn't end with a line break—you can safely ignore this warning.

Once your source code has been assembled into an object file, you need to use the GNU linker (`ld`) to convert your object file to an executable file:

```
$ ld -o fac fac.o
```

This command takes `fac.o` as an input, and outputs an executable file named `fac`. At this point, you can run your program with the following command:

```
$ ./fac
```

This command should immediately return to the next line with no output. This is because your program doesn't actually display any text to the screen. It simply calculates a factorial, saves the result in memory, then exits. To interact with the user, the program would need to request some help from the operating system. However, since we're trying to keep this program as minimal as possible, you don't need to do that.

LOAD THE PROGRAM WITH A DEBUGGER

Since your program doesn't output anything, how can you tell what it's doing? You can use a *debugger*, a program that can examine a process as it runs. A debugger can attach to a running program and then halt its execution. While the program is halted, the debugger can examine the registers and memory of the target process. Here, you use the GNU Debugger, `gdb`, as your debugger, and your target is the `fac` program.

To start, just execute the following command:

```
$ gdb fac
```

When you run this command, `gdb` loads the `fac` file but no instructions execute yet. From the (`gdb`) prompt, enter the following to view the start address of the program:

```
(gdb) info files
```

You should see a line like this, although the specific address may differ:

Entry point: 0x10074

This tells you that the program's entry point is address `0x10074`. Remember, when you wrote the program, you didn't know what memory addresses would be used, so you used labels instead. Now that the program has been built and loaded into memory, you have real memory addresses to examine. This entry point address corresponds to the `_start` label, since that's where the program begins. You can now use `gdb` to disassemble the machine code starting at the program entry point. *Disassembly* is the process of viewing machine code bytes as assembly language instructions. The following command uses `0x10074` as the start address; if your entry point is different, use that address instead.

```
(gdb) disas 0x10074
Dump of assembler code for function _start:
0x00010074 <+0>: ldr      r1, [pc, #40]    ; 0x100a4 <end+20>
0x00010078 <+4>:  ldr      r0, [r1]
0x0001007c <+8>:  subs    r3, r0, #1
0x00010080 <+12>: ble     0x10090 <end>
```

After running this command, you should see the first four instructions disassembled, as shown here. By default, `gdb` only disassembles a handful of instructions. That's a good start, but it is better to see the entire program. To do that you need to tell `gdb` the ending address of the code you want to see. If you look back at the earlier code you entered into `fac.s`, you can see that there are 12 instructions total in your program. Each instruction is 4 bytes, so the program should be 48 bytes in length. This means your program should end 48 bytes after the start address, so the ending address should be `0x00010074 + 48`. You can do this addition by hand or in a calculator program, but since you're in `gdb`, you can ask it to do that math for you and find the ending address of your program (again, replace `0x10074` with your entry point address if you need to):

```
(gdb) print/x 0x00010074 + 48
$1 = 0x100a4
```

The `print` command output can be a bit confusing at first. The `/x` in the command means “print the result in hexadecimal.” If you look at the output, the left-hand value (`$1`) is a *convenience variable*, a temporary storage location in `gdb`. Saving a value in a convenience variable is `gdb`'s way of making it easy for you to get back to this result later. The value after the equals sign is the printed value, the result of the calculation, `0x100a4` in this case.

So now you know the ending address (`0x100a4`), and you can ask `gdb` to disassemble the entire program. Note that if your starting address is different than mine, you need to replace the two addresses in the following command.

```
(gdb) disas 0x10074,0x100a4
Dump of assembler code from 0x10074 to 0x100a4:
0x00010074 <_start+0>: ldr      r1, [pc, #40]    ; 0x100a4
<end+20>①
```

```

0x00010078 <_start+4>:    ldr      r0, [r1]
0x0001007c <_start+8>:    subs     r3, r0, #1
0x00010080 <_start+12>:   ble     0x10090 <end>
0x00010084 <loop+0>:    mul     r0, r3, r0
0x00010088 <loop+4>:    subs     r3, r3, #1
0x0001008c <loop+8>:    bne     0x10084 <loop>
0x00010090 <end+0>:    ldr      r1, [pc, #16] ; 0x100a8 <end+24>❷
0x00010094 <end+4>:    str      r0, [r1]
0x00010098 <end+8>:    mov      r0, #0
0x0001009c <end+12>:   mov      r7, #1
0x000100a0 <end+16>:   svc      0x00000000

```

This looks very much like what you originally entered into *fac.s* and assembled, except now each instruction has been assigned an address, and the references to *n* and *result* have been replaced with memory offsets relative to the program counter register (for example, `[pc, #40]` ❶). The *program counter* register, or *instruction pointer*, holds the memory address of the current instruction. For the sake of keeping things simple, I'm not going into the details of why program counter offsets are used here, but just know that the instructions at `0x10074` ❶ and `0x10090` ❷ are loading the memory addresses of *n* and *result*, respectively, into *r1*.

RUN AND EXAMINE THE PROGRAM USING DEBUGGER BREAKPOINTS

Now that you can see the program loaded into memory, let's see if the program works as expected. To do this, you're going to set breakpoints on certain instructions, which allows you to examine the state of your program at that point. A *breakpoint* tells the debugger to halt execution when a certain address is reached. Setting a breakpoint on a certain address halts execution immediately *before* the corresponding instruction is executed. In the following example commands, I use the addresses shown on my system, but if your memory addresses are different, be sure to use those addresses instead.

You're going to set the following breakpoints:

0x10074 The start of the program.

0x1007c The beginning of the factorial logic, 8 bytes after the first instruction. When the program reaches this instruction, register *r0* should be the input value *n*, which was hard-coded to 5 in the program.

0x10090 The end of the factorial logic, 0x1C bytes after the first instruction. When the program reaches this instruction, register *r0* should hold the factorial value that was calculated.

0x100a0 The final instruction of the program. When the program reaches this instruction, the memory location labeled *result* should hold the factorial result.

Set the breakpoints as follows (again, adjust the addresses if your start address isn't `0x10074`):

```
(gdb) break *0x10074
(gdb) break *0x1007c
(gdb) break *0x10090
(gdb) break *0x100a0
```

Now begin running the program:

```
(gdb) run
Starting program: /home/pi/fac

Breakpoint 1, 0x00010074 in _start ()
```

You should see output like this, indicating that execution stopped at the first breakpoint. At this point the program is ready to execute the first instruction, and you can take a look around at the state of things. First, examine the registers, and really, the only one we care about at this point is the program counter (pc), because we want to confirm that the current instruction is the start address of 0x10074. Now ask the debugger to show the value of the pc register:

```
(gdb) info register pc
pc          0x10074  0x10074 <_start>
```

This tells you that the program counter is pointing to the start address and first breakpoint, as expected. Another way to confirm the current instruction is to simply disassemble the current code like so:

```
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010074 <+0>:    ldr      r1, [pc, #40]   ; 0x100a4 <end+20>
  0x00010078 <+4>:    ldr      r0, [r1]
  0x0001007c <+8>:    subs    r3, r0, #1
  0x00010080 <+12>:   ble     0x10090 <end>
```

Note the => symbol indicating the current instruction.

Now that you've confirmed that the program is ready to run its first instruction, you can examine the current values of the two labeled memory addresses: n and result. These should be 5 and 0, respectively, since that's what you defined their initial values as in the *fac.s* source code. You can again use the print command to see these values. When you do so, you need to specify a data type of int (a 32-bit integer) so the print command knows how to display these values.

```
(gdb) print (int)n
$2 = 5
(gdb) p (int)result
$3 = 0
```

Note how `p` is substituted for `print` in the second command. Shortened versions of commands are supported by `gdb`; this can save you some typing. As you can see, the `print` command makes it easy to print the values of labeled memory locations!

Although printing the value of a labeled memory location is convenient, it does raise a question: How does the `print` command in `gdb` know about the labels you gave these memory locations in your original `fac.s` file? The CPU doesn't use these labels; it just uses memory addresses. The machine code doesn't refer to these memory locations by name either. The debugger is able to do this because the file that holds the machine code, `fac`, also holds symbolic information. These *debug symbols* tell the debugger about certain named memory locations, such as `n` and `result`. Usually symbolic information is removed from executable files before they are distributed to end users, but the symbolic info is still present in your `fac` executable file.

Keeping in mind that `n` and `result` are just labels for memory locations, how do you find the actual memory addresses of these variables? One way is to print the address by using the `&` operator, which means "address of" in `gdb`. So `&n` means "the address of `n`." Now print the address of `n` and the address of `result`.

```
(gdb) p &n
$4 = (<data variable, no debug info> *) 0x200ac
(gdb) p &result
$5 = (<data variable, no debug info> *) 0x200b0
```

This tells you that the value of `n` is stored at address `0x200ac`, and the value of `result` is stored at address `0x200b0`. Note that these are consecutive values in memory, since both `n` and `result` are 4 bytes in length. You can examine this memory using the `x` command:

```
(gdb) x/2xw 0x200ac
0x200ac: 0x00000005 0x00000000
```

The `x/2xw` command means examine two consecutive values, displayed in hex, each "word" sized (4 bytes), starting at address `0x200ac`. So here again you can see that `n` is 5, and `result` is 0. This is just a different way of looking at memory, this time without using named labels.

So back to the program—you've now established that the initial memory values are set as expected. Continue execution to your next breakpoint, where you can verify that `r0` has been set to the initial value of `n`.

```
(gdb) continue
Continuing.

Breakpoint 2, 0x0001007c in _start ()

(gdb) disas
Dump of assembler code for function _start:
0x00010074 <+0>: ldr    r1, [pc, #40] ; 0x100a4 <end+20>
```

```
0x00010078 <+4>:    ldr    r0, [r1]
=> 0x0001007c <+8>:    subs   r3, r0, #1
  0x00010080 <+12>:   ble    0x10090 <end>
End of assembler dump.
```

```
(gdb) info registers r0
r0          0x5           5
```

From the preceding output you can see that the program has moved forward to instruction `0x1007c` as expected, and `r0` has the expected value of 5 (the value of `n`). So far, so good. Now, move ahead to your next breakpoint, where `r0` should now be the calculated value of 5 factorial, which is 120. You can shorten the `continue` command to just `c`, and the `info registers` command to just `i r`.

```
(gdb) c
Continuing.
```

```
Breakpoint 3, 0x00010090 in end ()
```

```
(gdb) disas
Dump of assembler code for function end:
=> 0x00010090 <+0>:    ldr    r1, [pc, #16]    ; 0x100a8 <end+24>
  0x00010094 <+4>:    str    r0, [r1]
  0x00010098 <+8>:    mov    r0, #0
  0x0001009c <+12>:   mov    r7, #1
  0x000100a0 <+16>:   svc    0x00000000
  0x000100a4 <+20>:   andeq r0, r2, r12, lsr #1
  0x000100a8 <+24>:   strreq r0, [r2], -r0    ; <UNPREDICTABLE>
End of assembler dump.
```

```
(gdb) i r r0
r0          0x78          120
```

That all looks good. Recall that at this point the factorial output hasn't been saved to the `result` memory address. Now verify that `result` is unchanged:

```
(gdb) p (int)result
$6 = 0
```

Although you have the factorial output temporarily stored in `r0`, it hasn't been written to memory yet. Continue to the end of program (the final breakpoint) and see if the `result` memory location has been updated.

```
(gdb) c
Continuing.
```

```
Breakpoint 4, 0x000100a0 in end ()
```

```
(gdb) p (int)result
$7 = 120
```

You should see a value of 120 for `result`. If so, nice work, your program worked as expected!

HACK THE PROGRAM TO CALCULATE A DIFFERENT FACTORIAL

This program is hard-coded to calculate the factorial of 5. What if you want it to calculate the factorial of some other number? Well, you could change the hard-coded value in the `fac.s` source code, rebuild the code, and run it again. Or you could write some code that allows the user to input a desired value of `n` at runtime. But imagine that you don't have access to the source code anymore, and you just want a quick way to alter this program's behavior while it runs, replacing the hard-coded value of `n` with some value other than 5.

First, restart the program using the `run` command, and answer `y` to the question:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi/fac

Breakpoint 1, 0x00010074 in _start ()
```

Now you're back at the beginning of the program, at breakpoint 1. You can edit the in-memory value of `n`, setting it to 7 rather than 5. First, get the memory address of `n`, then set the value at that address to 7. Then you can print out `n` to make sure the change worked.

```
(gdb) p &n
$8 = (<data variable, no debug info> *) 0x200ac

(gdb) set {int}0x200ac = 7

(gdb) p (int)n
$9 = 7
```

Now go to the end of the program and see if `result` gets updated to the expected value of 7 factorial, which is 5,040. You can get rid of your two middle breakpoints (numbers 2 and 3), since you want to go straight to the end:

```
(gdb) disable 2
(gdb) disable 3

(gdb) c
Continuing.

Breakpoint 4, 0x000100a0 in end ()

(gdb) p (int)result
$10 = 5040
```

You should see a value of 5,040 for `result`. If so, you've just successfully hacked a program to make it do your bidding—all without touching the source code!

At this point you may want to try setting `n` to other values and see if you get the expected results. To do this, restart the program using the `run` command, edit the in-memory value of `n`, continue to the final breakpoint, and check the value of `result`. However, if you use a value of `n` larger than 12, you get an incorrect result. See the answer to Exercise 8-1 in Appendix A for the reason why this is so.

If you allow the program to `continue` to the end, the process exits, and you should get a message like `Inferior 1 (process 946) exited normally`. This isn't an insult of your code, rather "inferior" is just how `gdb` refers to the target being debugged! You can exit the debugger at any time by entering `quit` in `gdb`.

PROJECT #13: EXAMINING MACHINE CODE

Prerequisite: Project #12.

Let's say that you were given the `fac` executable file, but not the original assembly language source file. You want to know what the program does, but you don't have the source code. As you saw in Project #12, you can use the `gdb` debugger to examine the `fac` executable file. In this project, I'll show you a different set of tools for examining machine code.

Open a terminal on your Raspberry Pi. By default, the terminal should open to the home folder, indicated by the `~` character. In this folder you should have three factorial-related files from the last project. Check this with the following command:

```
$ ls fac*
```

You should see

`fac` The executable file

`fac.o` The object file generated during assembly

`fac.s` The assembly language source code

In our fictional scenario, you only have the executable `fac` file, and you want to know what you can learn about the program from the contents of this file. First, look at the bytes contained in the file as hexadecimal values by using the `hexdump` tool:

```
$ hexdump -C fac
```

The beginning of the `hexdump` output should look something like Figure 8-4 (without the annotations), displaying the bytes in the `fac` executable file.

ELF	
00000000	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
00000010	02 00 28 00 01 00 00 00 74 00 01 00 34 00 00 00 ..(.....t...4...
00000020	94 02 00 00 00 02 00 05 34 00 20 00 02 00 28 00 4.(.
00000030	07 00 06 00 00 00 00 00 00 00 00 00 00 00 01 00
00000040	00 00 01 00 00 00 00 00 00 00 00 05 00 00 00 00
00000050	00 00 01 00 01 00 00 00 00 00 00 00 ac 00 02 00
00000060	ac 00 02 00 08 00 00 00 08 00 00 00 06 00 00 00
00000070	00 00 01 00 28 10 9f e5 00 00 91 e5 01 30 50 e2 (.....OP.
00000080	02 00 00 da 93 00 00 e0 01 30 53 e2 fc ff ff 1a OS.....
00000090	10 10 9f e5 00 00 81 e5 00 00 a0 e3 01 70 a0 e3 p..
000000a0	00 00 00 ef ac 00 02 00 b0 00 02 00 05 00 00 00
000000b0	00 00 00 00 41 13 00 00 00 61 65 61 62 69 00 01 A....aeabi..

Machine code starts at 0074

Data section starting at 00ac

Figure 8-4: Hex dump of Linux executable file

What you see is simply a sequential listing of the bytes in the file, each displayed as a two-character hexadecimal value. If the output of this command is too large to fit in your terminal window, scroll up to see the beginning bytes. The eight-character hex numbers along the left-hand column represent the offset into the file of the first byte in the corresponding row. There are 16 bytes on each row, meaning the offset number of each row (along the left) increases by 0x10. On the right-hand side of the output are the same bytes interpreted as ASCII. Bytes that do not correspond to a printable ASCII character code are indicated with a period.

At offset `00000000`, the very beginning of the file, you should see a `7F`, followed by `4c 46`, or in ASCII, `ELF`. This is an indicator that this is a file that is in *executable and linkable format (ELF)*. *ELF files* are the standard Linux format for executable programs. These 4 bytes mark the beginning of the *ELF header*, a set of properties that describe the contents of the file. Following the ELF header is a *program header*, which provides details needed by the operating system to run the program.

Now move past the headers and find the text section that contains the program's machine instructions. On my system, offset `00000074` is the beginning of the text section, and it starts with bytes `28 10 9f e5`. If you rearrange these bytes last-to-first, you get `e59f1028`, which is the machine code instruction for `ldr r1, [pc, #40]`. Each set of 4 bytes in this section is a machine instruction. Looking at the program in this way is a good reminder that the code of the `fac` program is simply represented as a sequence of bytes. Refer to Figure 8-1 for a reminder of how machine code is represented in binary.

Later in the output, at offset `000000ac` on my system, you should see the data section of the file, containing the two initial 4-byte values defined by the program. You don't see the `n` and `result` labels here, but you should see `05 00 00 00` and `00 00 00 00`. The offset of these bytes on your system may differ from mine.

As a side note, the order in which computers store bytes of data for larger numerical values is known as *endianness*. When a computer stores the least significant byte first (at the lowest address) this is called *little-endian*. Storing the most significant byte first is called *big-endian*. In the `hexdump` output, you see little-endian storage, since the 32-bit machine instruction of `e59f1028` was stored as bytes in this order: `28 10 9f e5`. The least significant byte was stored first. The same can be said of the values of `n` and `result`. The value of `n` is stored as `05 00 00 00`, meaning `00000005` when you consider it as a 32-bit integer.

If you want to view parts of this hexadecimal data, but grouped into sections, you can use the `objdump` tool:

```
$ objdump -s fac
```

This dumps out some of the same bytes as before, but grouped into sections, like so:

```
Contents of section .text:  
10074 28109fe5 000091e5 013050e2 020000da ( .....0P.....  
10084 930000e0 013053e2 fcffff1a 10109fe5 .....0S.....  
10094 000081e5 0000a0e3 0170a0e3 000000ef .....p.....  
100a4 ac000200 b0000200 .....  
Contents of section .data:  
200ac 05000000 00000000 .....  
Contents of section .ARM.attributes:  
0000 41130000 00616561 62690001 09000000 A....aeabi.....  
0010 06010801 ....
```

Note how the numbers along the left-hand side have changed. Instead of starting at `0074`, the `.text` section (that is, code) starts at `10074`. Instead of starting at `00ac`, the `.data` section containing the values of `n` and `result` starts at `200ac`. The `hexdump` tool simply shows the byte offset within the file, whereas `objdump` output refers to the address where the bytes are loaded in memory when the program runs. Another way to view the addresses of the various sections in an ELF executable file is with `readelf -e fac`. This displays the headers in the file.

You can now try another feature of `objdump`, disassembly of machine code, so you can see the assembly language instructions alongside the machine code byte values.

```
$ objdump -d fac
```

```
fac:      file format elf32-littlearm
```

```
Disassembly of section .text:
```

```
00010074 <_start>:  
 10074:    e59f1028      ldr      r1, [pc, #40] ; 100a4  
<end+0x14>①  
 10078:    e5910000      ldr      r0, [r1]  
 1007c:    e2503001      subs    r3, r0, #1  
 10080:    da000002      ble     10090 <end>
```

```
00010084 <loop>:  
10084:    e0000093      mul    r0, r3, r0  
10088:    e2533001      subs   r3, r3, #1  
1008c:    1affffffc      bne    10084 <loop>  
  
00010090 <end>:  
10090:    e59f1010      ldr    r1, [pc, #16] ; 100a8 <end+0x18>  
10094:    e5810000      str    r0, [r1]  
10098:    e3a00000      mov    r0, #0  
1009c:    e3a07001      mov    r7, #1  
100a0:    ef000000      svc    0x00000000  
100a4:    000200ac      .word  0x000200ac  
100a8:    000200b0      .word  0x000200b0
```

You should expect to see output similar to what is shown here. Note that the instruction at address **10074** ❶ is the same sequence of bytes highlighted in Figure 8-4, the first 4 bytes of machine code. This output is very similar to the output from `gdb` in the previous project. Consider what this means: using tools like `gdb` or `objdump`, you can easily view the machine code and corresponding assembly language for any executable!

Using the techniques I've described in the preceding pages, you can get a view of the contents of an ELF executable file. This applies to any standard ELF file on a Linux system, not just code you wrote. Feel free to explore the machine code of any ELF file on your computer. For example, say you want to see the machine code for `ls`—the tool you used earlier to list the contents of a directory. First, you need to find the filesystem location of the `ls` ELF file, like so:

```
$ whereis ls  
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

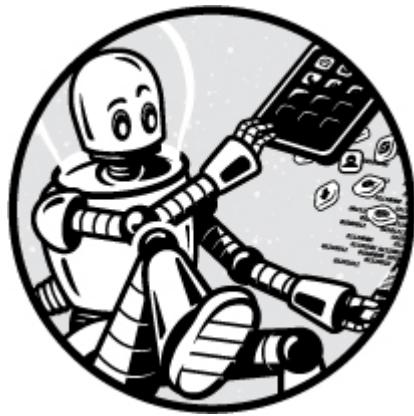
This tells us the binary executable file for `ls` is located at `/bin/ls` (you can ignore any additional results returned). Now you can run `objdump` (or any of the other tools already covered) to see the machine code for `ls`:

```
$ objdump -d /bin/ls > ls.txt
```

The output of this command is rather long, so it is redirected to a file named `ls.txt`. You don't see the disassembled code in the terminal window; instead it is written to the `ls.txt` file, which you can view using the text editor of your choice. Of course, since Linux is open source, you can just look at the source code for the `ls` tool online. However, not everything is open source, and this project should give you an idea of how you can view the disassembled code for any Linux executable program.

9

HIGH-LEVEL PROGRAMMING



In the last chapter, we looked at the fundamentals of software: machine code that runs on processors, and assembly language, a human-readable representation of machine code. Although eventually all software must take the form of machine code, most software developers work at a higher, more abstract level. In this chapter, you learn about high-level programming. We cover an overview of high-level programming, discuss common elements found across various programming languages, and look at example programs.

High-Level Programming Overview

Although it's possible to write software in assembly language (or even machine code!), doing so is time-consuming and error-prone, and it results in software that is hard to maintain. Furthermore, assembly language is specific to a CPU architecture, so if an assembly developer wishes to run their program on another type of CPU, the code must be rewritten. To address these shortcomings, *high-level programming languages* were developed; these allow programs to be written in a language that is independent from a specific CPU and is syntactically

closer to human language. Many of these languages require a *compiler*, a program that converts high-level program statements to machine code for a specific processor. Using a high-level language, a software developer can write a program once and then compile it for multiple types of processors, sometimes with little or no change to the source code.

The output of a compiler is an object file that contains machine code for a specific processor. As we covered in Project #12, object files aren't in the correct format for a computer to execute. Another program, called a *linker*, is used to convert one or more object files into an executable file that the operating system can then run. The linker can also bring in other libraries of compiled code when needed. The process of compiling and linking is illustrated in Figure 9-1.

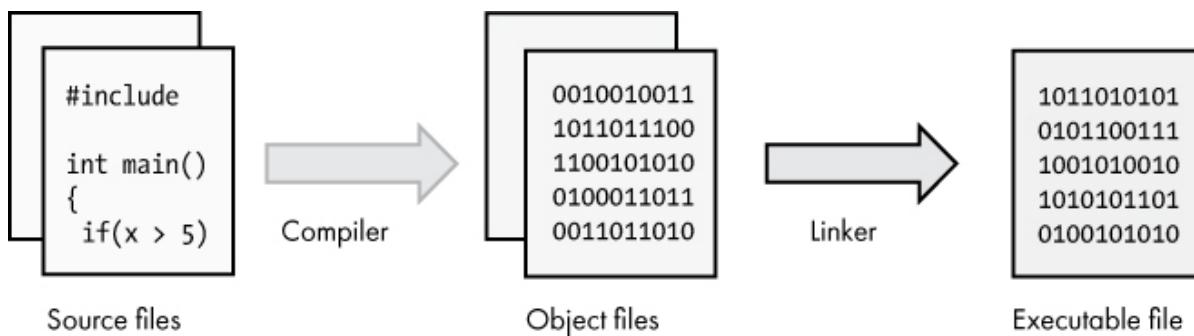


Figure 9-1: Building executable software from source code

The process of compiling and linking is referred to as *building* software. However, in common usage, software developers sometimes speak of *compiling* their code when they really mean the entire process of compiling, linking, and any other steps required to get their code into its final form. Compilers often invoke the linking step automatically, making it less visible to the software developer.

Introduction to C and Python

The best way to learn about high-level programming is to examine programming languages and write some programs in those languages.

For this chapter, I've chosen two high-level languages: C and Python. Both are powerful and useful, and they illustrate how programming languages tend to provide similar functionality but in different ways. Let's begin with a brief introduction to each.

The *C* programming language dates back to the early 1970s, when it was used to write a version of the Unix operating system. Despite being a high-level language, C isn't that far removed from underlying machine code, making it a great choice for operating system development or other software that interfaces directly with hardware. An updated version of C known as *C++* came about in the 1980s. C and C++ are powerful languages that can be used to accomplish nearly anything. However, these languages are complex and don't provide many safeguards against programmer mistakes. They remain a popular choice for programs that need to interface with hardware and those that require high performance, such as games. C is also useful for educational purposes, providing a straightforward mapping between low-level and high-level concepts, which is why I chose it for this chapter.

Compared to C, the *Python* programming language is further removed from underlying hardware. Initially released in the 1990s, Python has grown in popularity over the years. It's known for being easy to read and simple for beginners, while still providing everything necessary to support complex software projects. Python has a "batteries included" philosophy, meaning a standard distribution of Python includes a library of helpful capabilities that developers can easily use in their projects. The straightforward nature of Python makes it a good choice for teaching concepts of programming.

Let's now look at elements found across most high-level programming languages. The goal isn't to teach you to be a programmer in a specific language, but to instead familiarize you with the ideas commonly found in programming languages. Remember that the capabilities found in high-level programming languages are abstractions of CPU instructions. As you know, CPUs provide

instructions for memory access, math and logic operations, and control of program flow. Let's look at how high-level languages expose these underlying capabilities.

Comments

Let's begin with a feature of programming languages that doesn't actually instruct the CPU to do anything! Nearly all programming languages provide a way to include comments in code. A *comment* is text in source code that provides some insight into the code. Comments are intended to be read by other developers and are typically ignored by the compiler; they have no effect on the compiled software. In the C programming language, comments are specified like so:

```
/*
    This is a C-style comment.
    It can span multiple lines.
*/
// This is a single-line C comment, originally introduced in C++.
```

Python uses the hash character for comments, like this:

```
# This is a comment in Python.
```

Python doesn't provide any particular support for multiline comments; a programmer can simply use multiple single-line comments, one after another.

Variables

Memory access is a fundamental capability of processors, and therefore it must be a feature of high-level languages as well. The most basic way that programming languages expose memory is through variables. A *variable* is a named storage location in memory. Variables allow programmers to give a *name* to a memory address (or range of memory addresses) and then access data at that address. In most programming

languages, variables have a *type*, indicating what sort of data they hold. For example, a variable may be an integer type or a text string type. Variables also have a *value*, which is the data stored in memory. Although it's often hidden from the programmer, variables also have an *address*, the location in memory where the variable's value is stored. Lastly, variables have *scope*, meaning they can only be accessed from certain parts of the program, the parts where they are “in scope.”

Variables in C

Let's look at an example of a variable in the C programming language.

```
// Declare a variable and assign it a value in C.  
int points = 27;
```

This code *declares* a variable named `points` with a type of `int`, which in the C language means the variable holds an integer. The variable is then *assigned* a value of 27. When this code runs, the value of 27 decimal is stored at a memory address, but the developer doesn't need to worry about the specific address where the variable is stored. Most C compilers today treat an `int` as a 32-bit number, so at *runtime* (the time when a program executes) 4 bytes are allocated for this variable ($4 \text{ bytes} \times 8 \text{ bits per byte} = 32 \text{ bits}$), and the memory address of the variable refers to the first byte.

Let's now declare a second variable and assign it a value; then we can look at how the two variables are allocated in memory.

```
// Two variables in C  
int points = 27;  
int year = 2020;
```

Now we have two variables, `points` and `year`, declared one after another. Both are integers, so they each require 4 bytes for storage. The variables can be stored in memory as shown in Table 9-1.

Table 9-1: Variables Stored in Memory

Address	Variable name	Variable value
---------	---------------	----------------

Address	Variable name	Variable value
0x7efff1cc	?	?
0x7efff1d0	year	2020
0x7efff1d4	points	27
0x7efff1d8	?	?

The memory addresses used in Table 9-1 are just examples; the actual addresses vary depending on the hardware, operating system, compiler, and so forth. Note that the addresses increment by four, since we're storing 4-byte integers. The addresses before and after the known variables have a question mark for the variable name and value, since based on the preceding code, we don't know what might be stored there.

NOTE

Please see Project #14 on page 184, where you can look at variables in memory.

As the name *variable* implies, the value of a variable can change. If our earlier C program needed to set the value of `points` to another value, we could simply do this later in the program:

```
// Setting a new points value in C
points = 31;
```

Note that unlike our previous code snippet in C, this code does not specify `int` or any other type before the variable name. We only need to specify the type when the variable is initially declared. In this case, the variable was declared earlier, so here we just assign it a value. However, the C language requires that the variable's type remain the same, so once `points` is declared as an `int`, only integers can be assigned to that variable. Attempting to assign another type, such as a text string, results in a failure when the code is compiled.

Variables in Python

Not all languages require a declaration of type. Python, for example, allows a variable to be declared and assigned like so:

```
# Python allows new variables without specifying a type.  
age = 22
```

Now, in this case, Python recognizes that the type of data is an integer, but the programmer doesn't have to specify this. Unlike in C, the variable's type can change over time, so the following is valid in Python:

```
# Assigning a variable a value of a different type is valid in Python.  
age = 22  
age = 'twenty-two'
```

Let's take a closer look at what is actually happening in this example. A Python variable has no type, but the value to which it refers does have a type. This is an important distinction: the type is associated with the value, not the variable. A Python variable can refer to a value of any type. So when the variable is assigned a new value, it isn't really that the variable's type is changing, but rather that the variable has been bound to a value of a different type. Contrast this with C, where the variable itself has a type and can only hold values of that type. This difference explains why a variable in Python can be assigned values of different types, whereas a variable in C cannot.

NOTE

Please see Project #15 on page 186, where you can change the type of value referenced by a variable in Python.

Stack and Heap Memory

When a programmer uses a high-level language to access memory, the details of how that memory is managed behind the scenes is somewhat obscured, depending on the programming language in use. A programming language like Python makes the details of memory allocation nearly invisible to the programmer, whereas a language like C

exposes some of the underlying memory management mechanisms. Whether the details are exposed to the programmer or not, programs commonly make use of two types of memory: stack and heap.

The Stack

The *stack* is an area of memory that operates on a *last-in first-out (LIFO)* model. That is, the last item put on the stack is the first item that comes off the stack. You can think of a memory stack as being like a stack of plates. When you add to a stack of plates, you add the newest plate to the top. When the time comes to take a plate from the stack, you remove the top plate first. This does not mean that the items on the stack can only be *accessed* (read or modified) in LIFO order. In fact, any item currently on the stack can be read or modified at any time. However, when it comes time to remove unneeded items from the stack, the items are discarded from the top down, meaning the last item placed on the stack is the first to go.

The memory address of the value on the top of the stack is stored in a processor register known as the *stack pointer*. When a value is added to the top of the stack, the stack pointer's value is adjusted to increase the size of the stack and make room for the new value. When a value is removed from the top of the stack, the stack pointer is adjusted to decrease the size of the stack.

The compiler generates code that uses the stack to track the state of a program's execution and as a place to store local variables. The mechanics of this are transparent to a programmer in a high-level language. Figure 9-2 provides a look at how a C program uses the stack to hold the two local variables that we covered earlier in Table 9-1.

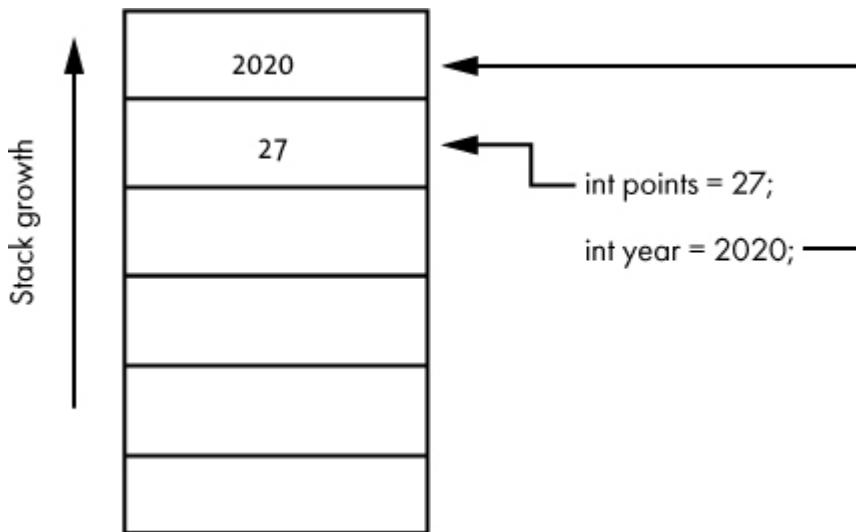


Figure 9-2: Stack memory is used to hold variable values in a program written in C.

In Figure 9-2, the `points` variable is declared first and assigned a value of 27, a value that is stored on the stack. Next, the `year` variable is declared and assigned a value of 2020. This second value is placed “above” the prior value on the stack. Additional values will continue to be added to the top of the stack until they are no longer needed, at which point they will be removed from the stack. Keep in mind that each slot in the diagram is just a location in memory with an assigned memory address, although the addresses aren’t shown in the diagram. You may be surprised to hear that in many architectures, the memory addresses assigned to the stack actually *decrease* as the stack grows. In this example, that means that the `year` variable has a lower memory address than the `points` variable.

Stack memory is fast and well suited for small memory allocations that have limited scope. A separate stack is made available to each thread of execution in a program. We’ll cover threads in more detail in Chapter 10, but for now you can think of threads as parallel tasks within a program. The stack is a limited resource; there’s a limit to how much memory is allocated to the stack. Putting too many values on the stack results in a failure known as a *stack overflow*.

The Heap

The stack is meant to hold small values that only need to be temporarily available. For memory allocations that are large or need to persist for a longer time, the heap is a better fit. The *heap* is a pool of memory that's available to a program. Unlike the stack, heap memory doesn't work on a LIFO model; there is no standard model for how heap memory is allocated. Whereas stack memory is specific to a thread, allocations made from the heap can be accessed by any of the program's threads.

Programs allocate memory from the heap, and that memory usage persists until it's freed by the program or the program terminates. To *free* a memory allocation simply means to release it back to the pool of available memory. Some programming languages automatically free heap memory when an allocation is no longer referenced; one common approach for doing this is known as *garbage collection*. Other programming languages, like C, require the programmer to write code to free heap memory. A *memory leak* occurs when unused memory isn't freed.

In the C programming language, a special kind of variable called a *pointer* is used to track memory allocations. A pointer is simply a variable that holds a memory address. The pointer value (a memory address) can be stored in a local variable on the stack, and that value can refer to a location in the heap, as illustrated in Figure 9-3.

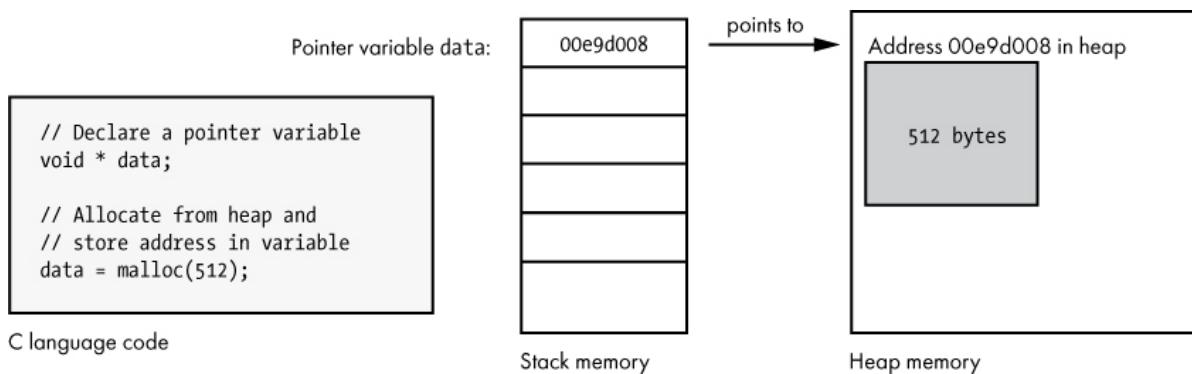


Figure 9-3: The pointer variable named *data* is on the stack and points to an address in the heap.

In Figure 9-3 we have a code snippet that declares a variable called *data*. This variable is of type `void *`, which means it's a pointer (indicated

by *) that points to a memory address that can hold any type of data (`void` means the type isn't specified). Because `data` is a local variable, it is allocated on the stack. The next line of code makes a call to `malloc`, a function in C that allocates memory from heap. The program is asking for 512 bytes of memory, and the `malloc` function returns the address of the first byte of the newly allocated memory. That address is stored in the `data` variable, on the stack. So we end up with a local variable at an address on the stack that holds the address of an allocation on the heap.

NOTE

Please see Project #16 on page 187, where you can see for yourself how variables are allocated in a running program.

Math

Since processors provide instructions for performing mathematical operations, high-level languages do too. In contrast to programming in assembly language, where specific named instructions are required for math (such as the `subs` instruction for subtraction on ARM processors), high-level languages generally include symbols that represent common mathematical operations, making it simple to perform math in code. A large number of programming languages, including C and Python, use the same operators for addition, subtraction, multiplication, and division, as shown in Table 9-2.

Table 9-2: Common Math Operators

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/

Another common convention across multiple programming languages is to use the equals sign (=) to represent an assignment rather

than equality. That is, a statement like `x = 5` means to *set* the value of `x` to 5. Assigning the result of a mathematical operation is represented in a natural way, such as in these statements:

```
// Addition is easy in C.  
cost = price + tax;
```

```
# Addition is easy in Python too.  
cost = price + tax
```

So far, we've focused on integer math, which is common in computing. However, computers and high-level languages also support something called *floating-point arithmetic*. Unlike integers that represent whole numbers, floating-point values can represent fractions. Some programming languages hide the details of this, but internally CPUs use different instructions for floating-point math than for integer math. In C, floating-point variables are declared using a floating-point type such as `float` or `double`, as shown here:

```
// Declaring a floating-point variable in C  
double price = 1.99;
```

On the other hand, Python infers type for its variables, so both integers and floating-point values are declared in the same way:

```
# Declaring integer and floating-point variables in Python  
year = 2020 # year is an int  
price = 1.99 # price is a float
```

The differences in integers versus floating-point numbers can lead to unexpected results sometimes. For example, let's say you have the following C code:

```
// Dividing integers in C  
int x = 5;  
int y = 2;  
int z = x / y;
```

What would you expect the value of `z` to be? It turns out that since all the numbers involved are integers, `z` ends up with a value of 2. Not

`2.5`, but `2`. As an integer, `z` cannot hold fractional values.

Now what if we changed the code slightly, like so:

```
// Dividing integers in C, result stored in a float
int x = 5;
int y = 2;
float z = x / y;
```

Note that `z` now has a type of `float`. Now what would you expect the value of `z` to be? Interestingly, `z` is now equal to `2.0`; it still isn't `2.5`! This is because the division operation occurred with two integers, so the result was an integer too. The division result was `2`, and when it was assigned to floating-point variable `z`, it was given the value `2.0`. The C language is very literal; it's compiled into instructions that closely mirror what the programmer said to do. This is great for programmers who need fine-grained control of processing, but not always so great for programmers who expect more intuitive behavior from their programming language.

Python tries to be more helpful, automatically assigning a type that allows for fractional results in a situation like this. If we write an equivalent version of this code in Python, the result stored in `z` will be `2.5`.

```
# Dividing integers in Python
# z will be 2.5 and its inferred type is float
x = 5
y = 2
z = x / y
```

Some languages provide mathematical operators that are abbreviated ways of stating an operation. For example, C provides increment (add 1) and decrement (subtract 1) operators, as shown here:

```
// In C, we can add one to a variable the long way,
x = x + 1;
// or we can use this shortcut to increment x.
x++;
// On the other hand, this will decrement x.
x--;
```

NOTE

Fun fact: the name of the programming language C++ is meant to convey the idea that it's an improvement upon, or an increment of, the C programming language.

Python also provides some shortcut operators for math. The `+=` and `-=` operators allow programmers to add to or subtract from a variable. For example:

```
# In Python, we can add 3 to a variable like this...
cats = cats + 3

# Or we can do the same thing with this shortcut...
cats += 3
```

The `+=` and `-=` operators work in C as well.

Logic

As we covered earlier, processors are very good at performing logical operations, since logic is the foundation for digital circuits. As you'd expect, programming languages also provide the capability to handle logic. Most high-level languages provide two kinds of operators that deal in logic: bitwise operators, which deal with the bits of integers, and Boolean operators, which deal with Boolean (true/false) values. The terminology here can be confusing, since different programming languages use different terms. Python uses "bitwise" and "Boolean," whereas C uses "bitwise" and "logical," and other languages use still other terms. Let's stick with "bitwise" and "Boolean" here.

Bitwise Operators

Bitwise operators act on the individual bits of integer values and result in an integer value. A bitwise operator is like a mathematical operator, but instead of adding or subtracting, it performs an AND, OR, or other logical operation on the bits of integers. These operators work according to the truth tables covered in Chapter 2, performing the operation on all the bits in the integer in parallel.

Many programming languages, including C and Python, use the set of operators shown in Table 9-3 for bitwise operations.

Table 9-3: Bitwise Operations as Commonly Expressed in Programming Languages

Bitwise operation	Bitwise operator
AND	&
OR	
XOR	^
NOT (complement)	~

Let's look at a bitwise example in Python.

```
# Python does bitwise logic.  
x = 5  
y = 3  
a = x & y  
b = x | y
```

The result of the code above is that `a` is 1 and `b` is 7. Let's look at those operations in binary (Figure 9-4) to make it clear why this is so.

$x = 5 = 0101$ $y = 3 = 0011$ <hr/> 0001	$x = 5 = 0101$ $y = 3 = 0011$ <hr/> 0111
AND	OR

Figure 9-4: Bitwise AND, OR operations on 5 and 3

Take a look at the AND operation in Figure 9-4 first; recall from Chapter 2 that AND means that the result is 1 when both inputs are 1. Here we look at the bits one column at a time, and as you can see, only the rightmost bit is 1 for both inputs. Therefore, the AND result is 0001 binary, or 1 decimal. Therefore, `a` is assigned a value of 1 in the preceding code.

On the other hand, OR means that the result is 1 if either input (or both inputs) is 1. In this example, the rightmost three bits are all 1 in one input or the other, so the result is 0111 binary, or 7 decimal. Therefore, `b` is assigned a value of 7 in the preceding code.

EXERCISE 9-1: BITWISE OPERATORS

Consider the following Python statements. What will be the values of `a`, `b`, and `c` after this code executes?

```
x = 11
y = 5
a = x & y
b = x | y
c = x ^ y
```

The answer can be found in Appendix A.

Boolean Operators

The other kind of logical operator in high-level programming languages is the *Boolean operator*. These operators work on Boolean values and result in a Boolean value.

Let's take a moment to talk about Boolean values. A *Boolean value* is either true or false. Different programming languages represent true or false in different ways. A *Boolean variable* is a named memory address that holds a Boolean value of true or false. For example, we could have a variable in Python that tracks whether an item is on sale: `item_on_sale = True`.

An expression can evaluate to true or false without the result being stored in a variable. For example, the expression `item_cost > 5` evaluates to either true or false at runtime depending on the value of the `item_cost` variable.

Boolean operators allow us to perform a logical operation like AND, OR, or NOT on Boolean values. For example, we can check if two conditions are both true using Python's Boolean AND operator: `item_on_sale and item_cost > 5`. The expressions to the left and right of `and` evaluate to Boolean values, and in turn, the entire expression evaluates to a Boolean value. Here C and Python use different operators, as shown in Table 9-4.

Table 9-4: Boolean Operators in C and Python Programming Languages

Boolean operation	C operator	Python operator
AND	&&	and
OR		or
NOT	!	not

While we’re on the subject of operators that return Boolean values, a *comparison operator* compares two values and evaluates to true or false as the result of the comparison. For example, the *greater than operator* allows us to compare two numbers and determine if one is larger than the other. Table 9-5 shows comparison operators used in both C and Python.

Table 9-5: Comparison Operators in C and Python Programming Languages

Comparison operation	Comparison operator
EQUALITY	==
NOT EQUAL	!=
GREATER THAN	>
LESS THAN	<
GREATER THAN OR EQUAL	>=
LESS THAN OR EQUAL	<=

You’ve already seen one of these in use, in our earlier example of `item_cost > 5`. Pay attention to the equality operator. Both C and Python use a double equals sign to represent an equality comparison, and they use a single equals sign to represent an assignment. That means `x == 5` is a comparison that returns true or false (is `x` equal to 5?), whereas `x = 5` is an assignment that sets the value of `x` to 5.

Program Flow

Boolean and comparison operators allow us to evaluate the truth of an expression, but that alone isn’t very useful. We need a way to do something in response! *Program flow*, or *control flow*, statements allow us

to do just that, altering the behavior of a program in response to some condition. Let's look at some common program flow constructs found across programming languages.

If Statements

An *if statement*, often coupled with an *else statement*, allows the programmer to do something if some condition is true. In turn, the *else statement* allows the program to do something different if the condition is false. Here's an example in Python:

```
# Age check in Python
❶ if age < 18:
    ❷ print('You are a youngster!')
❸ else:
    ❹ print('You are an adult.')
```

In this example, the first `if` statement ❶ checks if the `age` variable refers to a value that's less than 18. If so, it prints a message indicating that the user is young ❷. The `else` statement ❸ tells the program to print a different message if `age` is 18 or greater ❹.

Here is the same “age check” logic, this time written in C:

```
// Age check in C
if (age < 18)
❶ {
    printf("You are a youngster!");
❷ }
else
{
    printf("You are an adult.");
}
```

In the C example, note the curly braces used after the `if` statement ❶❷. These mark off a block of code that should execute in response to the `if`. In C, a code block can consist of multiple lines of code, although the braces can be omitted when the block consists of a single line. Python doesn't use braces to delimit a block of code; it uses indentation

instead. In Python, contiguous lines at the same level of indentation (say, four spaces) are considered part of the same block.

Python also includes an `elif` statement, which means “else if.” An `elif` statement is only evaluated if the preceding `if` or `elif` statement was false.

```
# A better age check in Python
if age < 13:
    print('You are a youngster!')
elif age < 20:
    print('You are a teenager.')
else:
    print('You are older than a teen.')
```

The same thing can be accomplished in C by using an `else` coupled with an `if`:

```
// A better age check in C
if (age < 13)
    printf("You are a youngster!");
else if (age < 20)
    printf("You are a teenager!");
else
    printf("You are older than a teen.");
```

Note that I’ve also omitted the curly braces since all my code blocks are single lines.

Looping

Sometimes a program needs to perform a certain action over and over. A *while loop* allows code to run repeatedly until some condition is met. In the following Python example, a `while` loop is used to print the numbers from 1 to 20.

```
# Count to 20 in Python.
n = 1
while n <= 20:
    print(n)
    n = n + 1
```

Initially, the variable `n` is set to 1. The `while` loop begins, indicating that the loop should run while `n` is less than or equal to 20. Since `n` is 1, it

meets that requirement, so the body of the `while` loop runs, printing the value of `n` and adding 1 to it. Now `n` is equal to 2, and the code goes back to the top of the `while` loop. This process continues until `n` is equal to 21, at which point it no longer meets the requirements of the `while` loop, so the loop ends.

The following is the same thing implemented in C.

```
// Count to 20 in C.  
int n = 1;  
while(n <= 20)  
{  
    printf("%d\n", n);  
    n++;  
}
```

In both examples, the body of the `while` loop increments the value of `n`. There's actually a cleaner way to do this. A *for loop* allows iteration over a range of numbers or a collection of values so that the programmer can perform some operation on each. Here we have an example in C that prints 1 through 10.

```
// C uses a for loop to iterate over a numeric range.  
// This will print 1 through 10.  
for(❶int x = 1; ❷x <= 10; ❸x++)  
{  
❹    printf("%d\n", x);  
}
```

The `for` loop declares `x` and sets its initial value to 1 ❶, states that the loop will continue while `x` is less than or equal to 10 ❷, and finally declares that `x` should be incremented after the body of the loop runs ❸. By putting all of this information in a `for` statement on a single line, we can more easily see the conditions under which the loop will run. The body of the `for` loop simply prints the value of `x` ❹.

Python takes a different approach with its `for` loops, allowing the program to take an action repeatedly on every item in a collection of values. The following Python example prints out the names of animals in a list.

```
# Python uses a for loop to iterate over a collection.  
# This will print each animal name in animal_list.  
animal_list = ['cat', 'dog', 'mouse']  
for animal in animal_list:  
    print(animal)
```

First, a list of animal names is declared and assigned to a variable named `animal_list`. In Python, a list is an ordered collection of values. Next, the `for` loop states that the code block runs once for each item in `animal_list`, and each time the code runs, the current value in the list is assigned to the `animal` variable. So the first time the body of the loop runs, `animal` is equal to `cat`, and the program prints `cat`. The next time through `dog` prints, and the final time `mouse` prints.

Functions

Looping allows a set of instructions to run multiple times in a row. However, it's also common for a program to run a particular set of instructions multiple times, but not necessarily in a loop. Instead, such instructions may need to be invoked from different parts of the program, at different times, and with varying inputs and outputs. When a programmer realizes that the same code is needed in multiple places, they may write that code as a function. A *function* is a set of program instructions that may be invoked, or called, by other code. Functions optionally take inputs (known as *parameters*) and return an output (known as a *return value*). Different high-level languages use different terms for a function, including *subroutine*, *procedure*, or *method*. In some cases, these various names actually convey slightly different meanings, but for our purposes, let's just stick with function.

Converting a character string to lowercase, printing text to the screen, and downloading a file from the internet are all examples of what you can do with reusable code in the form of a function. Programmers want to avoid typing out the same code multiple times. Doing so means maintaining several copies of the same code and increasing the overall size of a program. This violates a software

engineering principle known as *don't repeat yourself (DRY)* that encourages a reduction of duplicative code.

Functions are another example of encapsulation. We saw encapsulation earlier in the context of hardware, and here we see it again, this time in software. Functions encapsulate the internal details of a block of code while providing an interface for making use of that code. A developer who wants to use a function only needs to understand its inputs and outputs; a full understanding of the function's internal workings isn't needed.

Defining Functions

A function must be defined before it can be used. Once defined, you use a function by calling it. A *function definition* includes the name of the function, the input parameters, the program statements for the function (called the *body* of the function), and in some languages, the return value type. Here we have a sample C function that calculates the area of a circle, given its radius.

```
// C function to calculate the area of a circle
❶ double ❷areaOfCircle(❸double radius)
{
    double area = 3.14 * radius * radius;
    ❹ return area;
}
```

The `double` type at the beginning ❶ indicates that the function returns a floating-point number (`double` is one of the floating-point types in C). The function has a name, `areaOfCircle` ❷, meant to convey what the function does—in this case, calculate the area of a circle. The function takes one input parameter named `radius` ❸, also of type `double`.

Between the opening and closing curly braces we have the body of the function, which defines exactly how the function works. We declare a local variable named `area`. It's also of type `double`. The area is calculated as $\pi \times \text{radius}^2$ and assigned to the `area` variable. Finally, the function returns the value of the `area` variable ❹. Note that the `area` variable's

scope is limited; it cannot be accessed outside of this function. When the function returns, the local variable `area` is discarded (it was probably stored on the stack), but its value is returned to the caller, likely via a processor register.

The following is a similar area function, this time written in Python.

```
# Python function to calculate the area of a circle
def area_of_circle(radius)
    area = 3.14 * radius * radius
    return area
```

Let's compare the two function examples. Both calculate the `area` as $\pi \times \text{radius}^2$ and then return that value. Both take one input parameter named `radius`. The C version explicitly defines the return type as `double` and the type of `radius` as `double`, while the Python version doesn't require the types to be declared. Python indicates that a function definition is about to begin with the `def` keyword.

Calling Functions

Defining a function in a program isn't enough to ensure that function will run. A function definition simply makes the code available for other code to invoke when needed. Such an invocation is known as a *function call*. The calling code passes any needed parameters and hands control over to the function. The function then executes its code and returns control (and any output) back to the caller. The following demonstrates calling our example function in C:

```
// Calling a function twice in C, each time with a different input
double area1 = areaOfCircle(2.0);
double area2 = areaOfCircle(38.6);
```

and in Python:

```
# Calling a function twice in Python
area1 = area_of_circle(2.0)
area2 = area_of_circle(38.6)
```

Once the function returns, it's up to the calling code to store the returned value somewhere. Variables `area1` and `area2` are declared in both examples to hold the return values from the function calls. In both languages, `area1` is 12.56 and `area2` is 4,678.4744. Actually, the calling code can just ignore the returned value and not assign it to a variable, but that is not very useful, considering this function's purpose. Figure 9-5 illustrates how calling a function temporarily cedes control to that function.

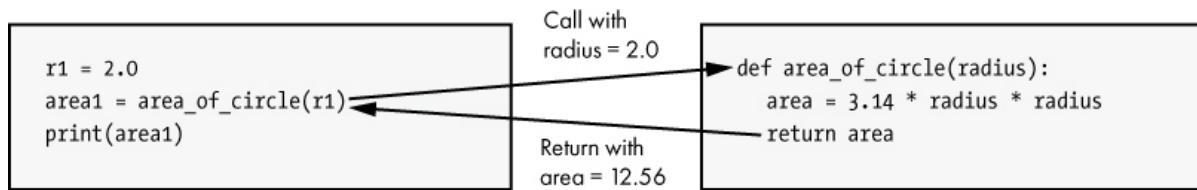


Figure 9-5: Calling a function

In Figure 9-5, the Python code on the left calls the `area_of_circle` function, passing it an input `radius` parameter value of `2.0`. The code on the left then waits until the function on the right completes its work. Once the function returns, the code on the left stores the returned value in variable `area1`, and it then resumes execution.

Using Libraries

Although programmers do define functions for their own use, an important part of programming is knowing how to best leverage functions that other people have already written. Programming languages usually include a large set of functions known as the *standard library* for that language. In this context, a *library* is a collection of code intended to be used by other software. Both C and Python include standard libraries that provide functions for things such as printing to the console, working with files, and text processing. Python's standard library is particularly extensive and well-regarded. Although not always the case, most implementations of a language include that language's standard library, so programmers can rely on those functions.

NOTE

Please see Project #17 on page 189, where you can use what you've learned to write a simple guessing game in Python. This includes using the Python standard library.

Outside of the standard library, additional libraries of functions are also available for many programming languages. Developers write libraries for others to use and share them in the form of source code or as compiled files. These libraries are sometimes shared informally, and certain programming languages have a well-known, accepted mechanism for publishing libraries. A shared set of libraries is known as a *package*, and a system for sharing such packages is known as a *package manager*. Several package managers are available for C, but none of them is universally accepted as a standard by C programmers. Python's included package manager is called `pip`. `pip` makes it easy to install community-developed software libraries for Python, and it's commonly used by Python developers.

Object-Oriented Programming

Programming languages are designed to support specific *paradigms*, or approaches, to programming. Examples include procedural programming, functional programming, and object-oriented programming. A language may be designed to support one or multiple paradigms, and it is up to the software developer to use the language in a way that fits a certain paradigm. Let's take a look at one popular paradigm: *object-oriented programming*, an approach to programming in which code and data are grouped together in a construct known as an *object*. Objects are meant to represent a logical grouping of data and functionality in a way that models real-world concepts.

Object-oriented programming languages commonly use a class-based approach. A *class* is a blueprint for an object. An object created from a class is said to be an *instance* of that class. Functions defined in a class are known as *methods*, and variables declared in a class are known as *fields*. In Python, fields that have different values for each instance of a

class are called *instance variables*, whereas fields that have the same value across all instances of the class are called *class variables*.

For example, a class could be written that describes a bank account. The bank account class might have a field for the balance, a field for the holder's name, and methods for withdrawing and depositing money. The class describes a generic bank account, but no specific instance of a bank account exists until a bank account object is created from that class. This is illustrated in Figure 9-6.

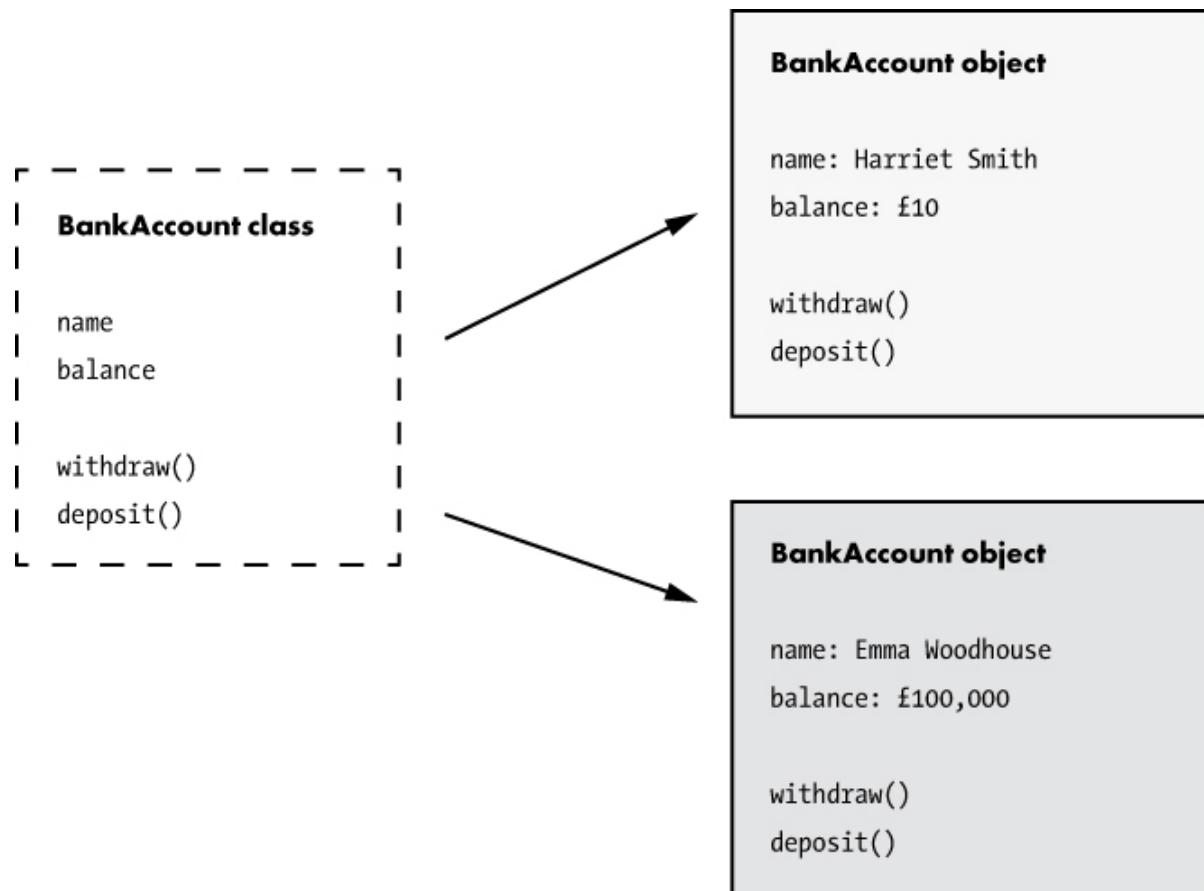


Figure 9-6: Bank account objects are created from a bank account class.

As you can see in Figure 9-6, the `BankAccount` class describes the fields and methods of the bank account, providing us with an understanding of what a bank account is like. Two objects, instances of the `BankAccount` class, have been created. These objects are specific bank accounts, with names and balances assigned. We can use each object's `withdraw` or `deposit`

method to modify its `balance` field. In Python, depositing into a bank account object named `myAccount` would look like this, resulting in an increase of 25 to its `balance` field:

```
myAccount.deposit(25)
```

NOTE

Please see Project #18 on page 190, where you can try a Python implementation of the bank account class just described.

Compiled or Interpreted

As mentioned earlier, source code is the text of a program as originally written by developers, and it usually isn't written in a programming language that CPUs understand directly. CPUs only understand machine language, so additional steps are required: source code must be either compiled to machine code or interpreted by other code at runtime.

In a *compiled language*, like C, source code is converted into machine instructions that can be directly executed by a processor. That process was described earlier in this chapter in “High-Level Programming Overview” on page 160. Source code is compiled during the development process, and the compiled executable files (sometimes called *binaries*) are delivered to end users. When end users run binaries, they don't need access to the source code. Compiled code tends to be fast, but it only runs on the architecture for which it was compiled. Figure 9-7 shows an example of how a developer would compile and run a C program from a command line using the GNU C Compiler (`gcc`).

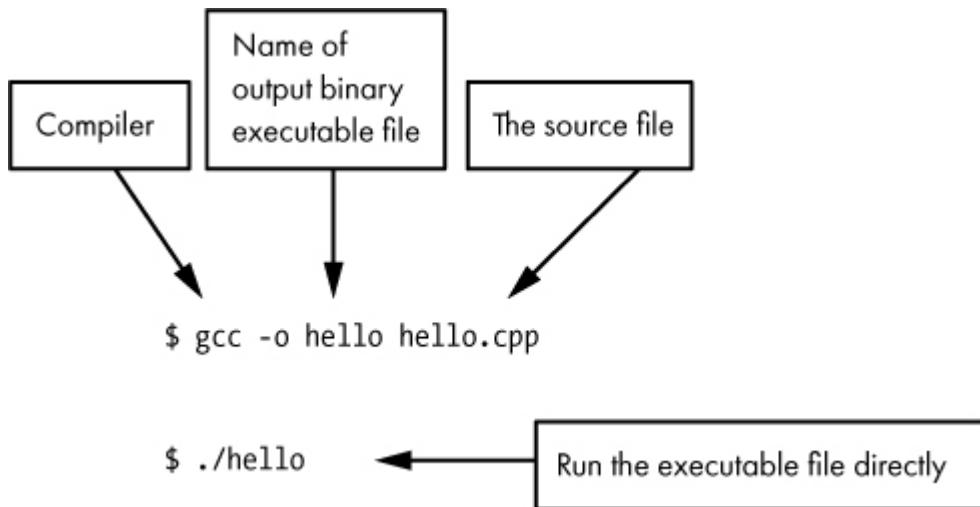


Figure 9-7: Compiling a C source file to an executable file that can run on its own

In an *interpreted language*, like Python, the source code is not compiled ahead of time. Instead, it is read by a program called an *interpreter* that reads and executes the program's instructions. It's the interpreter's machine code that actually runs on the CPU. Developers of code in interpreted languages can distribute their source code and end users can run it directly, without the need for a potentially complex compilation step. In this scenario, the developers don't need to worry about compiling their code for lots of different platforms—as long as the user has the appropriate interpreter on their system, they can run the code. In this way, the distributed code is platform-independent.

Interpreted code tends to run more slowly than compiled code due to the overhead of interpreting the code as it runs. Distributing interpreted code works best when the user already has the required interpreter installed or the user is technically proficient enough that installing an interpreter isn't a barrier. Otherwise, the developer needs to either bundle the interpreter with their software or guide the user through installing the interpreter. Figure 9-8 shows an example of running a Python program from a command line, assuming the Python version 3 interpreter is already installed. Note how the Python source code in *hello.py* is given directly to the interpreter—no intermediate step required.

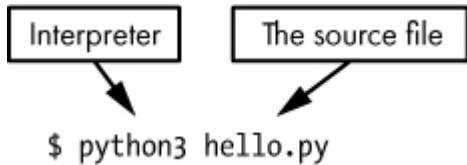


Figure 9-8: The Python interpreter runs Python source code.

Some languages use a system that's a hybrid of these two approaches. Such languages compile to an *intermediate language*, or *bytecode*. Bytecode is similar to machine code, but rather than targeting a specific hardware architecture, bytecode is designed to run on a virtual machine, as illustrated in Figure 9-9.

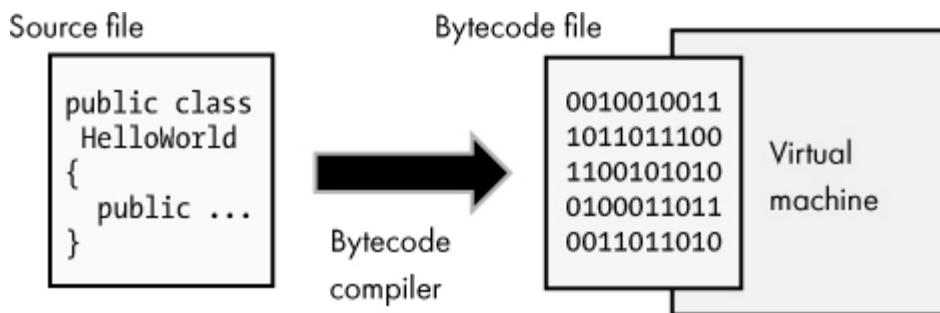


Figure 9-9: A bytecode compiler turns source code into bytecode that runs inside a virtual machine.

In this context, a *virtual machine* is a software platform designed to run other software. The virtual machine provides a virtual CPU and execution environment, abstracting the details of the real underlying hardware and operating system. For example, Java source code is compiled to Java bytecode, which then runs within the Java virtual machine. Similarly, C# source code is compiled to Common Intermediate Language and runs in the .NET Common Language Runtime (CLR) virtual machine. CPython, the original implementation of Python, actually converts Python source code to bytecode before running it, although this is an implementation detail of the CPython interpreter and mostly hidden from Python developers. Programming languages that use bytecode retain the platform-independence of interpreted languages while preserving some of the performance gains of compiled code.

Calculating a Factorial in C

To wrap up our look at high-level programming, let's now examine an implementation of the factorial algorithm, this time in the C language. We did this before in ARM assembly, so seeing the same logic in C should serve as a good comparison between assembly language and a high-level language. This C code uses several of the concepts we just covered. I chose to use C rather than Python because it's a compiled language, and we can examine the compiled machine code. Here's a simple C function that calculates the factorial of a number:

```
// Calculate the factorial of n.
int factorial(int n)
{
    int result = n;

    while(--n > 0)
    {
        result = result * n;
    }

    return result;
}
```

Other code can call this function, passing the `n` parameter as the value whose factorial should be calculated. The function then internally calculates the factorial value, storing it in the local variable `result` and returning the calculated value to the caller. As we did with the assembly code in Chapter 8, let's again use an exercise to explore this code in depth.

EXERCISE 9-2: RUN A C PROGRAM IN YOUR MIND

Try running the preceding factorial function in your head or use pencil and paper. Assume an input value of `n = 4`. When the function returns, the returned result should be the expected value of 24. I recommend that for each line, you keep track of the values of `n` and `result` before and after the statement completes. Work through the code until you reach the end of the `while` loop and see if you get the expected result. The answer is in Appendix A.

Note that the condition of the `while` loop (`--n > 0`) places the decrement operator (`--`) before the variable `n`. This means that `n` is decremented *before* its value is compared to

0. This happens each time the `while` loop condition is evaluated.

I hope that you find the C version of our algorithm more readable than the ARM assembly version! The other major advantage of this version of our factorial code is that it isn't tied to a specific processor type. It could be compiled for any processor, given an appropriate compiler. If you compile the earlier C code for an ARM processor, you see machine code generated that's similar to the ARM assembly we examined earlier. You'll get a chance to do that in Project #19, but for now I've compiled and disassembled the code for you:

Address	Assembly
0001051c	sub r3, r0, #1
00010520	cmp r3, #0
00010524	bxle lr
00010528	mul r0, r3, r0
0001052c	subs r3, r3, #1
00010530	bne 00010528
00010534	bx lr

As you can see, the code generated from the C source is quite similar to the assembly factorial example we covered in Chapter 8. There are some differences, but the specifics aren't relevant to our discussion. The thing to note here is that a program can be written in a high-level language like C, and a compiler can do the hard work of translating the high-level statements to machine code. You can see how working in a high-level language can simplify things for a developer, but in the end, we still end up with bytes of machine code, because that's what a processor needs.

NOTE

Please see Project #19 on page 191, where you can try compiling and then disassembling a factorial program in C.

Something interesting happened here, and I want to make sure you didn't miss it. We started with source code written in the C programming language, compiled it into machine code, and then disassembled it into assembly language. The implication of this is that if

you have a compiled program or software library on your computer, you can examine its code as assembly language! You may not have access to the original source code, but the assembly version of the program is within your grasp.

We've been looking at machine code and assembly language for the ARM processor specifically, but as mentioned earlier, one of the advantages of developing in a high-level language like C is that the same code can be compiled for a different processor. In fact, the same code can even be compiled for another operating system, as long as the code in question doesn't use functionality that's specific to a particular operating system. To illustrate this point, I've compiled the same factorial C code for a 32-bit x86 processor, this time on Windows rather than Linux. Here's the generated machine code, shown as assembly language:

Address	Assembly
00406c35	mov ecx,dword ptr [esp+4]
00406c39	mov eax,ecx
00406c3b	jmp 00406c40
00406c3d	imul eax,ecx
00406c40	dec ecx
00406c41	test ecx,ecx
00406c43	jg 00406c3d
00406c45	ret

I won't elaborate on the details of this code, but feel free to research the x86 instruction set and interpret the code yourself. The main thing I hope you take away from this example is that high-level languages, like C, allow developers to write code that's easier to understand than assembly and that can be easily compiled for various processors.

Summary

In this chapter we covered high-level programming languages. Such languages are independent from a specific CPU and syntactically closer to human language. You learned about common elements found across programming languages, such as comments, variables, functions, and

looping capabilities. You saw how these elements are expressed in two programming languages: C and Python. Finally, we examined an example program in C, and you saw the disassembled machine code generated by compiling high-level code.

In the next chapter, we'll cover operating systems. We'll start with an overview of the capabilities provided by operating systems, learn about the various families of operating systems, and dive deeper into how operating systems work. Along the way you'll have the opportunity to explore Raspberry Pi OS, a version of Linux tailored for the Raspberry Pi.

PROJECT #14: EXAMINE VARIABLES

Prerequisite: A Raspberry Pi, running Raspberry Pi OS. I recommend that you flip to Appendix B and read the entire “Raspberry Pi” section on page 341 if you haven’t already.

In this project, you’ll write high-level code that uses variables and examine how this works in memory. Use the text editor of your choice to create a new file called *vars.c* in the root of your home folder. Enter the following C code into your text editor (you don’t have to preserve indentation and empty lines, but be sure to maintain line breaks).

```
#include <stdio.h>❶
#include <signal.h>

int main()❷
{
    int points = 27;❸
    int year = 2020;❹

    printf("points is %d and is stored at 0x%08x\n", points, &points);❺
    printf("year is %d and is stored at 0x%08x\n", year, &year);

    raise(SIGINT);❻

    return 0;
}
```

Before continuing, let’s examine the source code. It begins by including a couple of header files ❶. These files include details required by the C compiler about the `printf` and `raise` functions that are used later in the program. Next you see the `main` function

defined ❷; this is the entry point of the program where execution begins. The program then declares two integer variables, `points` ❸ and `year` ❹, and assigns values to them. It then prints out both the values of the variables and their memory addresses (in hexadecimal) ❺. The `raise(SIGINT)` statement causes the program to halt execution ❻. This is not something you normally do in code that end users run; it's a technique we use here to assist with debugging.

Once the file is saved, use the GNU C Compiler (`gcc`) to compile your code into an executable file. Open a terminal on your Raspberry Pi and enter the following command to invoke the compiler. This command takes `vars.c` as an input, compiles and links the code, and outputs an executable file named `vars`.

```
$ gcc -o vars vars.c
```

Now try running the compiled code using the following command. The program should print out the values and addresses of the program's two variables.

```
$ ./vars
```

Once you've confirmed that the program works, run it under the GNU Debugger (`gdb`) and examine the variables in memory.

```
$ gdb vars
```

At this point `gdb` has loaded the file but no instructions have run yet. From the (`gdb`) prompt, type the following to run the program, which will continue until the `raise(SIGINT)` statement is executed.

```
(gdb) run
```

Once the program returns to a (`gdb`) prompt, you should see a couple of lines where the values and memory addresses of the variables were printed. Following those lines, you may also see a potentially worrisome statement about “no such file or directory”—you can ignore it. It's just the debugger trying to find some source code that isn't on your system. The output you do need to pay attention to should look something like this:

```
Starting program: /home/pi/vars
points is 27 and is stored at 0x7efff1d4
year is 2020 and is stored at 0x7efff1d0
```

Now you know the memory addresses, and since you're conveniently in the debugger, you're ready to examine what's stored at those addresses. In this output, you can see that `year` is stored at the lower address, and `points` is stored 4 bytes later, so you'll dump out memory starting at the address of the `year` variable, `0x7efff1d0` in my case. Your address may be different. The following command dumps out three 32-bit values in memory, in

hexadecimal, starting at address `0x7efff1d0`. Replace `0x7efff1d0` with the address of `year` on your system if they differ.

```
(gdb) x/3xw 0x7efff1d0  
0x7efff1d0: 0x000007e4 0x0000001b 0x00000000
```

You can see here that the value stored at `0x7efff1d0` is `0x000007e4`. That's 2020 in decimal, the expected `year` value. And the value stored 4 bytes later is `0x0000001b`, or 27 decimal, the expected `points` value. The next value in memory happens to be 0 and isn't one of our variables. Memory is usually examined in hexadecimal, but if you want to see these values in decimal, you can use the following command instead:

```
(gdb) x/3dw 0x7efff1d0  
0x7efff1d0: 2020 27 0
```

You're looking at memory in 32-bit (4-byte) chunks, since that's the size of the variables used in this program. But memory is actually byte addressable, meaning each byte has its own address. That's why `points` has an address 4 bytes greater than the address of `year`. Let's look at the same memory range as a series of bytes instead:

```
(gdb) x/12xb 0x7efff1d0  
0x7efff1d0: 0xe4 0x07 0x00 0x00 0x1b 0x00 0x00 0x00 0x0  
0  
0x7efff1d8: 0x00 0x00 0x00 0x00
```

Look at the value for `year`, emphasized here. Note how the least significant byte (`0xe4`) comes first. This is due to little-endian data storage, as discussed on page 156 in Project #13. You can exit `gdb` with `q` (it will ask you if you want to quit even though a debugging session is active; answer `y`).

PROJECT #15: CHANGE THE TYPE OF VALUE REFERENCED

BY A VARIABLE IN PYTHON

Prerequisite: A Raspberry Pi, running Raspberry Pi OS. I recommend that you flip to Appendix B and read the entire “Raspberry Pi” section on page 341 if you haven’t already.

In this project, you'll write code that sets a Python variable to a value of a certain type and then updates that variable to reference a value of a different type. Use the text editor of your choice to create a new file called `vartype.py` in the root of your home folder. Enter the following Python code into your text editor:

```
age = 22
print('What is the type?')
print(type(age))

age = 'twenty-two'
print('Now what is the type?')
print(type(age))
```

This code sets the variable named `age` to an integer value and then prints the type of that value. It then sets `age` to a string value and prints the type again.

Once the file is saved, you can run the file from a terminal window using the Python interpreter, like so:

```
$ python3 vartype.py
```

You should see output like the following:

```
What is the type?
<class 'int'>
Now what is the type?
<class 'str'>
```

You can see how the type changes from an integer to a string by simply assigning the variable a new value. Don't let the term `class` confuse you; in Python 3, built-in types such as `int` and `str` are considered classes (covered in “Object-Oriented Programming” on page 177). Setting a variable to a value of a different type is easy in Python but not allowed at all in C.

VERSIONS OF PYTHON

Two major versions of Python are in use today, Python 2 and Python 3. As of January 1, 2020, Python 2 is no longer supported, meaning no new bug fixes will be made to it. Python developers are encouraged to migrate old projects to Python 3, and new projects should target Python 3. Accordingly, the projects in this book use Python 3. On Raspberry Pi OS and some other Linux distributions, running `python` from a command line will invoke the Python 2 interpreter, while running `python3` will invoke the Python 3 interpreter. That's why the projects in this book have you specifically run `python3` rather than `python`. That said, on other platforms, or even on future versions of Raspberry Pi OS, this may not hold true, and entering `python` may actually invoke Python 3. You can check the invoked version of Python like so:

```
$ python --version
```

or

```
$ python3 --version
```

PROJECT #16: STACK OR HEAP

Prerequisite: Project #14.

In this project, you'll look at whether variables are allocated in stack or heap memory in a running program. Open a terminal on your Raspberry Pi, and begin by debugging the `vars` program you previously compiled in Project #14:

```
$ gdb vars
```

At this point `gdb` has loaded the file but no instructions have run yet. From the `gdb` prompt, type the following to run the program, which continues until the `SIGINT` statement is executed.

```
(gdb) run
```

Again, look at the memory addresses of the `points` and `year` variables. In my case, these variables were found at `0x7efff1d4` and `0x7efff1d0`, but your addresses may vary. Now use the following command to see all the mapped memory locations for your running program:

```
(gdb) info proc mappings
```

The output lists the start and end address of the various memory ranges in use by this program. Find the one that includes the addresses of your variables. Both variable addresses should fall within a single range. For me, this entry matched:

```
0x7efdf000 0x7f000000 0x21000 0x0 [stack]
```

As you can see, `gdb` indicates that this memory range is allocated for the stack, which is exactly where we would expect local variables to be. You can exit `gdb` with `q` (it will ask you if you want to quit even though a debugging session is active; answer `y`).

Let's now look at memory allocated on the heap. You need to modify `vars.c` and rebuild it so that the program allocates some heap memory. Use the text editor of your choice to open the existing `vars.c` file. Add the following line of code as the very first line:

```
#include <stdlib.h>
```

Then add these two lines immediately before the `SIGINT` line:

```
void * data = malloc(512);
printf("data is 0x%08x and is stored at 0x%08x\n", data, &data);
```

Let's cover what these changes mean. We call the memory allocation function `malloc` to allocate 512 bytes of memory from the heap. The `malloc` function returns the address of the newly allocated memory. That address is stored in a new local variable called `data`. The program then prints two memory addresses: the address of the new heap allocation and the address of the `data` variable itself, which should be on the stack.

Once the file is saved, use `gcc` to compile your code:

```
$ gcc -o vars vars.c
```

Now run the program again:

```
$ gdb vars
(gdb) run
```

Check the newly printed values. For me, the values are as follows:

```
data is 0x00022410 and is stored at 0x7efff1ac
```

We expect that the first address, the address that came back from `malloc`, to be on the heap. The second value, the address of the `data` local variable, should be on the stack. Again, run the following to see this program's memory ranges and see where these two addresses fall.

```
(gdb) info proc mappings
...
      0x22000    0x43000    0x21000      0x0 [heap]
...
0x7efdf000 0x7f000000    0x21000      0x0 [stack]
```

Find the matching address ranges on your system and confirm that the addresses fall in the expected ranges of heap and stack. You can exit `gdb` with `q`.

PROJECT #17: WRITE A GUESSING GAME

In this project, you'll write a guessing game in Python, building on what we've covered in this chapter. Use the text editor of your choice to create a new file named `guess.py` in the root of your home folder. Enter the following Python code into your text editor. In Python, indentation matters, so make sure you indent appropriately.

```
from random import randint❶

secret = randint(1, 10)❷
guess = 0❸
count = 0❹

print('Guess the secret number between 1 and 10')

while guess != secret:❺
    guess = int(input())❻
    count += 1

    if guess == secret:❼
        print('You got it! Nice job.')
    elif guess < secret:
        print('Too low. Try again.')
    else:
        print('Too high. Try again.')

print('You guessed {0} times.'.format(count))❽
```

Let's examine how this program works. This code starts by importing a function called `randint` that generates random integers ❶. This is an example of using a function that was written by someone else; `randint` is a part of the Python standard library. This call to the `randint` function returns a random integer in the range of 1 to 10, which we've stored in a variable named `secret` ❷. The code then sets a variable called `guess` to 0 ❸. This variable holds the player's guess, and it's assigned an initial value of 0, a value we can be sure won't match the `secret` value. A third variable named `count` ❹ keeps track of the number of times that the player guessed so far.

The `while` loop runs as long as the player's `guess` doesn't match the `secret` ❺. The code within the loop calls the built-in function `input` to get the user's guess from the console ❻, and the result is converted to an integer and stored in the `guess` variable. Each time a guess is entered, it's checked against the `secret` variable to see whether it's a match, too low, or too high ❼. Once the player's `guess` matches the `secret`, the loop exits, and the program prints the number of times that the player guessed ❽.

Once the file is saved, you can run it using the Python interpreter like so:

```
$ python3 guess.py
```

Try out the program several times; the secret number should change each time you run it. You may want to try modifying the program so that the range of allowed integers is larger, or maybe you want to put in your own custom messages. As a challenge, try modifying the program so that when the guess is really close, the program prints a different message.

PROJECT #18: USE A BANK ACCOUNT CLASS IN PYTHON

In this project, you'll write a bank account class in Python and then create an object based on that class. Use the text editor of your choice to create a new file named *bank.py* in the root of your home folder. Enter the following Python code into your text editor. You can skip typing in the comments (lines that begin with `#`) if you prefer. Note that `__init__` has two underscore characters at its beginning and end.

```
# Define a bank account class in Python.
class BankAccount:①
    def __init__(self, balance, name):②
        self.balance = balance③
        self.name = name④

    def withdraw(self, amount):⑤
        self.balance = self.balance - amount

    def deposit(self, amount):⑥
        self.balance = self.balance + amount

# Create a bank account object based on the class.
smithAccount = BankAccount(10.0, 'Harriet Smith')⑦

# Deposit some additional money to the account.
smithAccount.deposit(5.25)⑧

# Print the account balance.
print(smithAccount.balance)⑨
```

This code defines a new class called `BankAccount` **①**. Its `__init__` function **②** is automatically invoked when an instance of the class is created. This function sets instance variables `balance` **③** and `name` **④** to the values passed into the initializer function. The variables are unique to each object instance of the class that's created. The class definition also includes two methods: `withdraw` **⑤** and `deposit` **⑥**, which simply modify the balance. After the class is defined, the code proceeds to create an instance of the class **⑦**. This bank account object can now be used by accessing its variables and methods. Here a deposit is made **⑧**, followed by a retrieval of the new balance, which is printed **⑨**.

Once the file is saved, you can run it using the Python interpreter like so:

```
$ python3 bank.py
```

You should see the account balance of 15.25 print to the terminal window. In truth, this was an overly complicated way of calculating this bank balance! The numbers are all

hard coded in the program, and we really didn't need to use an object-oriented approach to solve this problem. However, I hope this example helps you understand how classes and objects work.

PROJECT #19: FACTORIAL IN C

Prerequisite: Projects #12 and #13

In this project, you'll build a factorial program in the C programming language, like the one we covered earlier in this chapter. You'll then examine the machine code that was generated when the code was compiled. Use the text editor of your choice to create a new file named *fac2.c* in the root of your home folder. Enter the following C code:

```
#include <stdio.h>

// Calculate the factorial of n.
int factorial(int n)❶
{
    int result = n;

    while(--n > 0)
    {
        result = result * n;
    }

    return result;
}

int main()❷
{
    int answer = factorial(4);❸
    printf('%d\n', answer);❹
}
```

You can see that the `factorial` function ❶ is exactly the same as the C example given earlier in the chapter; this is the core code for calculating the factorial. However, to make this a usable program, we also have a `main` function ❷ that serves as the entry point—this is where the program begins execution. From `main`, the program calls the `factorial` function with a value of 4, storing the result in a local variable named `answer` ❸. The program then prints the value of `answer` to the terminal ❹.

Once the file is saved, use `gcc` to compile your code into an executable file. The following command takes *fac2.c* as an input and outputs an executable file named *fac2*. No separate linking step is required. Also note the `-O` (that is a capital letter O) command

line option: this means enable compiler optimizations. I added this option here because in this case it produces code that's more similar to the assembly code from Project #12.

```
$ gcc -O -o fac2 fac2.c
```

Now try running the code using the following command. If everything works as expected, the program should print the calculated result of 24 on the next line.

```
$ ./fac2
```

Now that you have a `fac2` executable file, use the same techniques you used in Projects #12 and #13 to inspect the compiled file. I won't walk you through all the details again, but the same approaches you used before will work here as well. Here are a few commands to get you started:

```
$ hexdump -C fac2
$ objdump -s fac2
$ objdump -d fac2
$ gdb fac2
```

You should see right away that there's a lot of stuff in the `fac2` file! The compiled ELF binary carries some overhead required for a program written in C. On my computer, the original `fac` ELF file was 940 bytes, whereas the `fac2` ELF file is 8,364 bytes, a 9X increase! Of course, the C version does include additional functionality to print out the value, so some size increase is expected.

When looking at disassembled code, it's the `factorial` function you want to initially examine. Compare it to the factorial code that you wrote in assembly language back in Chapter 8. You may notice `gdb` shows a different entry point than `main`. This is because C programs have some initialization code that's called before your `main` entry point is called. If you want to skip this code and go right to the `factorial` function, you can set a breakpoint (`break factorial`) then `run`, and then `disassemble`.

The machine instructions generated on your machine may differ somewhat, but here's the `factorial` function machine code and corresponding assembly language generated on my computer. This is output from `objdump -d fac2`:

```
00010408 <factorial>:
 10408:    e2403001        sub    r3, r0, #1❶
 1040c:    e3530000        cmp    r3, #0❷
 10410:    d12ffff1e       bxle   lr❸
 10414:    e0000093        mul    r0, r3, r0❹
 10418:    e2533001        subs   r3, r3, #1❺
 1041c:    1affffffc       bne    10414 <factorial+0xc>❻
 10420:    e12ffff1e       bx    lr❾
```

Before this function is called, the value of `n` has been stored in `r0`. When the function begins, right away it decrements `n` and stores the result in `r3` ❶. The program then compares `r3` (that is, `n`) to zero ❷. If `n` is less than or equal to zero ❸, then the program returns from the function. Otherwise, `result`, stored in `r0`, is calculated as `result × n` ❹. Next `n` is decremented ❺, and if `n` is not zero ❻, the program goes through the loop again, branching back to address 10414 ❻. Once `n` reaches zero, the loop ends and the function returns ❼.

10

OPERATING SYSTEMS



So far, we've examined a computer's hardware and software. In this chapter, we look at a particular kind of software: operating systems. First, we cover the challenges of programming without an operating system (OS). Then we look at an overview of OSes. We spend the bulk of the chapter detailing some of the core capabilities of operating systems. In the projects, you have a chance to examine the workings of Raspberry Pi OS.

Programming Without an Operating System

Let's begin by considering what it's like to use and program a device without an OS. As you'll see in a minute, operating systems provide an interface between hardware and other software. However, on a device without an OS, the software has direct access to the hardware. There are many examples of computers that work this way, but let's focus on one type in particular: early video game consoles. If we look back at game consoles such as the Atari 2600, the Nintendo Entertainment System, or the Sega Genesis, we find hardware that runs code from a cartridge, with no operating system in place. Figure 10-1 illustrates the

idea that the game's software ran directly on the console hardware, with nothing in between.



Figure 10-1: Early video games ran directly on game console hardware, with no operating system.

To use such a system you simply inserted a cartridge and turned on the system to start the game. The game console ran only one program at a time—the game currently in the cartridge slot. On most systems of this kind, turning on the system without a cartridge inserted did nothing, since the CPU didn't have any instructions to run. To switch to a different game, you needed to turn off the system, swap cartridges, and turn it back on. There was no concept of switching between programs while the system was running. Nor were any programs running in the background. A single program, the game, had the complete attention of the hardware.

As a programmer, making a game for a system like this meant taking responsibility for directly controlling hardware with code. Once the system powered up, the CPU began running the code on the cartridge. The game developer not only had to write software for the game's logic but also had to initialize the system, control the video hardware, read the hardware state of the controller inputs, and so forth. Different console hardware had radically different designs, so a developer needed to understand the intricacies of the hardware they were targeting.

Fortunately for old-school game developers, a game console would retain the same hardware design, more or less, during the years it was manufactured. For example, all Nintendo Entertainment System (NES) consoles have the same type of processor, RAM, picture processing unit (PPU), and audio processing unit (APU). To be a successful NES developer you had to have a solid understanding of all this hardware, but at least the hardware was the same in every NES sold to gamers.

Developers knew exactly what hardware would be in a system, so they could target their code to that specific hardware, which allowed them to squeeze every ounce of performance from the system. However, to port their game to another type of game console, they often had to rewrite a substantial portion of their code. Additionally, every game cartridge had to include similar code to accomplish fundamental tasks, such as initializing the hardware. Although developers could reuse code they had previously written for other games, this still meant different developers were solving the same challenges over and over, with varying degrees of success.

Operating Systems Overview

Operating systems provide a different model for programming, and in doing so, address many of the challenges associated with writing code that directly targets specific hardware. An *operating system (OS)* is software that communicates with computer hardware and provides an environment for the execution of programs. Operating systems allow programs to request system services, such as reading from storage or communicating over a network. OSes handle the initialization of a computer system and manage the execution of programs. This includes running multiple programs in parallel, or *multitasking*, ensuring that multiple programs can share time on the processor and share system resources. An OS puts boundaries in place to ensure that programs are isolated from each other and from the OS, and to ensure that users who share a system are granted appropriate access. You can think of an operating system as a layer of code between hardware and applications, as illustrated in Figure 10-2.

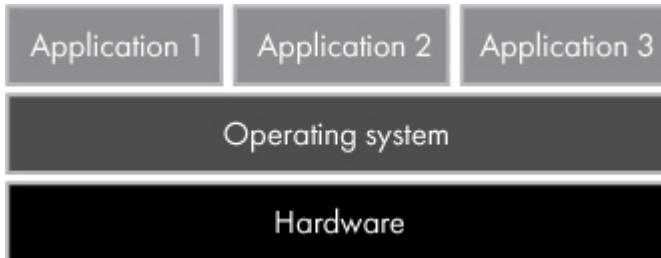


Figure 10-2: An operating system acts as a layer between hardware and applications.

This layer provides a set of capabilities that abstract away the details of the underlying hardware, allowing software developers to focus on the logic of their software, rather than on communicating with specific hardware. As you might expect, this is very useful, given the diversity of today's computing devices. Considering the amazing variety of hardware found in smartphones and PCs, writing code for each type of device is impractical. Operating systems hide the details of hardware and provide common services that applications can build on.

At a high level, the components included with an operating system can be categorized into two major buckets:

- The kernel
- Everything else

An operating system *kernel* is responsible for managing memory, facilitating device I/O, and providing a set of system services for applications. The kernel allows multiple programs to run in parallel and share hardware resources. It is the core part of an operating system, but it alone provides no way for end users to interact with the system.

Operating systems also include non-kernel components that are needed for a system to be of use. This includes the *shell*, a user interface for working with the kernel. The terms *shell* and *kernel* are part of a metaphor for operating systems, where the OS is thought of as a nut or seed. The kernel is at the core; a shell surrounds it. The shell can be either a command line interface (CLI) or a graphical user interface (GUI). Some examples of shells are the Windows shell GUI (including

the desktop, Start menu, taskbar, and File Explorer), and the Bash shell CLI found on Linux and Unix systems.

Some capabilities of operating systems are provided by software that runs in the background, distinct from the kernel, known as *daemons* or *services* (not to be confused with kernel system services mentioned earlier). An example of such a service is Task Scheduler on Windows or cron on Unix and Linux, both of which allow the user to schedule programs to run at certain times.

Operating systems also commonly include *software libraries* for developers to build on. Such libraries include common code that many applications can leverage. Additionally, components of the operating system itself, such as the shell and services, use the functionality provided by such libraries.

When it comes to interacting with hardware, the kernel acts in partnership with device drivers. A *device driver*, or simply *driver*, is software designed to interact with specific hardware. An operating system's kernel needs to work with a wide variety of hardware, so rather than designing the kernel to know how to interact with every hardware device in the world, software developers implement the code for specific devices in device drivers. Operating systems typically include a set of device drivers for common hardware and also provide a mechanism for installing additional drivers.

Most operating systems include a collection of basic applications like a text editor and calculator, often referred to collectively as *utilities*. A web browser is also a standard inclusion for many operating systems. Such utilities are arguably not truly part of the operating system and are rather simply applications, but in practice, most operating systems include this kind of software. Figure 10-3 provides a summarized view of the components included in an operating system.

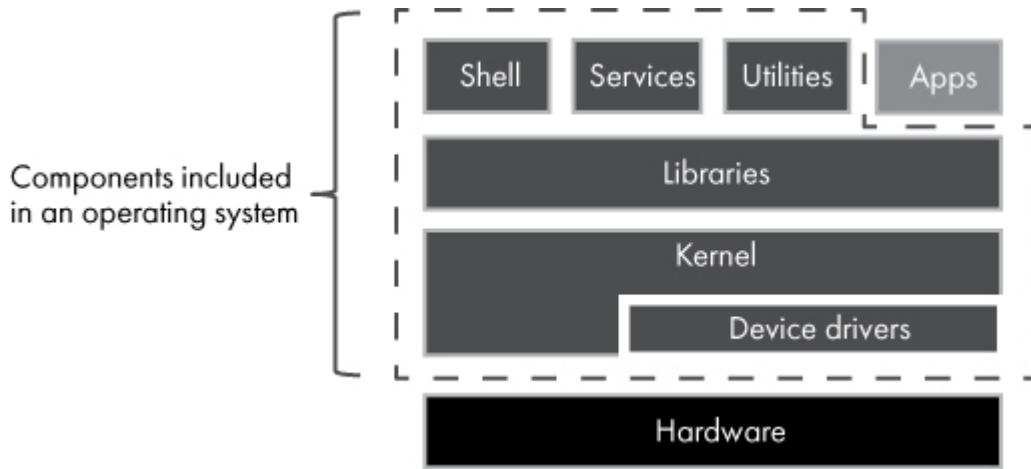


Figure 10-3: An operating system includes multiple components.

As you can see in Figure 10-3, at the foundation of the software stack, right above hardware, are the kernel and device drivers. Libraries provide functionality that applications build on, so libraries are shown as a layer between the kernel and applications. The shell, services, and utilities also build on libraries.

Operating System Families

Today, there are two dominant operating system families: Unix-like operating systems, and Microsoft Windows. As the name implies, *Unix-like* operating systems behave like a Unix operating system. Linux, macOS, iOS, and Android are all examples of Unix-like operating systems. *Unix* was first developed at Bell Labs and has a history that goes back to the 1960s. Unix initially ran on a PDP-7 minicomputer, but it has since been ported to many kinds of computers. Originally written in assembly language, Unix was later rewritten in C, allowing it to be compiled for various processors. Today it's used on servers, and it has a strong presence on personal computers and smartphones thanks to Apple's macOS and iOS, both of which are based on Unix. Unix supports multiple users, multitasking, and a unified, hierarchical directory structure. It has a robust command line shell, supported by

well-defined standard command line tools that can be used together to accomplish complex tasks.

The *Linux* kernel was originally developed by Linus Torvalds, who set out to create an operating system that was similar to Unix. Linux isn't Unix, but it's certainly Unix-like. It behaves much like Unix while not including any Unix source code. A *Linux distribution* is an OS that's a bundling of the Linux kernel with other software. The Linux kernel is *open source*, meaning its source code is freely available. Many distributions of Linux are available at no cost. A typical Linux distribution includes a Linux kernel and a collection of Unix-like components from the GNU project (pronounced "guh-new").

GNU, a recursive acronym that stands for *GNU's Not Unix*, is a software project started in the 1980s, with a goal of creating a Unix-like operating system as free software. The GNU project and Linux are separate efforts, but they have become closely associated. The release of the Linux kernel in 1991 prompted an effort to port GNU software to Linux. At the time, GNU didn't have a complete kernel, whereas Linux lacked a shell, libraries, and so forth. Linux provided a kernel for GNU code to run upon, while the GNU project provided a shell, libraries, and utilities to Linux. In this way, the two projects are complementary, and together form a complete operating system.

Today, people commonly use the term *Linux* to refer to operating systems that are combinations of the Linux kernel and GNU software. This is somewhat controversial, since calling the entire OS "Linux" doesn't recognize the large part that GNU software plays in many Linux distributions. That said, in this book I follow the prevalent convention of referring to the entire OS as Linux, rather than GNU/Linux or something similar.

Today, Linux is commonly found on servers and embedded systems, and it's popular with software developers. The Android operating system is based on the Linux kernel, so Linux has a huge presence in the smartphone market. Raspberry Pi OS (previously called Raspbian) is also a Linux distribution that includes GNU software, and we'll be

using Raspberry Pi OS to explore Linux further. In general, in this book I'm going to lean on Linux rather than Unix when giving examples of Unix-like behavior.

Microsoft Windows is the dominant operating system on personal computers, including desktops and laptops. It also has a strong presence in the server space (Windows Server). Windows is unique in that it doesn't trace its roots back to Unix. Early versions of Windows were based on MS-DOS (Microsoft Disk Operating System). Although popular in the home computer market, these early versions of Windows were not robust enough to compete against Unix-like operating systems in the server or high-end workstation market.

In parallel to the development of Windows, Microsoft partnered with IBM in the 1980s to create the OS/2 operating system, an intended successor to MS-DOS on the IBM PC. Microsoft and IBM disagreed on the direction of the OS/2 project, and in 1990, IBM took over development of OS/2, whereas Microsoft pivoted their efforts to another operating system they already had under development, Windows NT. Unlike the MS-DOS-based versions of Windows, Windows NT was based on a new kernel. Windows NT was designed to be portable across different hardware, be compatible with various types of software, support multiple users, and provide high levels of security and reliability. Microsoft hired Dave Cutler from Digital Equipment Corporation (DEC) to lead the work on Windows NT. He brought a number of former DEC engineers with him, and elements of the NT kernel's design can be traced to Dave Cutler's work on the VMS operating system at DEC.

In its early releases, Windows NT was positioned as a business-focused version of Windows that would coexist with the consumer-focused version of Windows. These two Windows versions were quite different in their implementations, but they shared a similar user interface and programming interface. The user interface similarities meant that users familiar with Windows could readily be productive on a Windows NT system. The common programming interface allowed software developed for DOS-based Windows to work, sometimes

without alteration, on Windows NT. With the release of Windows XP in 2001, Microsoft brought the NT kernel to a consumer-focused release of Windows. Since the release of Windows XP, all versions of desktop and server Windows have been built upon the NT kernel.

Table 10-1 lists some operating systems and devices commonly in use today and the OS family for each.

Table 10-1: Common Operating Systems

OS or device	Family	Notes
Android	Unix-like	Android uses the Linux kernel, although otherwise, it isn't very Unix-like. Its user experience and application programming interfaces are quite different from a typical Unix system.
iOS	Unix-like	iOS is based on the Unix-like open source Darwin operating system. Like Android, the iOS user experience and programming interface are different from a typical Unix system.
macOS	Unix-like	macOS is based on the Unix-like open source Darwin operating system.
PlayStation 4	Unix-like	The PlayStation 4 OS is based on the Unix-like FreeBSD kernel.
Raspberry Pi OS	Unix-like	Raspberry Pi OS is a Linux distribution.

OS or device	Family	Notes
Ubuntu	Unix-like	Ubuntu is a Linux distribution.
Windows 10	Windows	Windows 10 uses the Windows NT kernel.
Xbox One	Windows	Xbox One has an OS that uses the Windows NT kernel.

EXERCISE 10-1: GET TO KNOW THE OPERATING SYSTEMS IN YOUR LIFE

Choose a couple of computing devices that you own or use, say a laptop, smartphone, or game console. What operating system does each device run? To what operating system family (Windows, Unix-like, other) does each belong?

Kernel Mode and User Mode

An operating system is responsible for ensuring that the programs that run on it behave well. What does this mean in practice? Let's look at some examples. Each program must not interfere with other programs or with the kernel. Users shouldn't be able to modify system files. Applications must not be allowed to directly access hardware; all such requests must go through the kernel. Given these kinds of requirements, how can the operating system ensure that non-OS code complies with the mandates of the operating system? This is handled by leveraging a CPU capability that grants the operating system special rights while placing restrictions on other code; this is known as the *privilege level* of the code. A processor may offer more than two levels of privilege, but most operating systems only use two levels. The level of higher privilege is known as *kernel mode*, and the level of lower privilege is known as *user mode*. Kernel mode is also referred to as *supervisor mode*.

Code running in kernel mode has full access to the system, including access to all memory, I/O devices, and special CPU instructions. Code running in user mode has limited access. Generally speaking, the kernel and many device drivers run in kernel mode, whereas everything else runs in user mode, as illustrated in Figure 10-4.

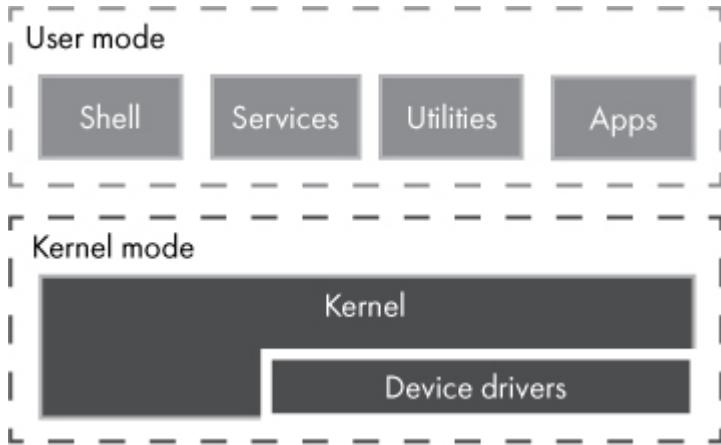


Figure 10-4: The division of code that runs in user mode vs. kernel mode

Code allowed to run in kernel mode is *trusted*, whereas user mode code is *untrusted*. Code that runs in kernel mode has full access to everything on a system, so it better be trustworthy! By only allowing trusted code to run in kernel mode, the operating system can ensure that user mode code is well-behaved.

KERNEL MODE COMPONENTS IN WINDOWS

It's worth noting that Microsoft Windows has a few other major components that run in kernel mode. In Windows, foundational kernel mode capabilities are actually split between two components: the kernel and the *executive*. The distinction is only relevant when discussing the internal architecture of Windows; the separation is not of concern to most software developers or users. In fact, the compiled machine code for both the kernel and the executive is contained in the same file (*ntoskrnl.exe*). I won't distinguish between the Windows NT kernel and executive for the remainder of this book. Besides the kernel, executive, and device drivers, Windows has other major components that run in kernel mode. The *Hardware Abstraction Layer (HAL)* isolates the kernel, executive, and device drivers from differences in low-level hardware, such as variations in motherboards. The *windowing and graphics system (win32k)* provides capabilities for drawing graphics and programmatically interacting with user interface elements.

Processes

One of an operating system's main functions is to provide a platform for programs to run. As we saw in the previous chapter, programs are sequences of machine instructions, typically stored in an executable file. However, a set of instructions stored in a file can't actually perform any work on its own. Something needs to load the file's instructions into memory and direct the CPU to run the program, all while ensuring the program doesn't misbehave. That's the job of the operating system.

When an operating system starts a program, it creates a *process*, a running instance of that program. Earlier we covered things that run in user mode (such as the shell, services, and utilities)—each of these execute within a process. If code is running in user mode, it's running within a process, as illustrated in Figure 10-5.

A process is a container in which a program runs. This container includes a private virtual memory address space (more on this later), a copy of the program code loaded into memory, and other information about the state of the process. A program can be started multiple times, and each execution results in the operating system creating a new process. Each process has a unique identifier (a number) called a *process identifier*, a *process ID*, or just a *PID*.

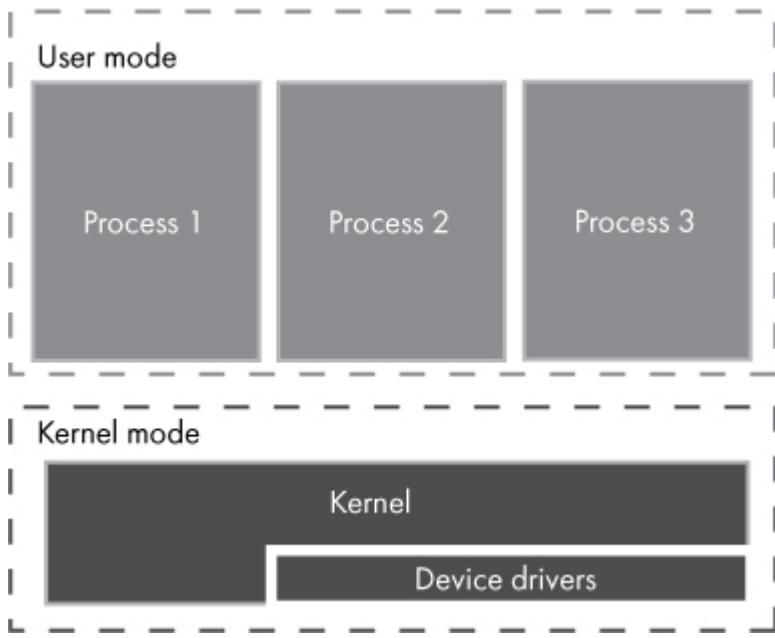


Figure 10-5: Processes run in user mode

Other than initial processes started by the kernel, every process has a parent, the process that started it. This relationship of parent to child creates a tree of processes. If a child's parent process terminates before the child, the child becomes an *orphan process*, meaning, not surprisingly, it has no parent. On Windows, the orphaned child process simply remains parentless. On Linux, an orphaned process is typically adopted by the *init process*, the first user mode process to start on a Linux system.

Figure 10-6 shows a process tree on Raspberry Pi OS. This view was generated using the `pstree` utility.

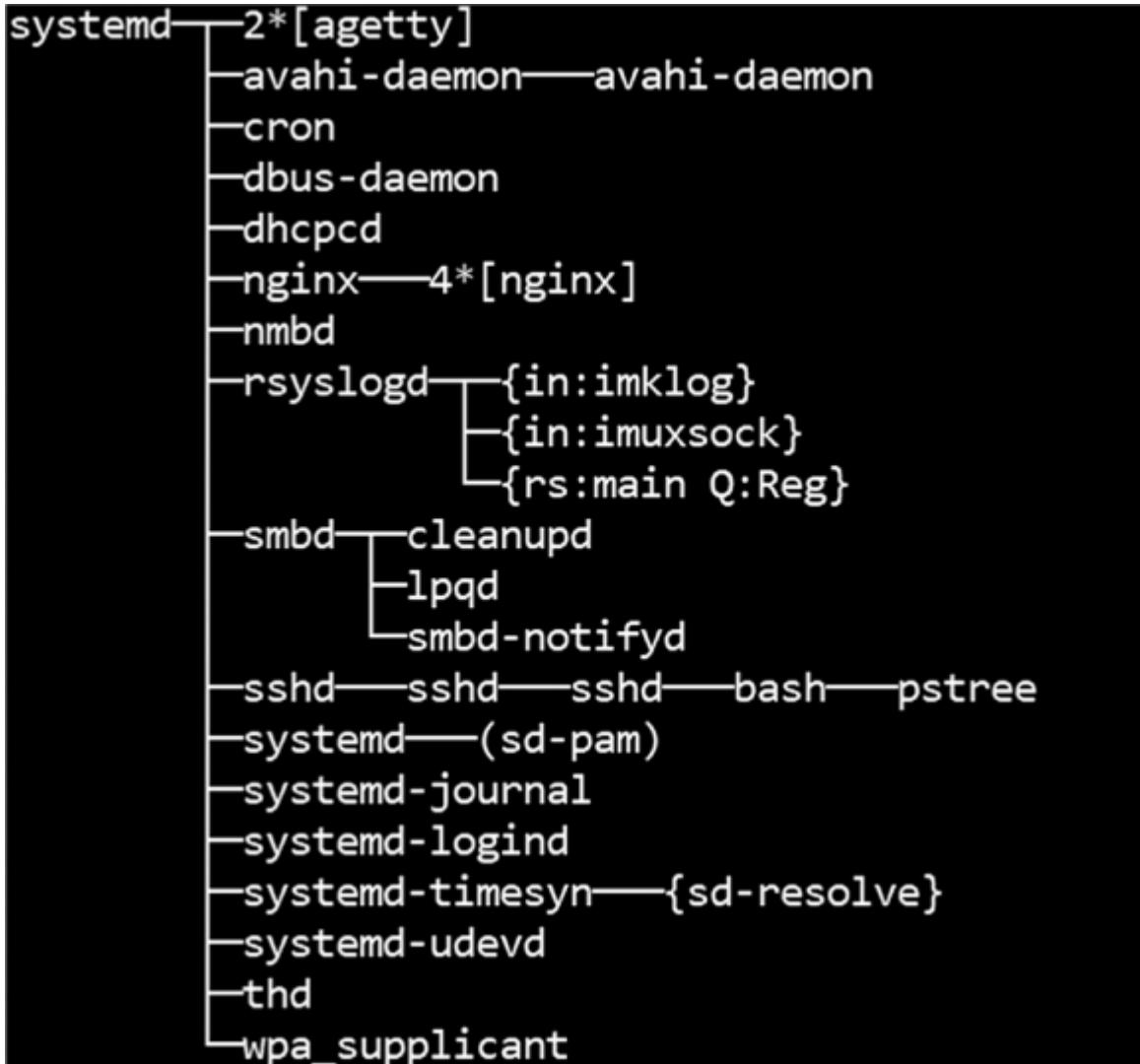


Figure 10-6: An example Linux process tree as shown by `pstree`

In Figure 10-6, we see that the init process was `systemd`; it was the first process to start, and it in turn started other processes. Child threads are shown with curly braces (more on threads soon). To generate this output, I ran the `pstree` command from a command line shell, and in the output, you can see that `pstree` itself is running, as expected. It's the child of `bash` (the shell), which in turn is the child of `sshd`. In other words, you can tell from this output that I ran `pstree` from a Bash shell that was opened in a remote Secure Shell (SSH) session.

To see the process tree on a computer running Windows, I recommend that you use the Process Explorer tool that you can

download from Microsoft. It's a GUI application that gives you a rich view of the processes running on your computer.

NOTE

Please see Project #20 on page 218, where you can look at running processes on your device.

Threads

By default, a program executes instructions sequentially, handling one task at a time. But what if a program needs to perform two or more tasks in parallel? For example, let's say a program needs to perform some long-running calculation while updating the user interface at the same time, perhaps to show a progress bar. If the program is completely sequential, once the program begins its calculation, the user interface is neglected, since the CPU time allocated to the program must be spent elsewhere. The desired behavior is that the UI updates while the calculation runs—these are two separate tasks that need to happen in parallel. Operating systems provide this capability with *threads of execution*, or just *threads*. A thread is a schedulable unit of execution within a process. A thread runs within a process and can execute any program code loaded in that process.

The code run by a thread typically encompasses a particular task that a program wishes to accomplish. Since threads belong to a process, they share an address space, code, and other resources with all the other threads in that process. A process begins with one thread and may create other threads as needed when work needs to be handled in parallel. Each thread has an identifier called a *thread ID*, or *TID*. The kernel also creates threads to manage its work. Figure 10-7 illustrates the relationship between threads, processes, and the kernel.

In Windows, threads and processes are distinct object types. A process object is a container, and threads belong to a process. In Linux, the distinction is more nuanced. The Linux kernel represents both processes and threads using a single data type that serves as both a process and a thread. In Linux, a group of threads that share an address

space and have a common process identifier are considered a process; there is no separate process type.

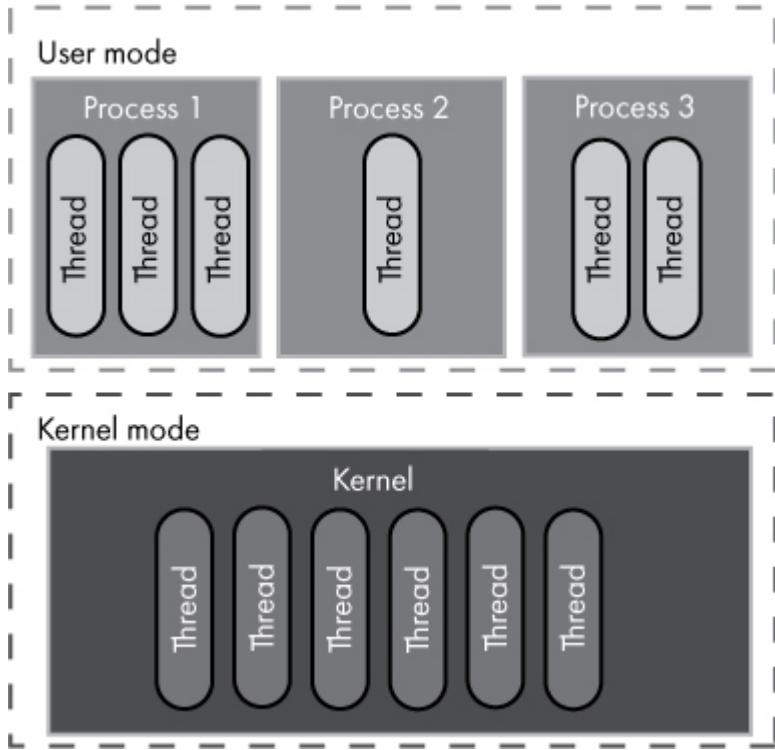


Figure 10-7: Threads belong to user mode processes or to the kernel.

The Linux terminology used to refer to the identifiers for processes and threads can be a bit confusing. In user mode, a process has a process ID (PID) and a thread has a thread ID (TID). This is just like Windows. However, the Linux kernel refers to the ID of a thread as a PID and the ID of a process as a *thread group identifier (TGID)*!

NOTE

Please see Project #21 on page 220, where you can create your own thread.

What does it really mean for multiple threads to run in parallel? Let's say your computer has 10 processes running, and each process has 4 threads. That's 40 threads in user mode alone! We say that threads run in parallel, but can all 40 threads really run at the same time? No, not unless your computer has 40 processor cores, which it probably doesn't.

Each processor core can only run one thread at a time, so the number of cores in a device determines how many threads can run at once.

PHYSICAL AND LOGICAL CORES

Not all cores are equally capable of parallelism. A *physical core* is a hardware implementation of a core within a CPU. *Logical cores* represent the ability of a single physical core to run multiple threads at once (one thread per logical core). Intel refers to this capability as *hyper-threading*. As an example, the computer I'm using to write this book has two physical cores, each with two logical cores, for a total of four logical cores. This means that my computer can run four threads at once, although logical cores cannot achieve the full parallelism of physical cores.

So if we have 40 threads that need to run, but only 4 cores, what happens? The operating system implements a *scheduler*, a software component that's responsible for ensuring that threads each get their turn to run. Different approaches are used across operating systems to implement scheduling, but the fundamental goal is the same: give threads time to run. A thread gets a short period of time to run (known as a *quantum*), then the thread is suspended to allow another thread to run. Later, the first thread is scheduled again, and it picks up where it left off. This is mostly hidden from the thread's code and the developer who wrote the application. From the perspective of the thread's code, it's running continuously, and developers write their multithreaded applications as if all their threads were running continuously in parallel.

Virtual Memory

Operating systems support multiple running processes, each of which need to use memory. Most of the time, one process does not need to read or write to the memory of another process, and in fact, it's generally undesirable. We don't want a misbehaving process stealing data or overwriting data in another process or, worse, in the kernel. Additionally, developers don't want their process's address space to

become fragmented from the memory usage of other processes. For these reasons, operating systems do not grant user mode processes access to physical memory, and instead each process is presented with *virtual memory*—an abstraction that gives each process its own large, private address space.

In Chapter 7, we covered memory addressing in which each physical byte in hardware is assigned an address. Such hardware memory addresses are called *physical addresses*. These addresses are typically hidden from user mode processes. Operating systems instead present processes with *virtual memory*, where each address is a *virtual address*. Each process is given its own virtual memory space to work in. To an individual process, memory appears as a large range of addresses. When a process writes to a certain virtual address, that address does not directly refer to a hardware memory location. The virtual address is translated to a physical address when needed, as shown in Figure 10-8, but the details of this translation are hidden from the process.

The advantage of this approach is that each process is given a large, private range of virtual memory addresses that it can work with. In general, each process on a system is presented the *same* range of memory addresses. For example, each process might be given 2GB of virtual address space, from address 0x00000000 to 0x7FFFFFFF. This might seem problematic; what happens when two programs try to use the same memory address? Can one program overwrite or read another program’s data? Thanks to virtual addressing, this isn’t a problem.

The same virtual address for multiple programs maps to different physical addresses, so there’s no chance of one program accidentally accessing another’s data in memory. This means that the data stored at a certain virtual address is different across different processes—the virtual addresses may be the same, but the data stored there differs. That said, mechanisms are in place for programs to share memory if they need to. In older operating systems, memory space wasn’t so cleanly divided, leading to abundant opportunities for programs to corrupt memory in other programs or even in the operating system. Fortunately, all

modern operating systems ensure separation of memory between processes.

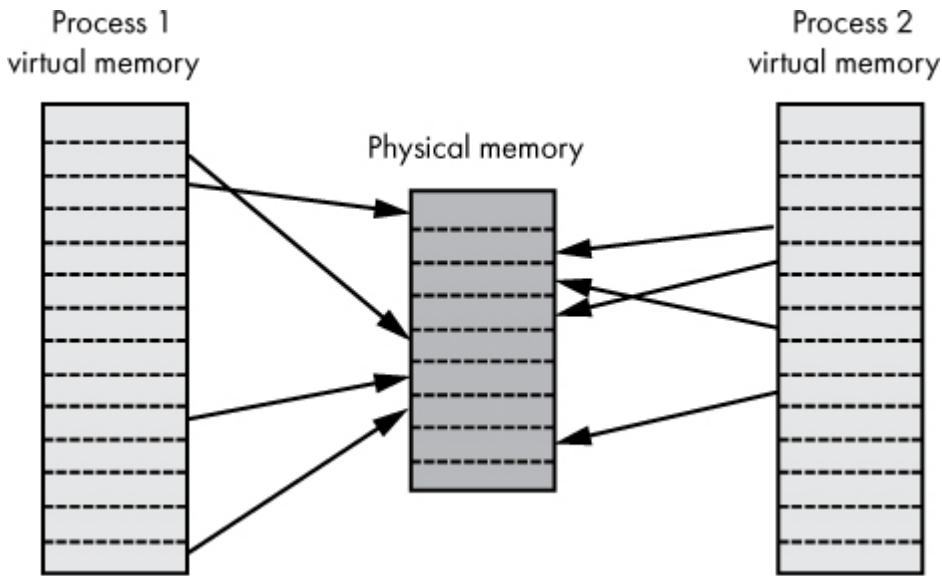


Figure 10-8: Virtual address space for each process is mapped to physical memory

It's important to understand that although the address range of a process may be 2GB in size (for example), that doesn't mean that all 2GB of virtual memory is immediately available for the process to use. Only a subset of those addresses is backed by physical memory. Think back to the projects you performed in Chapters 8 and 9; those were actually virtual memory addresses that you were examining, not physical ones.

The kernel has a separate virtual address space to work in with a range of addresses that's distinct from the address range assigned to user mode processes. Unlike user mode address space, kernel address space is shared by all code running in kernel mode. That means that any code running in kernel mode has access to everything in the kernel address space. This also gives such code the opportunity to modify the contents of any kernel memory. This reinforces the idea that code that runs in kernel mode must be trusted!

So how is virtual address space divided between user mode and kernel mode? Let's look at 32-bit operating systems. As discussed in Chapter 7, for a 32-bit system, memory addresses are represented as 32-

bit numbers, which means 4GB of address space in total. This address space's range of addresses must be split between kernel mode and user mode. For a 4GB address space, both Windows and Linux allow for a split of either 2GB user/2GB kernel or 3GB user/1GB kernel, based on a configuration setting. Figure 10-9 illustrates an even 2GB split of virtual memory.

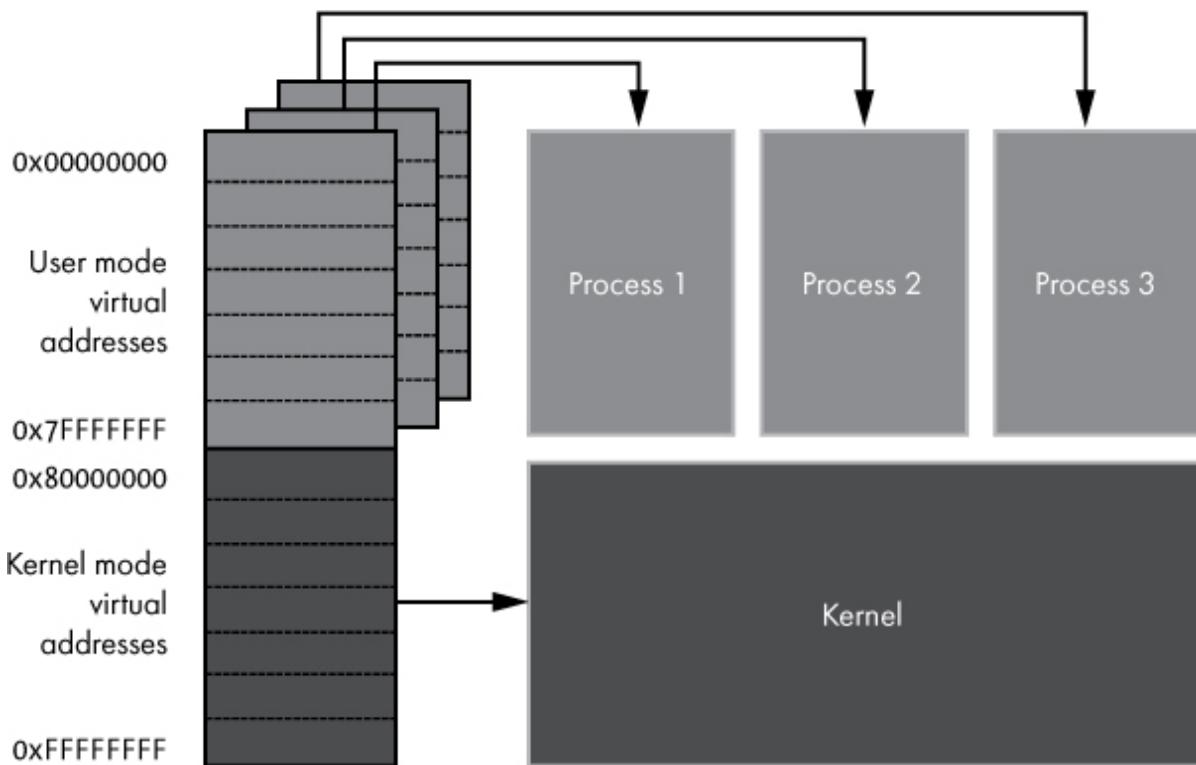


Figure 10-9: Virtual address space on a 32-bit system with an even 2GB/2GB split

Keep in mind that we're strictly concerned with *virtual* addresses here. A 32-bit system has 4GB of virtual address space regardless of how much *physical* memory it has. Let's say a computer only has 1GB of RAM; it still has 4GB of virtual address space under a 32-bit OS. Recall that a virtual address range doesn't represent mapped physical memory, only a range where physical memory *can be* mapped. That said, it's certainly possible for the kernel and all running processes to request more bytes of virtual memory than the total size of RAM. In that situation, the operating system can move bytes of memory to secondary storage to make room in RAM for newly requested memory, a process

known as *paging*. Typically, the least used memory gets paged first so that actively used memory can remain in RAM. When the paged memory is needed, the OS must load it back into RAM. Paging allows for greater virtual memory usage, at the cost of a performance hit incurred while bytes are moved to and from secondary storage. Keep in mind that secondary storage is significantly slower than RAM.

NOTE

Please see Project #22 on page 222, where you can examine virtual memory.

With the arrival of 64-bit processors and operating systems came the potential for much, much larger address spaces. If we represented memory addresses with a full 64 bits, virtual address space would be about 4 *billon* times the size of 32-bit address space! However, such a large address space isn't needed today, so 64-bit operating systems use a smaller number of bits to represent addresses. Different 64-bit operating systems on different processors use varying numbers of bits to represent an address. Both 64-bit Linux and 64-bit Windows support 48-bit addresses, which translates to 256TB of virtual address space, about 65,000 times the size of 32-bit address space—more than enough space for today's typical application.

Application Programming Interface (API)

When most people think of an operating system, they think of the user interface, the shell. The shell is what people see, and it influences how people perceive the system. For example, a Windows user typically thinks of Windows as the taskbar, Start menu, desktop, and so forth. However, the user interface is actually only a small part of the operating system's code, and it's just an interface, the point where the system and the user meet. From the perspective of an application (or a software developer), interacting with the operating system isn't defined by the UI, but by the operating system's *application programming interface (API)*. APIs are not only for operating systems; any software that wants to

allow a programmatic means of interaction can provide an API, but our focus here is specifically on OS APIs.

An OS API is a specification, defined in source code and described in documentation, that details how a program should interact with the OS. A typical OS API includes a list of functions (including their names, inputs, and outputs) and data structures needed for interacting with the operating system. Software libraries included with the operating system provide the implementation of the API specification. Software developers speak of “calling” or “using” an API as a shorthand way of saying that their code is invoking one of the functions specified in the API (and implemented in a software library).

In the same way that a UI defines an OS’s “personality” for users, the API defines the OS’s personality for applications. Figure 10-10 illustrates how users and applications interact with an operating system.

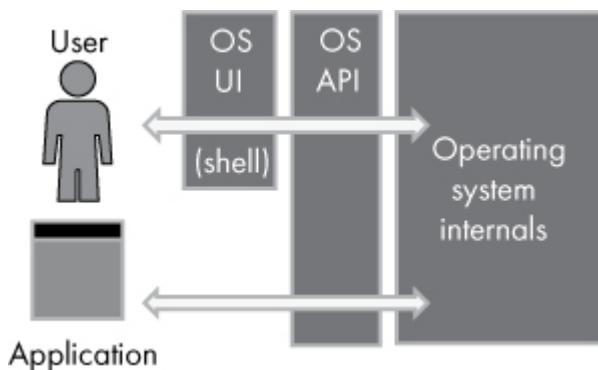


Figure 10-10: Operating system interfaces: UI for users; API for applications

As seen in Figure 10-10, users interact with the operating system user interface, also known as the shell. The shell translates the user’s commands into API calls. The API then invokes internal operating system code to perform the requested action. Applications don’t need to go through the UI; they simply call the API directly. From this point of view, the shell interacts with the operating system API just like any other application.

Let’s look at an example of interfacing with operating systems via an API. Creating a file is a common capability of operating systems, something that both users and applications need to do. Graphical shells

and command line shells provide simple ways for users to create files. However, an application doesn't need to go through the GUI or CLI to create a file. Let's examine how an application can go about creating a file programmatically.

For Unix or Linux systems, you can use an API function called `open` to create a file. The following C language example uses the `open` function to create a new file called `hello.txt`. The `O_WRONLY` flag indicates a write-only operation, and `O_CREAT` indicates that a file is to be created.

```
open("hello.txt", O_WRONLY|O_CREAT);
```

The same thing can be accomplished on Windows using the `CreateFileA` API function:

```
CreateFileA("hello.txt", GENERIC_WRITE, 0, NULL,
CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
```

Both of these examples use the C programming language. Operating systems are commonly written in C, so their APIs tend to be naturally suited for use in a C program. For programs written in other languages, the OS API must still be called when the program runs, but the programming language wraps that API call in its own syntax, hiding the details of the API from the developer. This allows for code that's portable across operating systems. Even the C language does this, providing a standard library of functions that work on any operating system. These functions, in turn, must make an OS-specific API call when they run. Consider again the example of creating a file; in C we can instead use the `fopen` function as shown in the following code. This function is part of the C language's standard library and works on any operating system.

```
fopen("hello.txt", "w");
```

As another example, we can use the following Python code to create a new file. This code works on any OS where a Python interpreter is

installed. The Python interpreter takes care of calling the appropriate OS API on behalf of the application.

```
open('hello.txt', 'w')
```

For Unix-like operating systems, the API varies somewhat based on the specific flavor of Unix or Linux and the version of the kernel. However, most Unix-like operating systems comply with a standard specification, either in full or in part. This standard is known as the *Portable Operating System Interface (POSIX)*, and it provides a standard not only for the OS API, but also for the shell's behavior and included utilities. POSIX provides a baseline for Unix-like operating systems, but a modern Unix-like OS often has its own API. *Cocoa* is Apple's API for macOS, and there is a similar API for iOS known as *Cocoa Touch*. Android also has its own set of programming interfaces, collectively known as the *Android Platform APIs*.

The other major OS family, Windows, has its own API. The *Windows API* has grown and expanded over time. The original version of the Windows API was a 16-bit version now known as *Win16*. When Windows was updated to a 32-bit operating system in the 1990s, a 32-bit version of the API, *Win32*, was released. Now that Windows is a 64-bit operating system, there is a corresponding *Win64* API. Microsoft also introduced a new API in Windows 10, the *Universal Windows Platform (UWP)*, with a goal of making app development consistent across various types of devices that run Windows.

NOTE

Please see Project #23 on page 224, where you can try interacting with the Linux operating system API.

The User Mode Bubble and System Calls

As mentioned earlier, code that runs in user mode has limited access to the system. So what are some of the things that user mode code *can* do?

It can read and write to its own virtual memory, and it can perform mathematical and logical operations. It can control the program flow of its own code. On the other hand, code running in user mode *cannot* access physical memory addresses, including addresses used for memory-mapped I/O. That means that it cannot, on its own, print text to a console window, get input from the keyboard, draw graphics to the screen, play a sound, receive touchscreen input, communicate over a network, or read a file from a hard drive! I like to say that “user mode code runs in a bubble” (Figure 10-11). It cannot interact with the outside world, at least not without some help. Another way of stating this is that user mode code cannot directly perform I/O. The practical effect of this is that code running in user mode can do useful work, but it cannot share the results of that work without assistance.

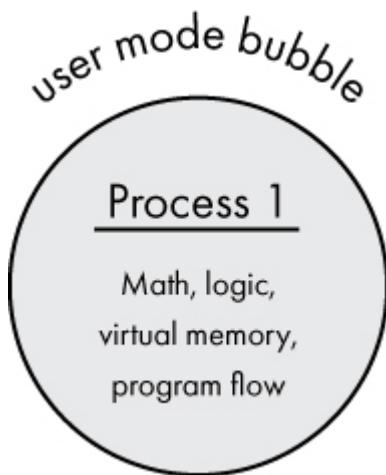


Figure 10-11: A process runs in a user mode bubble. It can do math, perform logic, access virtual memory, and control program flow, but it cannot interact directly with the outside world.

You may wonder how it is that user mode applications interact with users. Of course, applications are somehow able to interact with the outside world, but how is that accomplished? The answer is that user mode code has one other important capability: it can request that kernel mode code perform work on its behalf. When user mode code requests that kernel mode code perform a privileged operation on its behalf, this is known as a *system call*, as illustrated in Figure 10-12.

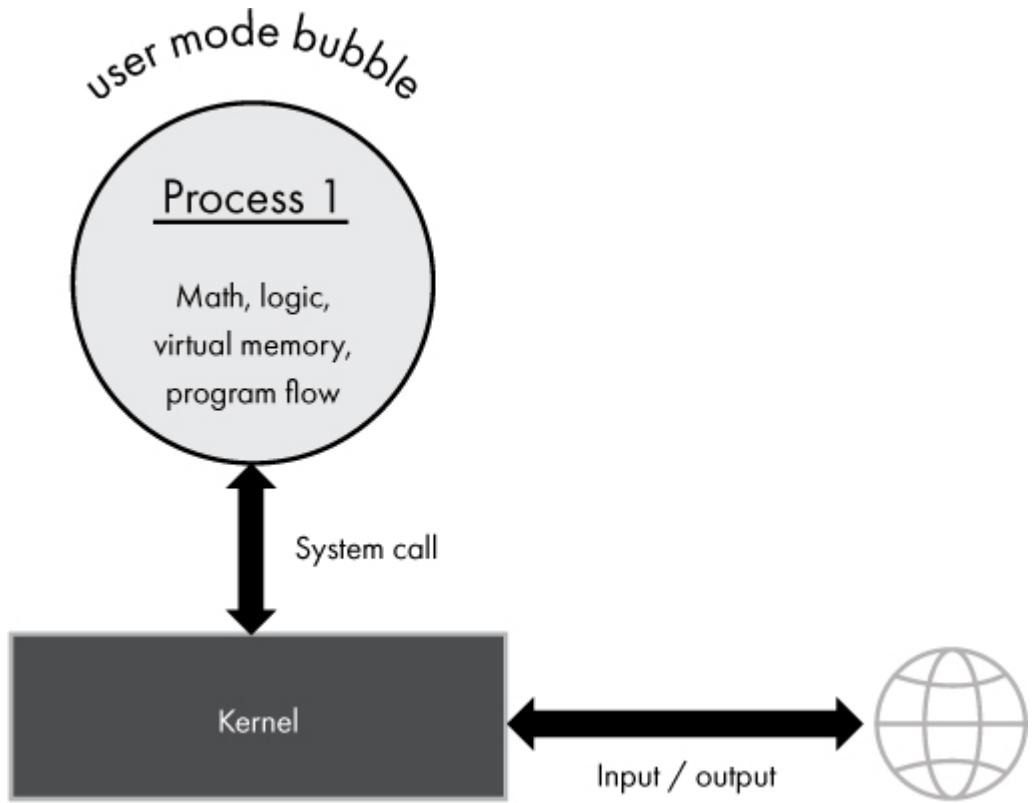


Figure 10-12: A user mode process can interact with the outside world with help from the kernel by making a system call.

For example, if user mode code needs to read from a file, it makes a system call to request that the kernel read certain bytes from a certain file. The kernel, working in conjunction with a storage device driver, performs the necessary I/O to read the file, and then provides the requested data back to the user mode process. This is illustrated in Figure 10-13.

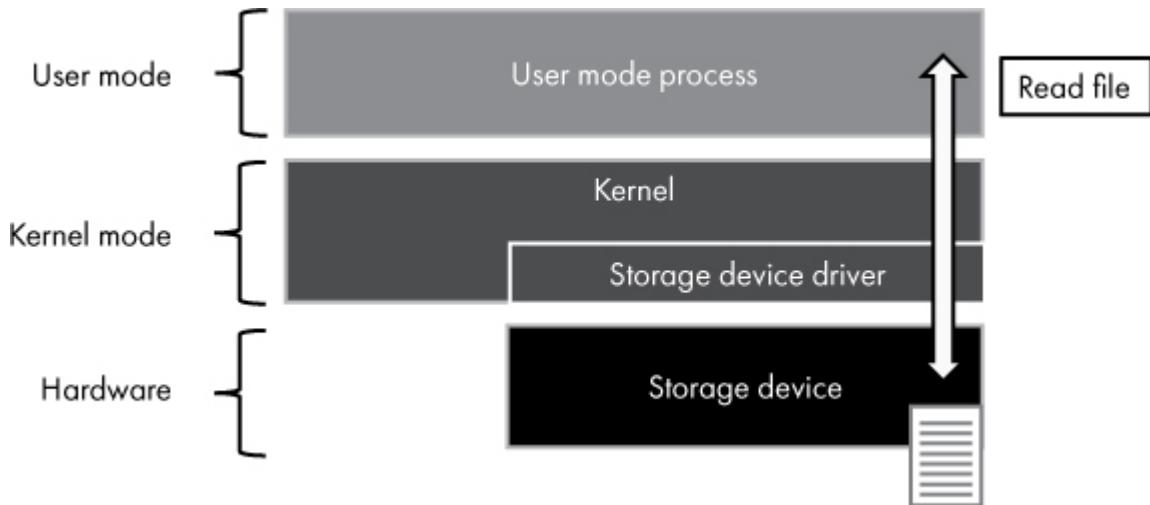


Figure 10-13: The kernel acts as an intermediary for user mode code that needs to access hardware resources, such as secondary storage.

The user mode code doesn't need to know anything about the physical storage device or any related device drivers. The kernel provides an abstraction, encapsulating the details and allowing the user mode code to simply get things done. The example API functions we covered earlier, `open` and `CreateFileA`, work this way behind the scenes, using system calls to request privileged operations. Of course, there are constraints on what the kernel will allow. A user mode process cannot, for example, read a file that it does not have access to.

CPUs provide instructions specifically to facilitate system calls. On ARM processors, the `svc` instruction (formerly `swi`) is used, and it's referred to as a *supervisor call*. On x86 processors, the `SYSCALL` and `SYSENTER` instructions are available for this purpose. Both Linux and Windows implement a large number of system calls, and each call is identified with a unique number. For example, on Linux for ARM, the `write` system call (which writes to a file) is number 4. To make a system call, a program needs to load a certain processor register with the desired system call number, put any additional parameters in other specific registers, and then execute the system call instruction.

Although software developers can make system calls directly in machine code or assembly language, fortunately this isn't needed in most cases. Operating systems and high-level programming languages

provide capabilities for making system calls in a natural way for programmers, usually through the OS API or the language's standard library. Programmers simply write code to perform an action and may not even realize that behind the scenes a system call is being made.

NOTE

Please see Project #24 on page 226, where you can observe system calls made from programs.

APIs and System Calls

Earlier we covered the topic of an operating system's API, and we just looked at system calls. How does an OS API differ from a system call? The two are related, but they are not equivalent. System calls define a mechanism for user mode code to request kernel mode services. The API describes a way for applications to interact with the operating system, regardless of whether kernel mode code is invoked. Some API functions make system calls, whereas other API functions do not require a system call. The specifics of this depend on the operating system.

Let's first look at Linux. If we restrict our definition of Linux to the kernel, we could say that the Linux API is effectively a specification for using Linux system calls, since system calls are the programmatic interface to the kernel. However, operating systems based on Linux are more than the kernel. For example, consider Android, which uses the Linux kernel. Android has its own set of programming interfaces, the Android Platform APIs.

In the case of Microsoft Windows, the Windows NT kernel provides a set of system calls, made available through an interface known as the Native API. Application developers rarely use the Native API directly; it's intended for use by operating system components. Instead, developers use the Windows API, which acts as a wrapper around the Native API. However, not all of the Windows API functions require a system call. Let's look at a couple of examples from the Windows API. The Windows API function `CreateFileW` creates or opens a file. It's a wrapper around the Native API `NtCreateFile`, which makes a

system call to the kernel. In contrast, the Windows API function `PathFindFileNameW` (which finds a filename in a path) does not interact with the Native API or make any system calls. Creating a file requires the help of the kernel, whereas finding a filename in a path string only requires virtual memory access, something that can happen in user mode.

To recap, an operating system API describes the programmatic interface for the OS. System calls provide a mechanism for user mode code to request privileged kernel mode operations. Certain API functions rely on system calls, whereas others do not.

Operating System Software Libraries

As mentioned earlier, an operating system API describes the programmatic interface to an operating system. Although a technical interface description is helpful to a programmer, when a program runs, it needs a concrete method of invoking the API. This is accomplished with software libraries. An *operating system's software library* is a collection of code, included with the OS, that provides an implementation of the OS API. That is, the library contains code that performs the operations described in the API specification. In Chapter 9, we talked about the libraries available for programming languages: both the language's standard library and additional libraries maintained by the community of developers who work in that language. The software libraries we're discussing here are similar; the only difference is that these libraries are part of operating systems.

An OS library is similar to an executable program; it's a file containing bytes of machine code. However, it typically has no entry point and therefore usually can't run on its own. Instead, the library *exports* (makes available) a set of functions that can be used by programs. A program that makes use of a software library *imports* functions from that library and is said to *link* to that library.

Operating systems include a set of library files that export the various functions defined by the API. Some of these functions are just wrappers that immediately make a kernel system call. Other functions are fully implemented in user mode code contained in the library file itself. Others are somewhere in-between, implementing some logic in user mode while also making one or more system calls, as shown in Figure 10-14.

In a typical Linux distribution, many of the available Linux kernel system calls are made available through the *GNU C Library* (or `glibc`). This library also includes the C programming language's standard library, including functions that do not require a system call. The primary `glibc` file is typically named something like `libc.so.6`, where `so` means *shared object* and `6` indicates the version. Using this library, a software developer working in C or C++ can easily make use of capabilities provided by the Linux kernel and by the C runtime library. Given the ubiquity of this library in most Linux distributions, it's reasonable to consider the functions in `glibc` as part of the standard Linux API.

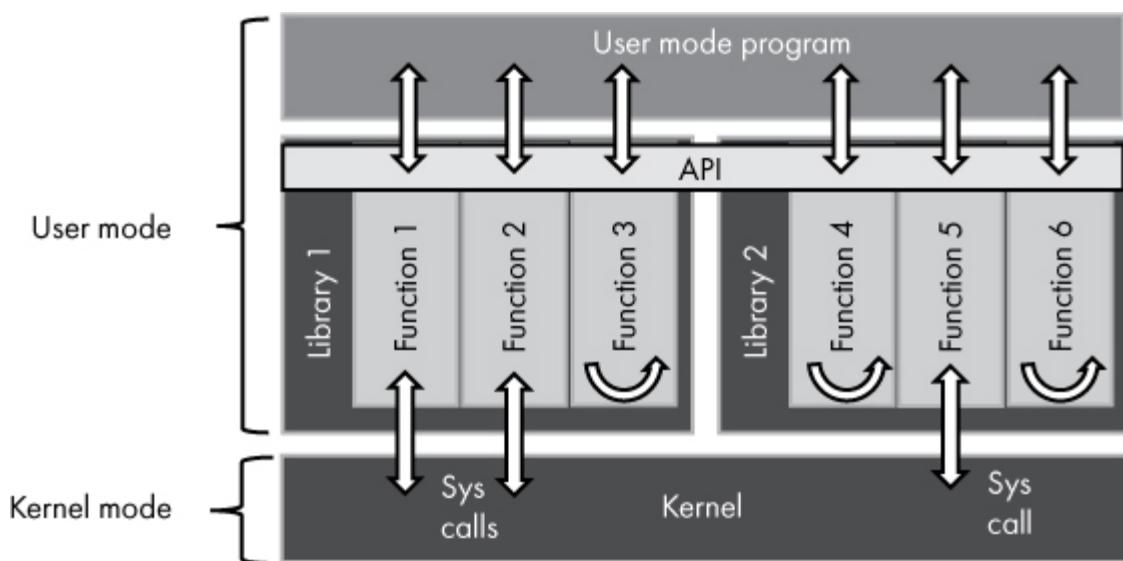


Figure 10-14: The operating system API is implemented across a set of libraries. Some functions in those libraries make system calls to the kernel; others do not. User mode programs interact with the API.

NOTE

Please see Project #25 on page 227, where you can try the GNU C Library.

The Microsoft Windows API is fairly extensive; it has grown to include many libraries over the years. The three fundamental Windows API library files are *kernel32.dll*, *user32.dll*, and *gdi32.dll*. System calls exported from the NT kernel are made available to user mode programs through *kernel32.dll*. System calls exported from win32k (the windowing and graphics system) are made available to user mode programs through *user32.dll* and *gdi32.dll*.

The *dll* extension on these files indicates that these are *dynamic link libraries*, similar to shared object (*.so*) files in Linux. That is, the *dll* file extension indicates that the file contains shared library code that a process can load and run. The 32 suffix in the filename was added as part of the 16-bit to 32-bit Windows transition. Today, 64-bit versions of Windows still retain the 32 suffix on these files for compatibility reasons. In fact, 64-bit versions of Windows include two versions of these files (same name, different directories), one for 32-bit applications and one for 64-bit applications.

NOTE

*It's possible for a program to invoke system calls without going through a software library. By setting values in processor registers and issuing a processor-specific instruction, such as *SVC* on ARM or *SYSCALL* on *x86*, a program can directly make a system call. However, this requires programming in assembly language, leading to source code that won't work across processor architectures. Furthermore, an operating system's API can include functions that aren't implemented with a system call, so making direct system calls isn't a replacement for the operating system's software libraries.*

WINDOWS SUBSYSTEM FOR LINUX

The Linux kernel and the Windows NT kernel expose different system calls, and their executables are stored in different formats, making software compiled for one OS incompatible with the other. However, in 2016, Microsoft announced the *Windows Subsystem for Linux (WSL)*, a Windows 10 feature that allows many 64-bit Linux programs to run, without modification, on Windows. In the first version of WSL, this was accomplished by intercepting system calls made by Linux executables and handling them within the NT kernel. A second version of WSL relies on a real Linux kernel to

handle system calls. This Linux kernel runs in a virtual machine alongside the NT kernel. We'll cover more on virtual machines in Chapter 13.

Application Binary Interface

Now that we've covered the concept of an application programming interface (API) and how it relates to system calls and libraries, let's examine a related concept, an ABI. An *application binary interface (ABI)* defines the machine code interface to a software library. This is in contrast to the API, which defines a source code interface. Generally speaking, an API is consistent across various processor families, whereas an ABI varies across processor families. A developer can write code that utilizes an operating system API, then compile the code for multiple processor types. The source code targets a common API, whereas the compiled code targets an architecture-specific ABI.

Once compiled, the resulting machine code adheres to the ABI for the target architecture. This means that at execution time, it's really the ABI, not the API, that defines the interaction between compiled programs and software libraries. It's important that the ABI exposed by OS libraries remains consistent over time. Such consistency allows older programs to continue to run on newer releases of the operating system without needing to be recompiled.

Device Drivers

Today's computers support a wide variety of hardware devices, such as displays, keyboards, cameras, and so forth. These devices each implement an interface for input/output, allowing the device to communicate with the rest of the system. Different device types use different approaches for I/O; a Wi-Fi adapter has very different needs from a game controller. Even devices of the same general type may implement different I/O approaches. For example, two different models of video cards may communicate very differently with the rest of the

system. Direct interactions with hardware are restricted to code running in kernel mode, but it isn't reasonable to expect an operating system kernel to know how to communicate with every device out there. This is where device drivers come in. A *device driver* is software that interacts with a hardware device and provides a programmatic interface to that hardware.

Typically, a device driver is implemented as a *kernel module*, a file containing code that the kernel can load and execute in kernel mode. This is needed to allow drivers access to hardware. Because of this, device drivers have wide-ranging access, similar to the kernel itself, so only trusted drivers should be installed. The kernel works in conjunction with device drivers to interact with hardware on behalf of code running in user mode. This allows hardware interactions to occur without the operating system or applications knowing the details of how to work with specific hardware. This is a form of encapsulation. In some cases, drivers can execute in user mode (such as those using Microsoft's User-Mode Driver Framework), but such an approach still requires some component in kernel mode, usually provided by the operating system, to handle hardware interactions.

NOTE

Please see Project #26 on page 230, where you can see loaded kernel modules, including device drivers, on Raspberry Pi OS.

Filesystems

Nearly every computer has some kind of secondary storage, usually a hard disk drive (HDD) or a solid-state drive (SSD). Such devices are effectively containers of bits that can be read and written, and where data persists even when the system is powered down. Storage devices are divided into regions called *partitions*. Operating systems implement *filesystems* to organize the data on storage devices into files and directories. A partition must be *formatted* with a particular filesystem before it can be used by the operating system. Different OSes use

different filesystems. Linux commonly uses the ext (extended) family of filesystems (ext2, ext3, ext4), whereas Windows uses FAT (File Allocation Table) and NTFS (NT File System). Some operating systems present storage as a *volume*, a logical abstraction built on one or more partitions. In such a system, filesystems reside on a volume rather than on a partition.

A *file* is a container of data, and a *directory* (also known as a folder) is a container of files or other directories. The contents of a file can be anything; the structure of the data stored within the file is determined by the program that wrote the file to storage. Unix-like systems organize their directory structure as a unified hierarchy of directories. The hierarchy starts at the root, designated with a single forward slash (/), and all other directories are descendants of the root. For example, library files are stored in */usr/lib*, where *usr* is a subdirectory of the root, and *lib* is a subdirectory of *usr*. This unified hierarchy applies even when there is more than one storage device on the system. Additional storage devices are mapped to a location in the directory structure; this is known as *mounting* a device. For example, a USB drive could be mounted to */mnt/usb1*.

In contrast, Microsoft Windows assigns a drive letter (A–Z) to each volume. So rather than a unified directory structure, each drive has its own root and hierarchy of directories. Windows uses a backslash (\) in its directory paths, and a colon (:) after a drive letter. For example, the Windows system files, stored on the C drive, are typically located under *C:\windows\system32*. This convention dates back to DOS (and earlier), when drives A and B were reserved for floppy disks, and drive C represented an internal hard drive. To this day, drive C is typically used as the drive letter for the volume where Windows is installed.

NOTE

Please see Project #27 on page 230, where you can check out the details of storage and files on Raspberry Pi OS.

Services and Daemons

Operating systems provide the ability for processes to automatically run in the background, without user interaction. Such processes are called *services* on Windows and *daemons* on Unix-like systems. A typical operating system includes a number of such services that run by default, such as a service to configure network settings, or a service that runs tasks on a schedule. Services are used to provide capabilities that aren't tied to a specific user, don't need to run in kernel mode, but do need to be available on demand.

Operating systems usually include a component responsible for managing services. Some services need to start when the OS boots; others need to run in response to a particular event. Often services should be restarted in the case of an unexpected failure. In Windows, the *Service Control Manager (SCM)* performs these types of functions. The SCM's executable file is *services.exe*, which is started early in the Windows boot process and continues to run as long as Windows itself is running. Many modern Linux distributions have adopted `systemd` as the standard component for managing daemons, although other mechanisms can be used in Linux to start and manage daemons. As discussed earlier, `systemd` also acts as the init process, so it's started very early in the Linux boot process and continues to run while the system is up.

The Unix and Linux term *daemon* comes from Maxwell's demon, a hypothetical being described in a physics thought experiment. This creature worked in the background, much like a computer daemon. Outside of computing, *daemon* is typically pronounced just like "demon," but when referring to background processes, "DAY-mon" is an equally acceptable pronunciation. Historically, *service* was a Windows-specific term, but now it is used on Linux as well, often to refer to daemons that are started by `systemd`.

NOTE

Please see Project #28 on page 231, where you can check out services on Raspberry Pi OS.

Security

An operating system provides a security model for the code that runs on that OS. In this context, *security* means that software, and users of that software, should only have access to appropriate parts of the system. This may not seem like a big deal for a personal device like a laptop or smartphone. If only one user logs into a system, shouldn't they have access to everything? Well, no, at least not by default. Users make mistakes, including running code that isn't trustworthy. If a user accidentally runs malicious software on their device, the OS can help limit the damage by restricting that user's access. On a shared system where multiple users log in, a user should not be able to read or modify another user's data, at least not by default.

Operating systems make use of multiple techniques to provide security. Let's look at just a few here. Simply putting applications in a user mode bubble goes a long way toward ensuring that software doesn't intentionally or accidentally mess with other applications or with the kernel. Operating systems also implement filesystem security, ensuring that data stored in files can only be accessed by appropriate users and processes. Virtual memory itself can be secured—regions of memory can be marked as read only or as executable, helping to limit misuse of memory. Providing a login system for users allows the operating system to manage security based on the user's identity. These are all baseline expectations of a modern operating system. Unfortunately, security vulnerabilities are regularly discovered in operating systems, allowing malicious actors to bypass the defenses of the OS. Keeping modern internet-connected operating systems up-to-date with the latest updates is critical to maintaining security.

Summary

In this chapter we covered operating systems, software that communicates with computer hardware and provides an environment for the execution of programs. You learned about the operating system

kernel, non-kernel components, and the separation of kernel mode and user mode. We reviewed the two dominant operating system families: Unix-like operating systems and Microsoft Windows. You learned that a program runs in a container known as a process, and multiple threads can execute in parallel within that process. We looked at various aspects of programmatically interacting with an operating system: the API, system calls, software libraries, and the ABI. In the next chapter, we'll move beyond single-device computing and examine the internet, looking at the various layers and protocols that make the internet possible.

PROJECT #20: EXAMINE RUNNING PROCESSES

Prerequisite: A Raspberry Pi, running Raspberry Pi OS. I recommend that you flip to Appendix B and read the entire “Raspberry Pi” section if you haven’t already.

In this project, you’ll look at processes running on a Raspberry Pi. The `ps` tool provides various views of running processes. Let’s begin with the following command, which provides a tree view of processes.

```
$ ps -eH
```

The output should look something like the following text. I’ve only reproduced a portion of it here.

```
 1 ?      00:00:10 systemd
 93 ?     00:00:09  systemd-journal
133 ?     00:00:01  systemd-udevd
233 ?     00:00:01  systemd-timesyn
274 ?     00:00:02    thd
275 ?     00:00:01    cron
276 ?     00:00:00  dbus-daemon
286 ?     00:00:03   rsyslogd
287 ?     00:00:01  systemd-logind
291 ?     00:00:08  avahi-daemon
296 ?     00:00:00      avahi-daemon
297 ?     00:00:01  dhcpcd
351 tty1  00:00:00    agetty
352 ?     00:00:00    agetty
358 ?     00:00:00    sshd
5016 ?    00:00:00      sshd
5033 ?    00:00:00      sshd
5036 pts/0  00:00:00      bash
5178 pts/0  00:00:00      ps
```

The indentation level indicates a parent/child relationship. For example, in the preceding output, we see that `systemd` is the parent of `systemd-journal`, `systemd-udevd`, and so forth. Or inversely, we can see that `ps` (the command currently running) is the child of `bash`, which is the child of `sshd`, and so forth.

The displayed columns are as follows:

PID The process ID

TTY The associated terminal

TIME The cumulative CPU time

CMD The executable name

The number of processes running may surprise you when you run `ps` in this way! The operating system handles many things, and as a result, it's normal for a large number of processes to run at any given time. Typically you see that the first process listed is PID 2, `kthreadd`. This is the parent of kernel threads, and the children you see listed under `kthreadd` are threads running in kernel mode. The other process to note is PID 1, the init process, the first user mode process that starts. In the preceding output, the init process is `systemd`. The Linux kernel starts both the init process and `kthreadd`, in that order, which ensures they are assigned PIDs 1 and 2, respectively.

Let's take a look at the init process. This is the first user mode process to start, and the specific executable that runs can vary on different versions of Linux. You can use `ps` to find the command used to start PID 1:

```
$ ps 1
```

You should see output like the following:

PID	TTY	STAT	TIME	COMMAND
1	?	Ss	0:03	/sbin/init

This tells you that the command used to kick off the init process was `/sbin/init`. So how does running `/sbin/init` result in `systemd` executing, as you saw in the earlier `ps` output? This happens because `/sbin/init` is actually a symbolic link to `systemd`. A *symbolic link* references another file or directory. You can see this with the following command:

```
$ stat /sbin/init
```

File: /sbin/init	-> /lib/systemd/systemd		
Size: 20	Blocks: 0	IO Block: 4096	symbolic link

In this output, you can see that `/sbin/init` is a symbolic link to `/lib/systemd/systemd`.

Another convenient view of the process tree can be generated by using the `pstree` tool, as mentioned earlier in this chapter. Running `pstree` presents a nicely formatted user mode process tree, starting with the init process. Give it a try:

```
$ pstree
```

Alternatively, if your Raspberry Pi is configured to boot to the desktop environment, you may also want to try the Task Manager application that's included with Raspberry Pi OS. It provides a graphical view of running processes, as shown in Figure 10-15.

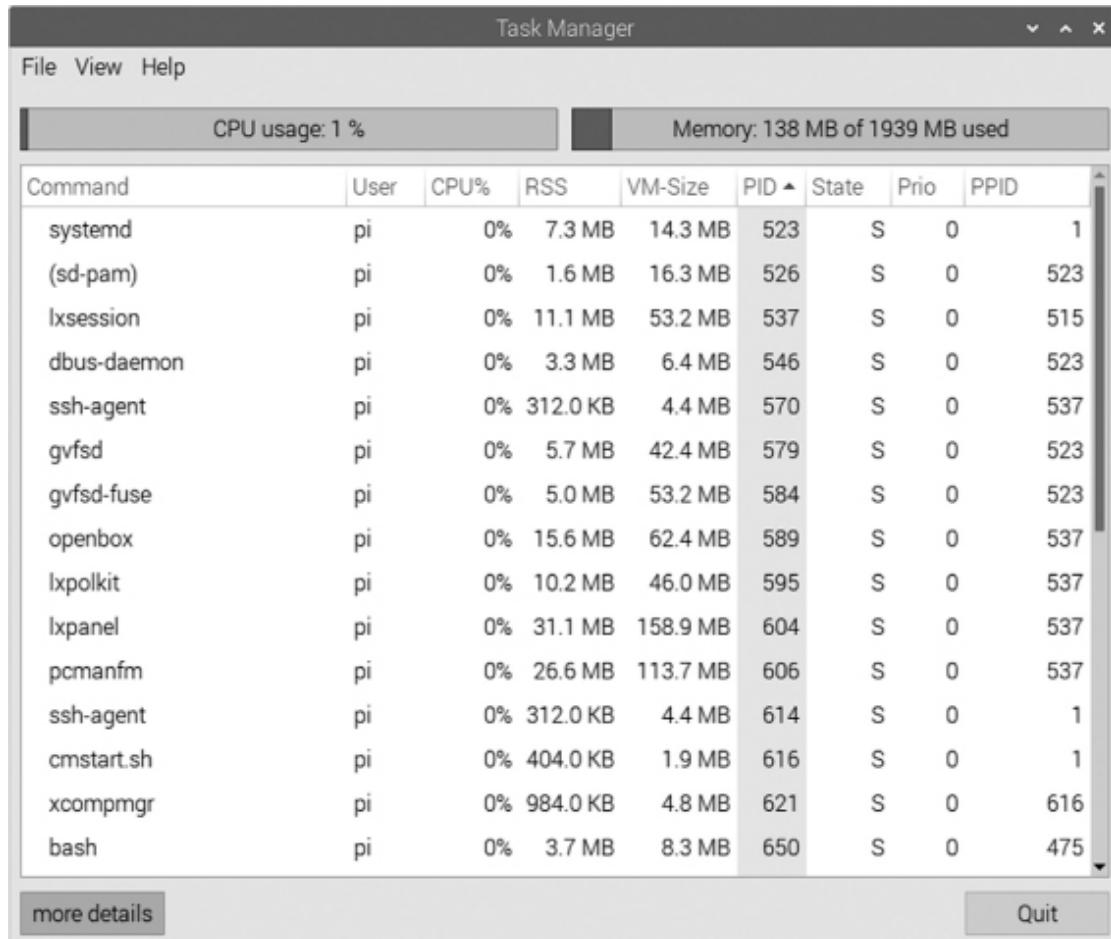


Figure 10-15: Task Manager in Raspberry Pi OS

PROJECT #21: CREATE A THREAD AND OBSERVE IT

Prerequisite: A Raspberry Pi, running Raspberry Pi OS. See “Raspberry Pi” on page 341.

In this project, you'll write a program that creates a thread. You'll then observe the thread running. Use the text editor of your choice to create a new file called *threader.c* in the root of your home folder. Enter the following C code into your text editor (you don't have to preserve indentation and empty lines, but be sure to maintain line breaks).

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>

void * mythread(void* arg)❶
{
    while(1)❷
    {
        printf("mythread PID: %d\n", (int)getpid());❸
        printf("mythread TID: %d\n", (int)syscall(SYS_gettid));
        sleep(5);❹
    }
}

int main()❺
{
    pthread_t thread;

    pthread_create(&thread, NULL, &mythread, NULL);❻

    while(1)❻
    {
        printf("main      PID: %d\n", (int)getpid());❼
        printf("main      TID: %d\n", (int)syscall(SYS_gettid));
        sleep(10);❽
    }

    return 0;
}
```

Before continuing, let's examine the source code. I won't go into all the details here, but in summary, the program starts in the `main` function ❺, which creates a thread ❻ that runs the function `mythread` ❶. This means there are two threads, the `main` thread and `mythread`. Both threads run in an infinite loop ❷❻, where every so often they print the PID and TID of the current thread ❸❼. For variety, `mythread` prints about every 5 seconds ❹, while `main` prints approximately every 10 seconds ❽. This helps illustrate that the threads are in fact running in parallel and doing work on their own schedule. Let's try it out.

Once the file is saved, use the GNU C Compiler (`gcc`) to compile your code into an executable file. The following command takes *threader.c* as an input and outputs an executable file named *threader*.

```
$ gcc -pthread -o threader threader.c
```

Now try running the code using the following command:

```
$ ./threader
```

The running program should output something like this, although the PID and TID numbers will be different:

```
main      PID: 2300
main      TID: 2300
mythread  PID: 2300
mythread  TID: 2301
```

As the program runs, expect the two threads to continue printing their PID and TID information. The TID and PID numbers won't change for this instance of the program, since it's the same process and threads running the entire time. You should see `mythread` print twice as often as `main`—every 5 seconds versus every 10 seconds.

Leave that program running and look at your list of running processes and threads. To do this, you need to open a second terminal window and run the following command (the | symbol can be entered with SHIFT-backslash right above ENTER, on US keyboards).

```
$ ps -e -T | grep threader
2300 2300 pts/0    00:00:00 threader
2300 2301 pts/0    00:00:00 threader
```

Adding the T option to the `ps` command shows threads as well as processes. The `grep` utility filters your output to only see the `threader` process information. In this output, the first column is the PID and the second column is the TID. So you can see that the output from `ps` matches the output from your program. The two threads share a PID but have different TIDs. Also, note that the `main` thread's TID matches its PID. This is expected for the first thread in a process.

To halt execution of the `threader` program, you can press CTRL-C in the terminal window where it's running. Or, from the second terminal window, you can use the `kill` utility, specifying the PID of the main thread, like so:

```
$ kill 2300
```

PROJECT #22: EXAMINE VIRTUAL MEMORY

Prerequisite: A Raspberry Pi, running Raspberry Pi OS. See “Raspberry Pi” on page 341.

In this project, you’ll examine virtual memory usage on Raspberry Pi OS. Let’s begin with a look at how address space is divided between kernel mode and user mode. This project assumes you are running the 32-bit version of Raspberry Pi OS, meaning there is 4GB of virtual address space. Linux allows for a split of that 4GB as either 2GB user and 2GB kernel, or 3GB user and 1GB kernel. Lower addresses are used for user mode, and higher addresses are used for kernel mode. That means that in a 2:2 split, kernel mode addresses start at 0x80000000, and in a 3:1 split, kernel mode addresses start at 0xC0000000. You can see the start of kernel mode address space with this command:

```
$ dmesg | grep lowmem
```

If the `dmesg` command does not produce any output, simply restart your Raspberry Pi and then run the `dmesg` command again. The command should produce output similar to the following.

```
lowmem : 0x80000000 - 0xbb400000 ( 948 MB)
```

If you’re wondering why you need to restart the Raspberry Pi if this command comes up empty, here’s some background information. The Linux kernel logs diagnostic messages to something called the kernel ring buffer, which the `dmesg` tool displays. The messages in the buffer are intended to give users some insight into the workings of the kernel. Only a limited number of messages are stored here; as newer messages are added, older messages are removed. The particular message we want to see (regarding `lowmem`) is written when the system starts, so if your system has been running for a while, it may have been overwritten. Restarting the system ensures that the message is written again.

As you can see, on my system, kernel `lowmem` starts at `0x80000000`, indicating a 2:2 split. This means that user mode processes can use addresses `0x00000000` to `0x7fffffff`. That range of addresses can reference 2GB of memory, and although the entire address space is available to every process, a typical process only actually needs to use a portion of that range. Certain addresses are mapped to physical memory, but others are left unmapped.

If your system returns a value of `0xc0000000` for the beginning of `lowmem`, then your system is running with a 3:1 split. This gives user mode processes 3GB of virtual address space, from `0x00000000` to `0xbfffffff`.

Let’s pick a process and examine its virtual memory usage. Raspberry Pi OS uses Bash as its default shell process, so if you’re working from a command line in Raspberry Pi OS, at least one instance of `bash` should be running. Let’s find the PID of a `bash` instance:

```
$ ps | grep bash
```

This should output text similar to the following:

```
2670 pts/0      00:00:00 bash
```

In my case, the PID of `bash` was `2670`. Now, run the following command to see the virtual memory mapping in the `bash` process. When you enter the command, be sure to replace `<pid>` with the PID returned on your system.

```
$ pmap <pid>
```

The output will be similar to the following, where each line represents a region of virtual memory in the process address space.

```
2670: -bash
00010000  872K r-x-- bash
000f9000    4K r---- bash
000fa000   20K rw--- bash
000ff000   36K rw--- [ anon ]
00ee7000 1044K rw--- [ anon ]
76b30000   36K r-x-- libnss_files-2.24.so
76b39000   60K ----- libnss_files-2.24.so
76b48000    4K r---- libnss_files-2.24.so
76b49000    4K rw--- libnss_files-2.24.so
...
7ec2c000   132K rw--- [ stack ]
7ec74000    4K r-x-- [ anon ]
7ec75000    4K r---- [ anon ]
7ec76000    4K r-x-- [ anon ]
fffff0000    4K r-x-- [ anon ]
total     6052K
```

The first column is the region start address, the second column is the size of the region, the third column represents the permissions of the region (`r` = read, `w` = write, `x` = execute, `p` = private, `s` = shared), and the final column is the region name. The region name is either a filename or a name that identifies the memory region if it isn't mapped from a file.

You can see that almost every region in the output is within the expected user mode range of `0x00000000` to `0x7fffffff`. The one exception is the last entry, which corresponds to the ARM CPU vector page, and represents a special case, as it's outside the standard user mode address range. As you can see in the preceding output, this particular instance of `bash` only has a **total** of `6052K` (about 6MB) of virtual memory mapped out of a possible 2GB, or around 0.3 percent.

PROJECT #23: TRY THE OPERATING SYSTEM API

Prerequisite: A Raspberry Pi, running the Raspberry Pi OS.

In this project, you'll try invoking the operating system API in various ways. You'll specifically focus on creating a file and writing some text to it. Use the text editor of your choice to create a file called *newfile.c* in the root of your home folder. Enter the following C code into your text editor.

```
#include <fcntl.h>
#include <unistd.h>

#define msg "Hello, file!\n"❶

int main()❷
{
    int fd;❸
    fd = open("file1.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);❹
    write(fd, msg, sizeof(msg) - 1);❺
    close(fd);❻
    return 0;❼
}
```

Before we continue, let's examine the source code to understand exactly what it does. In short, the program uses three API functions, `open`, `write`, and `close`, to create a new file, write some text to it, and finally close the file. Our focus here is to see how the operating system's API allows a program to interact with the computer's hardware, specifically a storage device. Let's go through the program in more detail.

After the requisite include statements, the next line defines `msg` as a text string ❶ that later will be written to the newly created file. The code then defines `main`, the entry point of the program ❷. Within `main`, an integer named `fd` is declared ❸. Next, the OS API `open` function is called to create a new file named *file1.txt* ❹. The other arguments passed to the `open` function specify the details of how the file should be opened. For simplicity, I won't cover those details here, but feel free to research the meanings of these arguments. The `open` function returns a file descriptor, which is saved in the `fd` variable.

The `write` function is then used to write the `msg` text to *file1.txt* (identified by the file descriptor stored in `fd`) ❺. The `write` function requires inputs of both the data to write (`msg`) and the number of bytes to write, determined by `sizeof(msg) - 1`. You subtract 1 because the C language terminates strings with a null character, and you don't need to write that byte to the output file. The program is now finished working with the file and calls the `close` function on the file descriptor to indicate that the file is no longer in use ❻. Finally, the program exits with a return code of 0 ❼, indicating success.

Once the file is saved, use the GNU C Compiler (`gcc`) to compile the code into an executable file. The command below takes *newfile.c* as an input and generates an executable file named *newfile*.

```
$ gcc -o newfile newfile.c
```

Now try running the code using the following command. You won't see any output, since the text is written to a file rather than the terminal.

```
$ ./newfile
```

To determine if the program ran successfully, you need to see if a file was created. The file should be named *file1.txt* and exist in your current directory. You can use the `ls` command to list the contents of the current directory and look for the file. Assuming *file1.txt* is present, you can see its contents using the `cat` command.

```
$ ls  
$ cat file1.txt
```

This command should print `Hello, file!` to the terminal, since that's the text the program wrote to the file. Or you can view the file's properties in the File Manager application of the Raspberry Pi OS desktop, and you can open *file1.txt* in your text editor of choice.

When you use the C programming language you get a look at specifics of the OS API functions, since `open`, `write`, and `close` are defined as C functions. However, you aren't limited to C when interacting with the OS. Other languages provide their own layer on top of the API, hiding some of the complexity from software developers. To illustrate this, let's write an equivalent program in Python.

Use the text editor of your choice to create a file called *newfile.py* in the root of your home folder. Enter the following Python code into your text editor.

```
f = open('file2.txt', 'w')❶  
f.write('Hello from Python!\n');❷  
f.close()
```

Before continuing, let's examine the source code. This program effectively does the same thing as the previous program, except the output filename is different (*file2.txt*) ❶, and the text written to that file is also different ❷. In this case, Python happens to use the same names as the OS API (`open`, `write`, `close`), but these are not direct calls to the operating system; rather, they are calls into the Python standard library.

Once you've saved this code, you can run it. Remember that Python is an interpreted language, so rather than compiling your Python code, just run it using the Python interpreter, like so:

```
$ python3 newfile.py
```

To determine if the program ran successfully, you need to see if *file2.txt* was created with the expected contents. You can again use `ls` and `cat` to verify this, or you can look in the desktop File Manager to see the file.

```
$ ls  
$ cat file2.txt
```

Although it may seem as if you're just leveraging Python's capabilities to manipulate a file, keep in mind that Python cannot do this on its own. The Python interpreter is making system API calls on your behalf when it runs your code. You'll get to observe this in the next project.

PROJECT #24: OBSERVE SYSTEM CALLS

Prerequisite: Complete Project #23.

In this project, you'll observe the system calls made by the programs you wrote in Project #23. To do this you'll use a tool called `strace`, which traces system calls and prints the output to the terminal.

Open a terminal on your Raspberry Pi and use `strace` to run the `newfile` program you previously wrote in C and compiled:

```
$ strace ./newfile
```

The `strace` tool launches a program (`newfile` in this case) and shows all the system calls that are made while that program runs. At the beginning of the output, you can see a number of system calls that represent the work required to load the executable file and required libraries. This is work that happens before the code you wrote runs; you can skip past that text. Near the end of the output, you should see text similar to the following:

```
openat(AT_FDCWD, "file1.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) = 3  
write(3, "Hello, file!\n", 13)           = 13  
close(3)                            = 0
```

This should look familiar; it's almost the same three API functions that you used to create `file1.txt` and write text to it. The C functions that you called from your program are just thin wrappers around the system calls of the same name, with the exception of `open`, which invokes the `openat` system call. The values after the equals signs are the return values from the three system calls. On my system, the `openat` function returned `3`, which is a number known as a *file descriptor* that refers to the opened file. You can see the file descriptor value used as a parameter to the subsequent calls to `write` and `close`. The `write` function returned `13`, the number of bytes written. The `close` function returned `0`, an indicator of success.

Now use the same approach to also check out the system calls made from the Python program you wrote in Project #23.

\$ strace python3 newfile.py

Expect to see even more output here, since **strace** is actually monitoring the Python interpreter, which in turn has to load *newfile.py* and run it. If you look near the end of the output, you should see calls to **openat**, **write**, and **close**, just as you did in the C program. This shows that despite the source code differences between C and Python, in the end, the same system calls are invoked to interact with files.

The **strace** tool can be used to quickly get an idea of how a program interacts with the operating system. For example, earlier in this chapter, we used the **ps** utility to get a list of processes. If you want to understand how **ps** works, you can run **ps** under **strace**, like so:

\$ strace ps

Look at the output from this command to see what system calls **ps** makes.

PROJECT #25: USE GLIBC

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll write code to use the C library and examine the details of how this works. Use the text editor of your choice to create a new file called *random.c* in the root of your home folder. Enter the following C code into your text editor.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    srand(time(0));❶
    printf("%d\n", rand());❷
    return 0;
}
```

This little program simply prints a random integer value to the terminal. The first thing the program does is call the **srand** function to seed the random number generator ❶, a necessary step to ensure a unique sequence of numbers is generated. The current time, as returned from the **time** function, is used as the seed value. The next line prints

out a random value returned from the `rand` function ❷. To accomplish all of this, the program uses four functions from the C library (`time`, `srand`, `rand`, and `printf`).

Once the file is saved, you can use the GNU C Compiler (`gcc`) to compile the code into an executable file. The following command takes `random.c` as an input and outputs an executable file named `random`.

```
$ gcc -o random random.c
```

Now try running the code using the following command. The program should output a random number. Run it multiple times to confirm that it outputs different numbers. However, quickly running it twice may produce the same result, since the seed value returned from the `time` function only increments every second.

```
$ ./random
```

Once you've ensured that the program works, look at the libraries that the program imports. One way to do this is to run the `readelf` utility, like so:

```
$ readelf -d random | grep NEEDED
```

Look for the `NEEDED` sections in the output, like the following:

0x00000001 (NEEDED)	Shared library: [libc.so.6]
---------------------	-----------------------------

This tells you that the `libc.so.6` library is required for this program to run. This is expected, as this is the GNU C Library (also known as `glibc`). In other words, because the program relies on functions in the C standard library, the operating system must load the `libc.so.6` library file so that the library code is available. This is a good start, but what if you want to see the specific list of functions that the `random` program uses from this library? You can observe this with the following:

```
$ objdump -TC random
```

This gives you output like the following:

random:	file format elf32-littlearm	
DYNAMIC SYMBOL TABLE:		
00000000	w D *UND* 00000000	__gmon_start__
00000000	DF *UND* 00000000 GLIBC_2.4	srand
00000000	DF *UND* 00000000 GLIBC_2.4	rand
00000000	DF *UND* 00000000 GLIBC_2.4	printf
00000000	DF *UND* 00000000 GLIBC_2.4	time
00000000	DF *UND* 00000000 GLIBC_2.4	abort
00000000	DF *UND* 00000000 GLIBC_2.4	__libc_start_main

In the preceding output, in the rightmost column, you can see the expected four functions (`srand`, `rand`, `printf`, and `time`) along with some additional functions.

Now that you've established which `glibc` functions were imported by your `random` program, you may wish to see the list of all functions that are exported by `glibc`. These are the functions this library makes available for programs to use. You can get this information with the following command:

```
$ objdump -TC /lib/arm-linux-gnueabihf/libc.so.6
```

Sometimes it's useful to see information about loaded libraries while you're debugging a running process. Let's try that by debugging the `random` program. To start, enter the following command:

```
$ gdb random
```

At this point `gdb` has loaded the file but no instructions have run yet. From the (`gdb`) prompt, enter the following to start running the program. The debugger halts execution once it reaches the beginning of the `main` function.

```
(gdb) start
```

Look at the loaded shared libraries:

```
(gdb) info sharedlibrary
From          To            Syms Read  Shared Object Library
0x76fce30  0x76fea150  Yes           /lib/ld-linux-armhf.so.3①
0x76fb93ac  0x76fb300   Yes (*)      /usr/lib/arm-linux-gnueabihf/libarmmem-
v71.so②
0x76e6e050  0x76f702b4  Yes           /lib/arm-linux-gnueabihf/libc.so.6③
(*): Shared library is missing debugging information.
```

The first library, `ld-linux-armhf.so.3` ①, is the Linux dynamic linker library. It's responsible for loading other libraries. Linux ELF binaries are compiled to use a specific linker library; this information is in the ELF header of the compiled program. You can find the linker library for the `random` program using the following command from a terminal window (not in `gdb`):

```
$ readelf -l random | grep interpreter
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
```

As you can see in the preceding output, the linker library specified for the `random` program is `ld-linux-armhf.so.3`, the same dynamic linker library we just discussed.

Take a look back at the `info sharedlibrary` output in `gdb`; you can see that the second library listed is `libarmmem-v71.so` ②. This library is specified in the file

/etc/ld.so.preload, a text file that lists libraries that load for every program that's executed on the system.

Now move on to the third library, which is the one of interest, *libc.so.6* ❸, the GNU C Library (*glibc*). In the `readelf` and `objdump` output earlier you saw that this library was imported by the executable file, and here you can see that it did indeed successfully load while running. You can also see the specific address range where it loaded (`0x76e6e050` to `0x76f702b4`), and the specific directory path from which it loaded.

You can exit the debugger at any time by typing `quit` in `gdb`.

PROJECT #26: VIEW LOADED KERNEL MODULES

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll take a look at the loaded kernel modules, including device drivers, on Raspberry Pi OS. Device drivers are typically implemented as kernel modules on Linux, although not all kernel modules are device drivers. To list the loaded modules, you can either examine the contents of the */proc/modules* file or use the `lsmod` tool like so:

```
$ lsmod
```

To view more details about a specific module, use the `modinfo` utility like so (using the `snd` module as an example):

```
$ modinfo snd
```

PROJECT #27: EXAMINE STORAGE DEVICES AND FILESYSTEMS

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll take a look at storage devices and filesystems. Let's begin by listing the block devices, which is how Linux characterizes storage devices.

```
$ lsblk
NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
mmcblk0    179:0    0 29.8G  0 disk❶
```

```
| -mmcblk0p1 179:1      0  256M  0 part /boot②  
| _mmcblk0p2 179:2      0 29.6G  0 part /③
```

Here we see a single “disk” named `mmcblk0` ①, which is the microSD card in the Raspberry Pi. You can see that it’s divided into two partitions of varying sizes. Partition 1 is mapped to the `/boot` directory in the unified directory structure ②, while partition 2 is mapped to the root (/) ③.

Now take a look at the overall usage of the storage device using the `df` command:

```
$ df -h -T  
Filesystem  Type      Size  Used Avail Use% Mounted on  
/dev/root   ext4      30G  3.0G  25G  11% /①  
devtmpfs    devtmpfs  459M   0  459M  0% /dev  
tmpfs       tmpfs     464M   0  464M  0% /dev/shm  
tmpfs       tmpfs     464M   6.3M  457M  2% /run  
tmpfs       tmpfs     5.0M  4.0K  5.0M  1% /run/lock  
tmpfs       tmpfs     464M   0  464M  0% /sys/fs/cgroup  
/dev/mmcblk0p1 vfat     253M  52M  202M  21% /boot②  
tmpfs       tmpfs     93M   0   93M  0% /run/user/1000
```

This command lets you view the various mounted filesystems, their size, and how full they are. Only the `root` ① and `/boot` ② directories are mapped to storage devices. The others are temporary filesystems that reside in memory, not a persistent storage device.

You can get a view of the directories on your system by running the `tree` command. The parameters used here limit your output to directories only, and only go three levels deep in the hierarchy.

```
$ tree -d -L 3 /
```

You can also see a similar view from the desktop environment using the File Manager application.

PROJECT #28: VIEW SERVICES

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you’ll look at services/daemons. Raspberry Pi OS uses the `systemd` init system, and it includes a utility called `systemctl` that you can use to see the state of services:

```
$ systemctl list-units --type=service --state=running
```

This should produce output similar to the following:

UNIT	LOAD	ACTIVE	SUB	DESCRIPTION
avahi-daemon.service	loaded	active	running	Avahi mDNS/DNS-SD Stack
bluealsa.service	loaded	active	running	BluezALSA proxy
bluetooth.service	loaded	active	running	Bluetooth service
cron.service	loaded	active	running	Regular background ...
dbus.service	loaded	active	running	D-Bus System Message Bus
dhpcd.service	loaded	active	running	dhpcd on all interfaces
getty@tty1.service	loaded	active	running	Getty on tty1
hciuart.service	loaded	active	running	Configure Bluetooth Modems
...				
rsyslog.service	loaded	active	running	System Logging Service
ssh.service	loaded	active	running	OpenBSD Secure Shell server
systemd-journald.service	loaded	active	running	Journal Service
systemd-logind.service	loaded	active	running	Login Service
systemd-timesyncd.service	loaded	active	running	Network Time Synchronization
systemd-udevd.service	loaded	active	running	udev Kernel Device Manager
triggerhappy.service	loaded	active	running	triggerhappy global hotkey daemon
user@1000.service	loaded	active	running	User Manager for UID 1000

If you aren't automatically returned to a terminal prompt, hit Q in your terminal to exit the view of services. To see the details of a particular service, try this command, using `cron.service` as an example:

```
$ systemctl status cron.service
```

The output of this command includes the path and PID of the process that's associated with the service. In the case of `cron.service`, the path on my system is `/usr/sbin/cron`, and it happened to be PID 367.

Another approach to viewing daemon processes is to view all the processes that are children of `systemd`, that is, PID 1. This is relevant since services are started by `systemd` and appear as child processes of PID 1. Note that this output may include more than just services/daemons, since orphaned processes are adopted by PID 1.

```
$ ps --ppid 1
```

11

THE INTERNET



So far, we've focused on computing that occurs on a single device. In this chapter and the next, we look at computing that spans multiple devices. We're going to examine two significant innovations in computing, the internet and the world wide web, which are not the same thing! This chapter focuses on the internet, and we begin by defining key terms. Then we look at a layered model of networks and dig into some of the foundational protocols used on the internet.

Networking Terms Defined

To discuss the internet and networks in general, you first need to become familiar with some concepts and terms, which we cover here. A *computer network* is a system that allows computing devices to communicate with each other, as illustrated in Figure 11-1. Networks can be connected wirelessly, using technologies like Wi-Fi, which transmits data using radio waves. Networks can also be connected with cables, such as copper wiring or fiber optics. Computing devices on a network must use a common *communications protocol*, a set of rules that describe how information is to be exchanged.

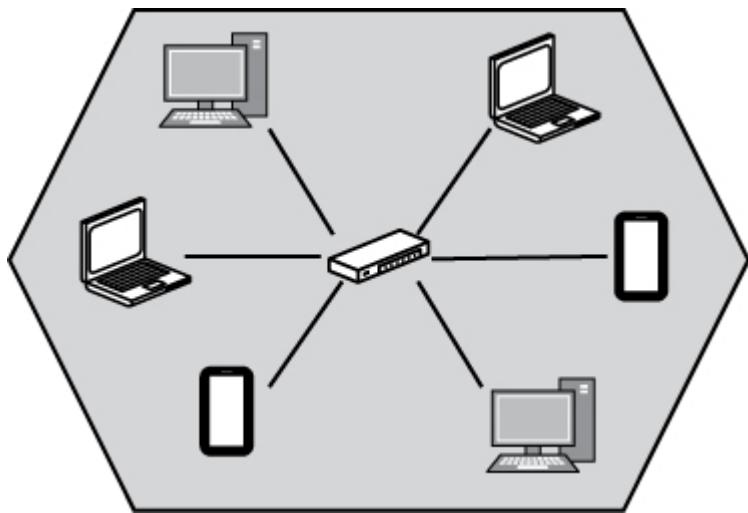


Figure 11-1: A computer network

The *internet* is a globally connected set of computer networks that all use a suite of common protocols. The internet is a *network of networks*, connecting networks from various organizations all around the world, as shown in Figure 11-2.

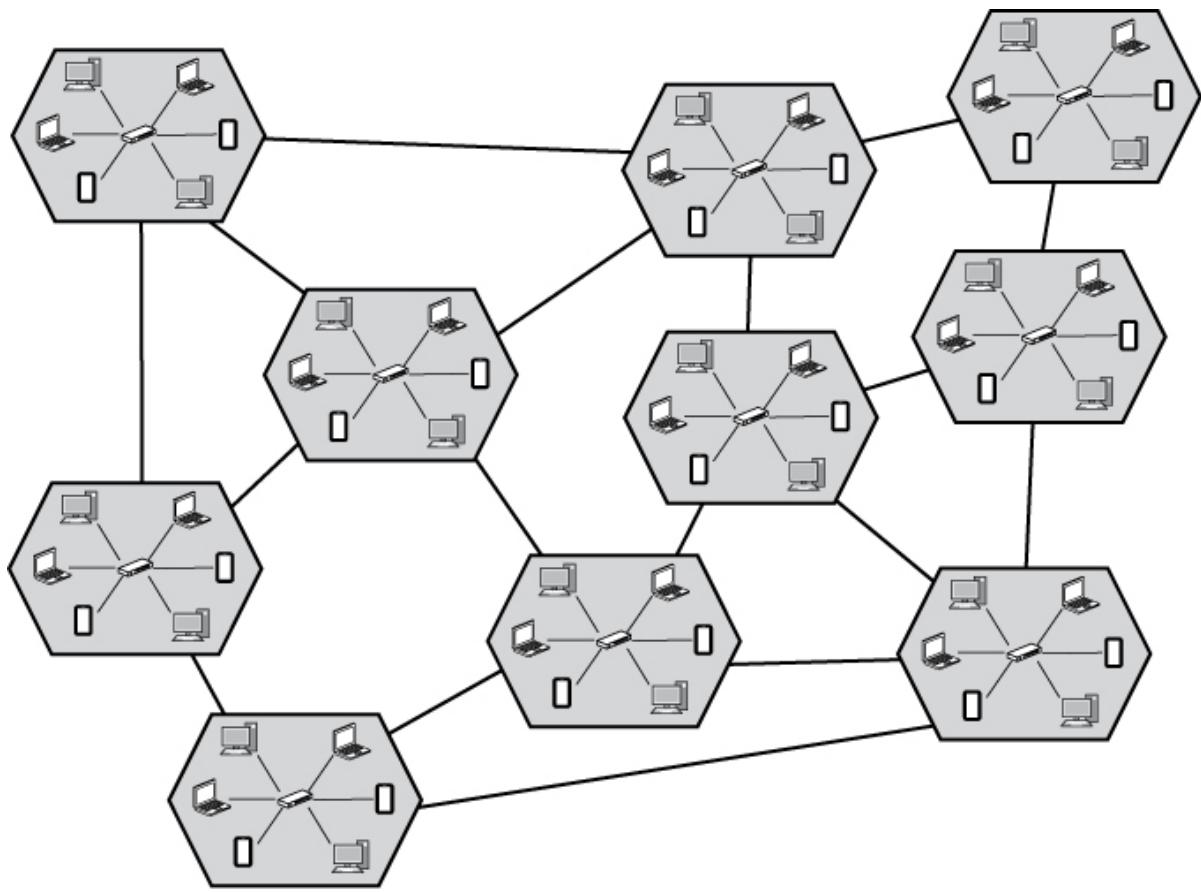


Figure 11-2: The internet: a network of networks

A *host* or *node* is a single computing device attached to a network. A host can act as a server or a client on the network, or sometimes both. A network *server* is a host that listens for inbound network connections and provides services to other hosts. Examples are a web server and an email server. A network *client* is a host that makes outbound connections and requests services from network servers. Example clients are smartphones or laptops running web browsers or email apps. A client makes a *request* to a server, and the server replies with a *response*, as illustrated in Figure 11-3.

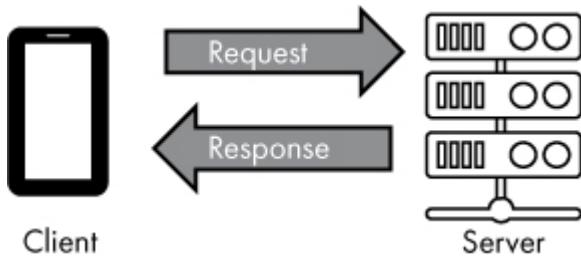


Figure 11-3: A client makes a request to a server, and the server responds

The term *server*, as just used, refers to any device that accepts inbound requests and provides services to clients. However, *server* can also refer to a class of computer hardware that's specifically intended to act as a network server. These specialized computers are physically designed to be mounted into racks of computers in a datacenter and often include hardware redundancy and management capabilities not found in a typical PC. However, any device with the right software can act as a server on a network.

The Internet Protocol Suite

Physically connecting the networks of the world isn't enough to allow the devices on those networks to communicate with each other. All participating computers need to communicate in the same way. The *internet protocol suite* standardizes the method of communication on the internet, ensuring that all devices on the network speak the same language. The two foundational protocols in the internet protocol suite are *Transmission Control Protocol (TCP)* and *Internet Protocol (IP)*, collectively known as *TCP/IP*.

Network protocols operate in a layered model, and an implementation of such a model is referred to as a *network stack* (not to be confused with a stack in memory, as covered in Chapter 9). The protocols at the lowest layer interact with the underlying networking hardware, whereas applications interact with protocols in the upper layers. Protocols in the intermediate layers provide services such as addressing and reliable delivery of data. A protocol at a certain layer

doesn't have to concern itself with the entire networking stack, only the layers with which it interfaces, simplifying the overall design. This is another example of encapsulation.

The internet protocol suite is designed around a four-layer model. This is sometimes called the *TCP/IP model*. The four layers of the TCP/IP model, starting from the bottom up, are the link layer, the internet layer, the transport layer, and the application layer, as shown in Figure 11-4.

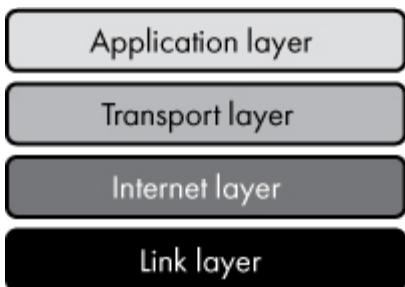


Figure 11-4: The internet protocol suite model of networking

OSI—ANOTHER NETWORK MODEL

Another commonly used model for network protocols is the *Open Systems Interconnection (OSI) model*. The OSI model divides protocols into seven layers rather than four. This model is often referenced in technical literature, but the internet is based on the internet protocol suite, so this book focuses on the TCP/IP model.

These networking layers represent an abstraction, a model for us to use when discussing the operation of the internet. In practice, each layer is realized with specific networking protocols. Each network layer represents a scope of responsibilities, and protocols must fulfill the responsibilities of their assigned layer. Table 11-1 provides a description of each layer.

Table 11-1: Description of the Four Layers of the Internet Protocol Suite

Layer	Description	Example protocols

Layer	Description	Example protocols
Application	<p>Protocols that operate at the application layer provide application-specific functionality, such as sending an email or retrieving a web page. These protocols accomplish tasks that end users (or backend services) wish to complete.</p> <p>Application layer protocols structure the data used in process-to-process communication across a network.</p> <p>All the lower layer protocols exist as “plumbing” to support the application layer.</p>	HTTP, SSH

Layer	Description	Example protocols
Transport	Transport layer protocols provide a communications channel for applications to send and receive data between hosts. An application structures data according to an application layer protocol and then hands off that data to a transport layer protocol for delivery to a remote host.	TCP, UDP

Layer	Description	Example protocols
Internet	<p>Internet layer protocols provide a mechanism for communicating across networks.</p> <p>This layer is responsible for identifying hosts with addresses and enabling the routing of data from network to network across the internet. The transport layer relies on the internet layer for addressing and routing.</p>	IP

Layer	Description	Example protocols
Link	<p>Link layer protocols provide a way to communicate on a local network. Protocols at this layer are closely associated with the type of networking hardware on a local network, such as Wi-Fi. Protocols at the internet layer rely on link layer protocols to communicate on a local network.</p>	Wi-Fi, Ethernet

Protocols at each layer communicate with the protocols in adjacent layers. An outgoing transmission from a host travels down through the network layers, from an application layer protocol, to a transport layer protocol, to an internet layer protocol, and finally to a link layer protocol. An incoming transmission to a host travels up through the network layers, reversing the order just described.

Although network hosts (such as a client or server) make use of protocols from all four layers, other types of networking hardware (such as switches and routers) only use protocols associated with lower layers. Such devices can perform their jobs without bothering to examine the higher layer protocol data contained in a network transmission.

An outgoing request from a client to a server, and its relationship to the networking layers, is illustrated in Figure 11-5.

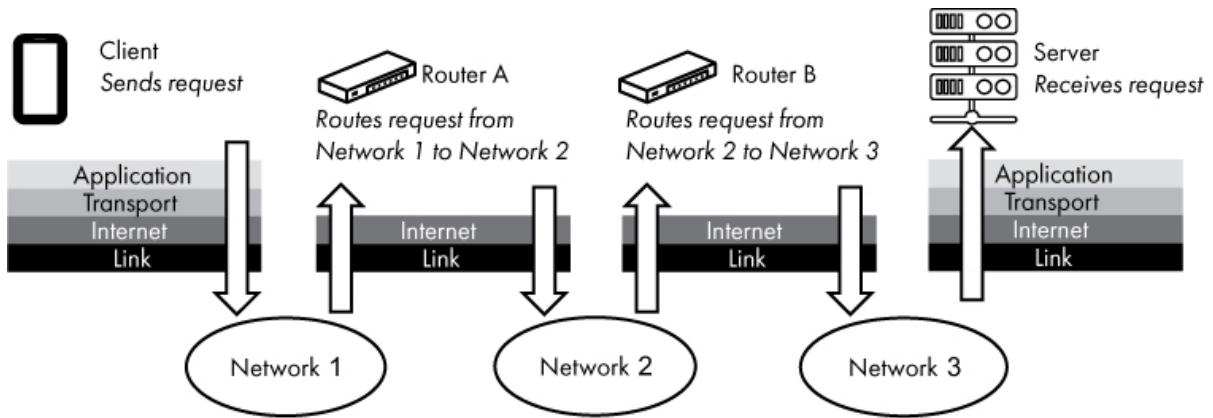


Figure 11-5: A network request travels through various network layers

Let's walk through the flow of Figure 11-5. An application on the client device forms a request using an application layer protocol. That request is handed off to a transport layer protocol, then to an internet layer protocol, and finally to a link layer protocol. All of this happens on the client device. At this point the request is transmitted onto the local network, labeled Network 1 in the diagram. The request makes its way across the internet, going from network to network. In this example, Router A routes the request from Network 1 to Network 2, and Router B routes the request from Network 2 to Network 3. Once the request reaches the destination server, it works its way up through the networking protocols, starting with a link layer protocol, and ending at an application layer protocol. A process running on the server receives the request, which is formatted according to the application layer protocol originally used by the client. The server process interprets the request and responds in an appropriate manner.

Let's now take a look at each layer, starting from the bottom.

Link Layer

The lowest level of the internet protocol suite is the *link layer*. The physical and logical connections between hosts are known as network *links*. Link layer protocols are used by devices on the same network to communicate with each other. Each device on a link has a network address that uniquely identifies it. For many link layer protocols, this

address is known as a *media access control address* (or *MAC address*). Link layer data is divided into small units known as *frames*, each including a header describing the frame, a payload of data, and finally, a frame footer used to detect errors. This is illustrated in Figure 11-6.



Figure 11-6: A link layer frame

The frame header contains source and destination MAC addresses. The header also includes a descriptor of the type of data carried in the frame data section.

If your home has a Wi-Fi network, Wi-Fi is the link between the hosts on your network. The Wi-Fi protocol, defined by the IEEE 802.11 specifications, doesn't know or care what type of data is being sent over the wireless network; it simply enables communication between devices. Each device connected to the Wi-Fi network has a MAC address and receives frames sent to its address. MAC addresses are only useable on a local network; a computer on a remote network cannot directly send data to a MAC address on your local network.

Another notable link layer technology is *Ethernet*, used for wired physical connections. Ethernet is defined by the IEEE 802.3 standards. Ethernet typically uses a cable with pairs of copper wires inside that ends in a connector commonly known as *RJ45*, shown in Figure 11-7.

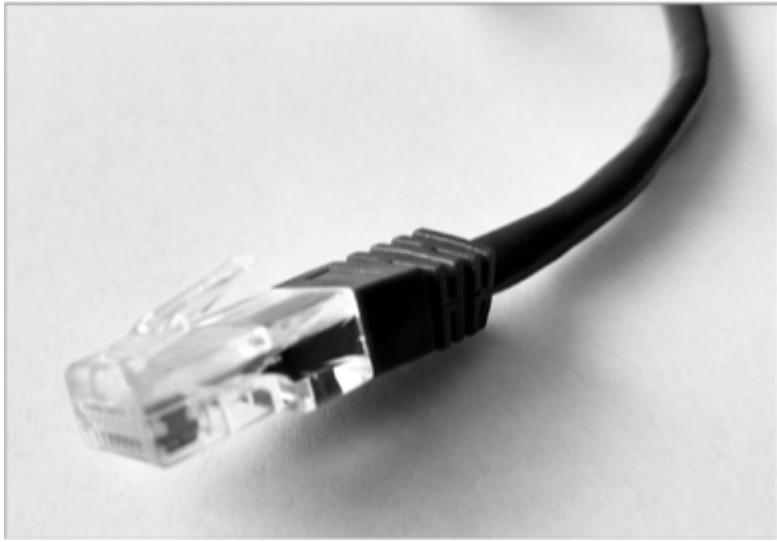


Figure 11-7: The cable commonly used for Ethernet

All devices connected to the internet participate in the link layer. This is required, since it's the link layer that provides connectivity (either wired or wireless) to a local network. A host, like a laptop or smartphone, participates in all layers, but certain networking devices operate at the link layer only. The most basic example of this is a hub. A *network hub* is a networking device that connects multiple devices on a local network without any intelligence regarding the frames being sent. A simple hub might provide multiple Ethernet ports for connecting devices. The hub simply retransmits every frame it receives on one physical port to all its other ports. A more intelligent link layer device is a *network switch*, which examines the MAC addresses in the frames it receives and sends those frames to the physical port where the device with the destination MAC address is connected.

NOTE

Please see Project #29 on page 254, where you can look at link layer devices and MAC addresses.

Internet Layer

The *internet layer* allows data to travel beyond the local network. The primary protocol used in this layer is simply called Internet Protocol (IP). It enables *routing*, the process of determining a path for data that's

transmitted between networks. Every host on the internet is assigned an *IP address*, a number that uniquely identifies the host on the global internet. It's also possible to have private IP addresses that aren't directly exposed on the internet. IP addresses are usually assigned by a server on the local network, and a device's IP address typically changes when it connects to a new network. We'll cover more on address assignment and private IP addresses later.

Data sent over the internet layer is called a *packet*, which is enclosed in a link layer frame. Figure 11-8 illustrates the idea that a packet fits within a frame's data section.

The IP packet header contains a source IP address and a destination IP address. The header also includes information that describes the packet, such as the IP version in use and the header length. The data section of the IP packet contains the payload that the IP layer is carrying.

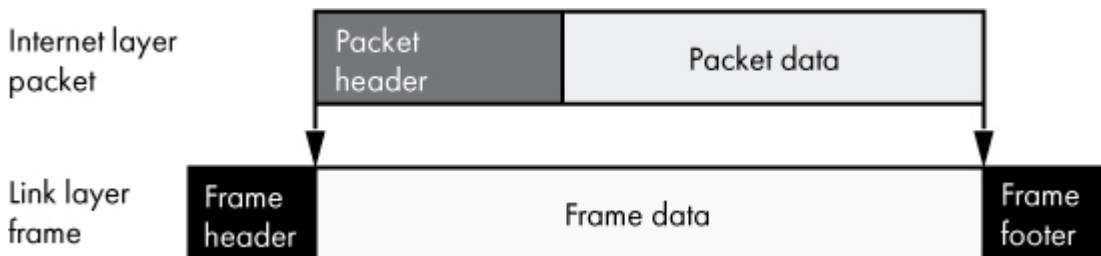


Figure 11-8: A packet is contained in the data section of a frame

Two versions of Internet Protocol are in use on the internet today. *Internet Protocol Version 4 (IPv4)* is the dominant version in use, and the other active version is *Internet Protocol Version 6 (IPv6)*. You may wonder what happened to IPv5. No such protocol ever existed, but an experimental protocol called Internet Stream Protocol identified its IP version as 5, and so IPv5 was skipped when the successor to IPv4 was developed. A significant difference between IPv4 and IPv6 is the size of an IP address. An IPv4 address is 32 bits in length, whereas an IPv6 address is 128 bits. This difference allows for a vastly larger number of addresses with IPv6. This change in address size is meant to help deal with the relatively short supply of IPv4 addresses. In this book, we focus

on IPv4 addresses (and just refer to them as *IP addresses*), as they are still the primary means of addressing on the internet today.

A 32-bit IP address is typically displayed in dotted decimal notation, meaning the 32 bits are separated into four groups of 8 bits each, the 8-bit numbers are displayed in decimal (rather than hexadecimal or binary), and the four decimal numbers are separated by periods (dots). An example IP address, displayed in dotted decimal notation, is 192.168.1.23. Each 8-bit decimal number can be referred to as an *octet*.

Computers connected to the same local network have IP addresses that begin with the same leading bits and are said to be on the same *subnet*. Computers that are on the same subnet are able to communicate directly with each other at the link layer because they are operating on the same physical network. Computers that are on different subnets must send their traffic through a *router*, a device that connects subnets and operates at the internet layer.

Subnetting divides the IP address into two parts: the *network prefix*, which all devices on the same subnet share, and the *host identifier*, which is unique to a host on that subnet. The number of bits included in the network prefix varies based on the network configuration.

Let's look at an example. Assume a subnet uses a 24-bit network prefix, leaving us with 8 bits to represent the host. Also assume that a host on this subnet uses the example IP address from earlier—192.168.1.23. Given this IP address and network prefix, the IP address is divided as shown in Figure 11-9.

In this example, all hosts on the local subnet have an IP address that begins with 192.168.1. Each host has a different value for the last octet, with 23 being assigned to this specific host. This example uses a 24-bit prefix length, meaning the prefix neatly aligns with the first three octets of the IP address. This makes for a nice example, but the prefix length doesn't always align with an octet boundary. A 25-bit prefix, for example, would also include the first bit of the last octet, leaving only 7 bits for identifying the host.

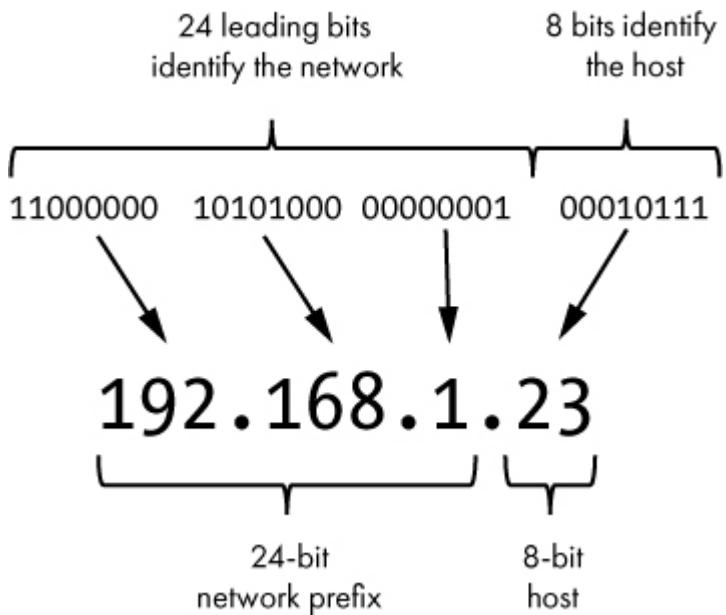


Figure 11-9: An example IP address using a 24-bit network prefix

The number of bits reserved for the network prefix is commonly expressed in one of two ways. *Classless Inter-Domain Routing (CIDR)* notation lists an IP address followed by a slash (/), and then the number of bits used for the network prefix. In our example this would be 192.168.1.23/24. Another common way to represent the number of prefix bits is with a *subnet mask*, a 32-bit number where a binary 1 is used for each bit that's part of the network prefix and a 0 is used for each bit that's part of the host number. Subnet masks are also written in dotted decimal notation, so our example of a 24-bit network prefix would result in a subnet mask of 255.255.255.0, as shown in Figure 11-10.

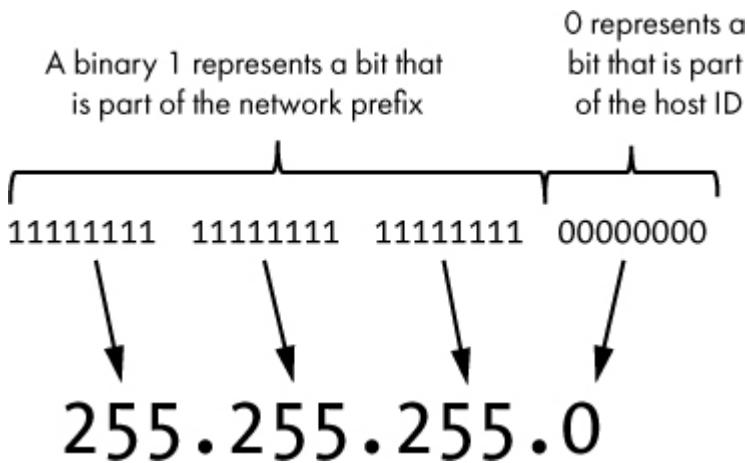


Figure 11-10: A 24-bit network prefix expressed as a subnet mask

Let's look at how this is useful in practice. Say your computer has an IP address of 192.168.0.133 and a subnet mask of 255.255.255.224, or, expressed in CIDR notation, 192.168.0.133/27. Your computer wishes to connect to another computer with an IP address of 192.168.0.84. As mentioned earlier, two computers can communicate directly if they are on the same subnet, and if not, they must go through a router. So your computer must determine if the other computer is on the same subnet. How can it do this?

Performing a bitwise logical AND of an IP address and its subnet mask produces the first address in a subnet. This first address, where the host bits are all 0, serves as an identifier for the subnet itself. This is commonly referred to as the *network ID*. Two computers that share a network ID are on the same subnet. A host can perform this AND operation against both its own IP address and the IP address it wishes to connect to, to see if they share a network ID and thus are on the same subnet. Let's try this with our example computer's IP address, as shown here:

IP = 192.168.0.133	= 11000000.10101000.00000000.10000101
MASK = 255.255.255.224	= 11111111.11111111.11111111.11100000
AND = 192.168.0.128	= 11000000.10101000.00000000.10000000 = The network ID

Now perform the same operation for the second computer in our example:

IP = 192.168.0.84	= 11000000.10101000.00000000.01010100
MASK = 255.255.255.224	= 11111111.11111111.11111111.11100000
AND = 192.168.0.64	= 11000000.10101000.00000000.01000000 = The network ID

As you can see from this example, this operation produced two different network IDs (192.168.0.128 and 192.168.0.64). This means that the second computer is not on the same subnet as your computer. To communicate, these computers need to send their messages through a router connecting the two subnets.

EXERCISE 11-1: WHICH IPS ARE ON THE SAME SUBNET?

Is IP address 192.168.0.200 on the same subnet as your computer? Assume your computer has an IP address of 192.168.0.133 and a subnet mask of 255.255.255.224.

Here's another way to look at this: the network prefix describes the range of addresses that can be used on a subnet. The first address in that range is defined as the network prefix bits followed by all binary 0s for the host identifier. Continuing with our example computer at 192.168.0.133, the first address on its subnet is 192.168.0.128. The last address in the range is the network prefix bits followed by all binary 1s for the host identifier. In our example that's 192.168.0.159. The first and last addresses have special meanings—the first identifies the network, the last is the *broadcast address* (used for sending a message to all hosts on the subnet). All the addresses in between can be used for hosts on the subnet. Our example IP address of 192.168.0.133 is clearly in this range (from 192.168.0.128 to 192.168.0.159), while the other computer with an IP address of 192.168.0.84 is outside this range.

You can also use the number of bits reserved for the host identifier to determine how many IP addresses are available for hosts on a subnet. In our example, 27 bits are reserved for the network prefix, leaving 5 bits for host identifiers. These 5 bits give us 32 possible host addresses, since 2^5 is 32. However, as mentioned earlier, the first and last addresses have special purposes, so really only 30 hosts can be identified using this

network prefix. This aligns with our earlier findings: the first host identifier is 128, and $128 + 32$ gives us 160, the first address in the next subnet, so 159 would be the last host in our range.

NOTE

Please see Project #30 on page 255, where you can look at the internet layer using your Raspberry Pi.

Transport Layer

The *transport layer* provides a communications channel that applications may use to send and receive data. There are two commonly used transport layer protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP provides a reliable connection between two hosts. It ensures that errors are minimized, data arrives in order, lost data is resent, and so forth. Data sent with TCP is known as a *segment*. On the other hand, UDP is a “best effort” protocol, meaning its delivery is unreliable. UDP is preferred when speed is valued over reliability. Data sent with UDP is known as a *datagram*. Both protocols have their place, but to keep things simple, I cover only TCP for the remainder of the chapter. Figure 11-11 illustrates the idea that a TCP segment fits within a packet’s data section, which in turn fits within a frame’s data section.

As we saw earlier, the link layer includes a destination MAC address in the frame header to identify a local network interface, and the internet layer includes a destination IP address in the packet header to identify the host on the internet. That’s enough information to get a packet to a specific device on the internet. Once a packet has reached its destination host, the transport layer header includes a destination network *port number* that identifies the specific service or process that will receive the data. A host with a single IP address can have multiple active ports, each used for performing a different type of activity on the network.

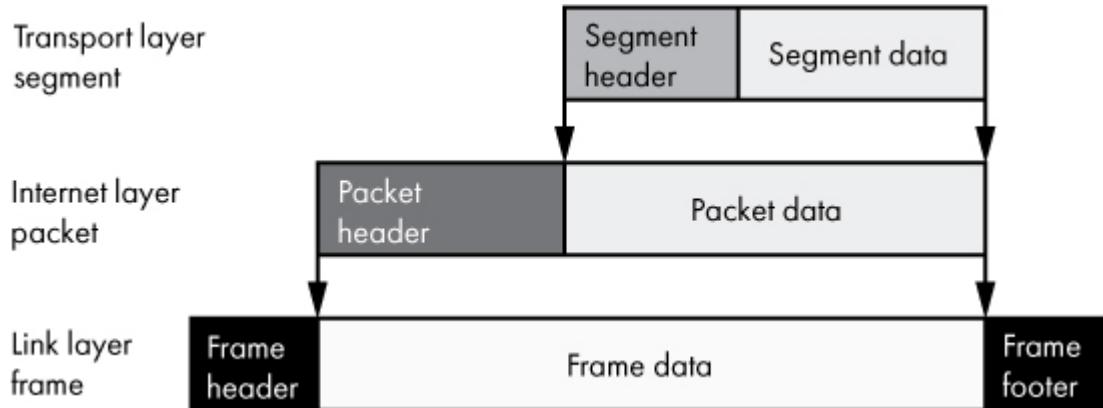


Figure 11-11: A TCP segment is contained in the data section of an IP packet

To use an analogy, an IP address is like the street address of an office building, and a network port number is like the office number of a worker in that office building. The IP address uniquely identifies a host computer, just as a street address uniquely identifies an office building. Using internet protocol, a packet can be delivered to a host in the same way that a package can be delivered to an office building. However, once a packet arrives at the computer, the operating system must decide what to do with it. The packet isn't intended for the OS itself, but for some process running on the computer. In the same way, a package arriving at an office building likely isn't intended for the mailroom worker but for someone else in the building. An operating system examines the port number and delivers the inbound data to the process listening on the specified port, just as a mailroom worker examines the name or office number on the package to deliver the package to the right person.

Network ports in the range of 0 to 1,023 are called *well-known ports*, whereas ports in the range of 1,024 to 49,151 can be registered with the Internet Assigned Numbers Authority (IANA) and are known as *registered ports*. Ports with a value greater than 49,151 are *dynamic ports*. Technically, any process with sufficient privileges can listen on any port that isn't already in use on a system, potentially ignoring the typical use case for that port number. However, when a client application wishes to connect to a remote service on another computer, it needs to know what port to use, so it makes sense to standardize port numbers. For example,

web servers typically listen on port 80 and port 443 (for encrypted connections). A web browser assumes that it should use port 80 or 443 unless directed otherwise.

EXERCISE 11-2: RESEARCH COMMON PORTS

Find the port numbers for common application layer protocols. What are the port numbers for Domain Name System (DNS), Secure Shell (SSH), and Simple Mail Transfer Protocol (SMTP)? You can find this information online with a search, or by looking at the IANA registry, here: <http://www.iana.org/assignments/port-numbers>. The IANA listings sometimes use an unexpected term for the service name. For example, DNS is simply listed as “domain.”

Servers use well-known ports to make it easy for clients to connect. However, most network communication is a two-way street (a client sends a request and a server responds), so the client needs to have an open port as well so that it can receive data from the server. A client only needs to temporarily open such a port, just long enough for it to complete its communication with a server. Such ports are called *ephemeral ports* and are assigned by the networking components in the operating system. For example, a client web browser connects to a web server on port 80, and an ephemeral port on the client is also opened, let’s say port number 61,348. The client sends its web request to port 80 on the server, and the server sends its response to port 61,348 on the client.

An IP address plus a port number form an *endpoint*, and an instance of an endpoint is known as a *socket*. A socket can listen for new connections, or it can represent an established connection. If multiple clients connect to the same endpoint, each has its own socket.

NOTE

Please see Project #31 on page 256, where you can look at the port usage of your Raspberry Pi.

Application Layer

The *application layer* is the final, topmost layer of the internet protocol suite. While the lower three layers provide a generalized foundation for communication over the internet, the protocols at the application layer focus on accomplishing a specific task. For example, web servers use *HyperText Transfer Protocol (HTTP)* for retrieving and updating web content. Email servers use *Simple Mail Transfer Protocol (SMTP)* for sending and receiving email messages. File transfer servers use *File Transfer Protocol (FTP)* to, you guessed it, transfer files! In other words, the application layer is where we get to the protocols that describe the behavior of applications, whereas the lower layers of the stack are the “plumbing” that enables applications to do the things they want to do over the internet. Figure 11-12 provides a completed view of the four layers.

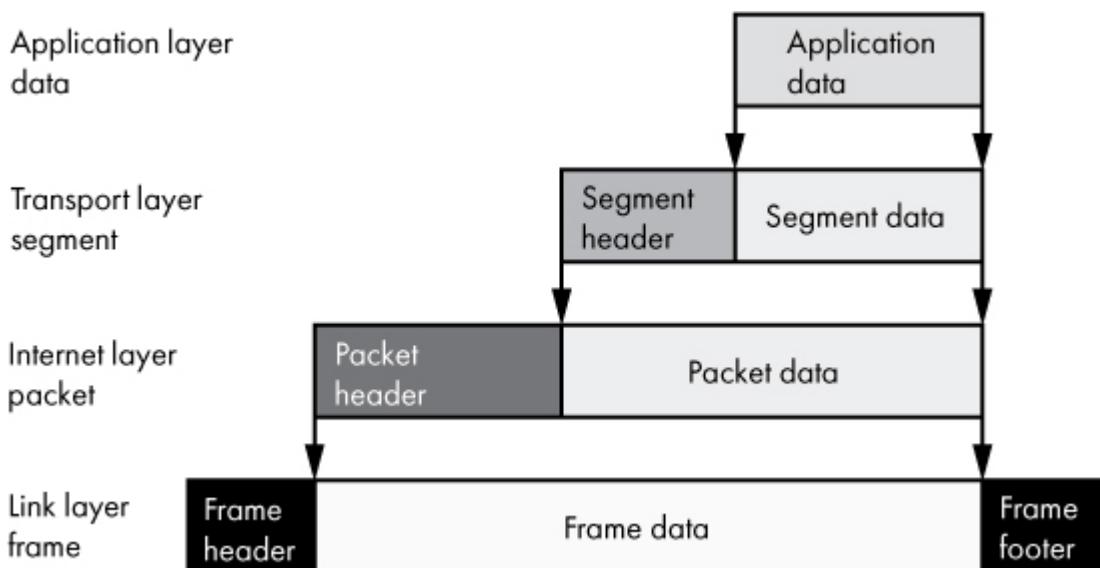


Figure 11-12: The application layer’s data is contained in the segment’s data section.

Figure 11-12 is a breakout view of how each layer fits in the lower layer’s data payload. Assembling all the layers together in Figure 11-13, we can see a representation of what is contained in a frame sent to a device on the internet.

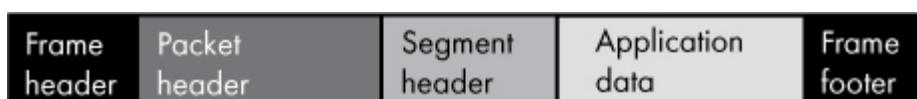


Figure 11-13: A frame containing an IP packet, a TCP segment, and application data

We've walked through the contents of a network frame from the bottom up, starting with the layer closest to hardware. When a frame is received by a host, it is processed by the host in that same order, from the link layer up to the application layer. In contrast, when a frame is sent from a host, the frame is assembled in the reverse order. A process prepares application data that is enclosed in a segment, a packet, and finally, a frame.

A Trip Through the Internet

Now that you're familiar with each of the four layers in the TCP/IP networking model, let's look at an example of how data travels across the internet. We'll see how various devices along the way interact with each of the networking layers. Figure 11-14 illustrates this, showing a client in the upper left communicating with a server in the lower left.

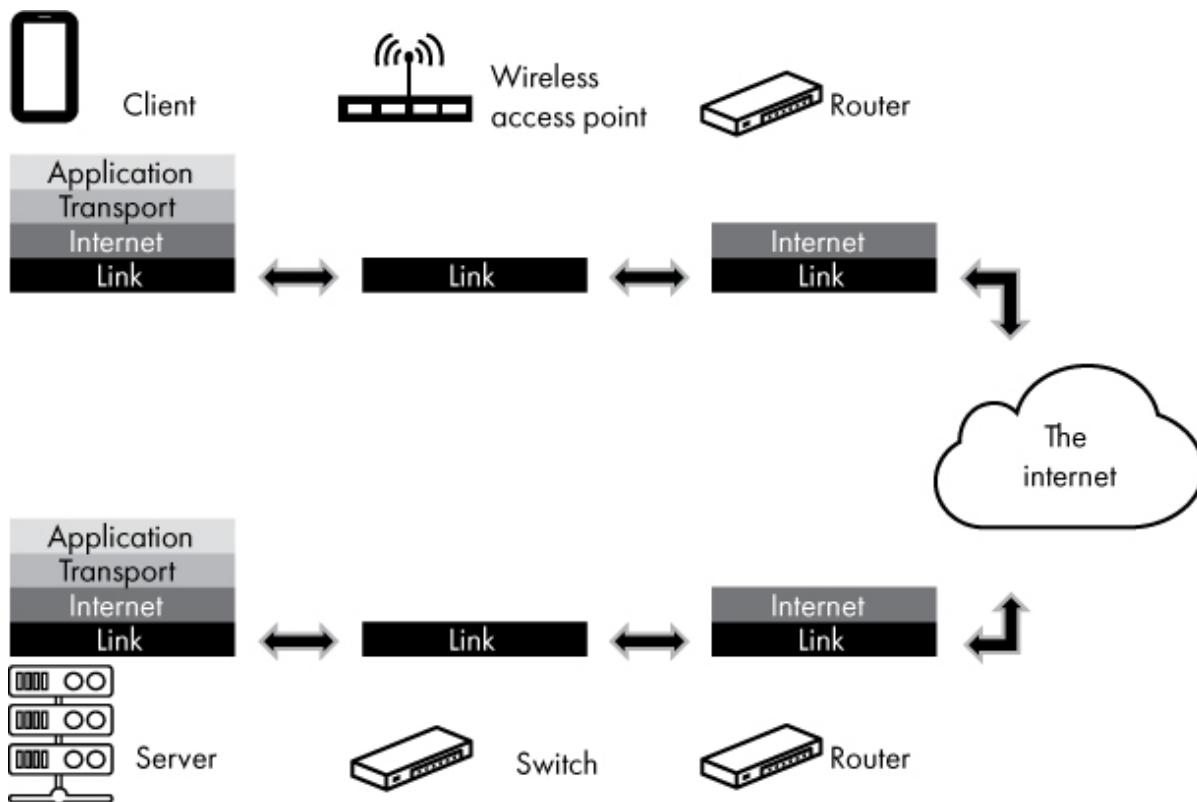


Figure 11-14: Different devices interact at different layers of the networking stack

I'll set up the scenario in Figure 11-14. A client device (upper left of diagram) is connected to a wireless Wi-Fi network. That network is connected to the internet via a router. Somewhere else we have a server (lower left of diagram), which has a wired connection to the internet through a switch and router. A user of the client device opens a web browser and requests a web page hosted on the server. For simplicity, let's assume that the client already knows the IP address of the server.

The web browser on the client "speaks" HTTP, the application layer protocol of the web, so it forms an HTTP request intended for the destination server. The browser then hands off the HTTP request to the operating system's TCP/IP software stack, asking that the data be delivered to the server—specifically the server's IP address and port 80, the standard port for HTTP. The TCP/IP software stack on the client operating system then encapsulates the HTTP payload in a TCP segment (transport layer), setting the destination port to 80 in the segment header. If necessary, TCP divides the application layer data into multiple segments, each with its own header. The internet layer software on the client then wraps the TCP segment in an IP packet, which includes the destination IP address of the server in the packet header. If necessary, IP divides the packet into multiple smaller fragments in preparation for transmission over the network link. At the link layer on the client, the IP packet is encapsulated in a frame with the MAC address of the local router in its header. This frame is wirelessly transmitted by the client device's Wi-Fi hardware.

The wireless access point receives the frame. The access point, operating at the link layer, sends the frame along to the router. The router examines the internet layer packet to determine the destination IP address. To reach the server, the request needs to travel through multiple routers on the internet. The local router encapsulates the packet in a new frame, with a new destination MAC address (the address of the next router), and sends the new frame on its way. This routing

process continues through multiple routers on the internet until the request reaches the router on the subnet where the server is connected.

The last router encapsulates the packet in a frame suitable for the server's local network. This frame's header includes the MAC address of the server. The switch on the server's subnet looks at the MAC address in the frame and forwards the frame out the appropriate physical port. There's no need for the switch to look at any higher layers. The server receives the frame, and the driver for the network interface passes the TCP/IP packet up to the TCP/IP software stack, which in turn, hands off the HTTP data to the process listening on TCP port 80. Web server software, listening on port 80, handles the request. This includes replying to the client, and to do that, this entire process happens again, except in reverse order.

NOTE

Please see Project #32 on page 258, where you can see the route from your Raspberry Pi to a host on the internet.

Foundational Internet Capabilities

Whereas TCP/IP provides the necessary plumbing for reliable transfer of data across the internet, other protocols provide additional foundational internet capabilities. These features are implemented as application layer protocols. Let's now look at two such protocols (DHCP and DNS) and a system for translating IP addresses (NAT).

Dynamic Host Configuration Protocol

Every host on the internet needs an IP address, a subnet mask, and the IP address of its router (also called its *default gateway*) in order to communicate with other hosts. How are IP addresses assigned? A device can be given a *static IP address*, which requires someone to edit the configuration on the device and set its IP information manually. This is sometimes useful, but it requires the user to ensure that the IP address in question isn't already taken and is valid for the subnet. Most end

users don't have the expertise to manually configure the IP settings for their devices, nor do they want to deal with the hassle of manual configuration. Fortunately, most IP addresses are assigned dynamically using *Dynamic Host Configuration Protocol (DHCP)*. With DHCP, when a device connects to a network, it receives an IP address and related information without user intervention.

For DHCP to be available on a network, a device on the network must be configured as a *DHCP server*. This server has a pool of IP addresses that it's allowed to assign to devices on the network. The flow of DHCP is illustrated in Figure 11-15.

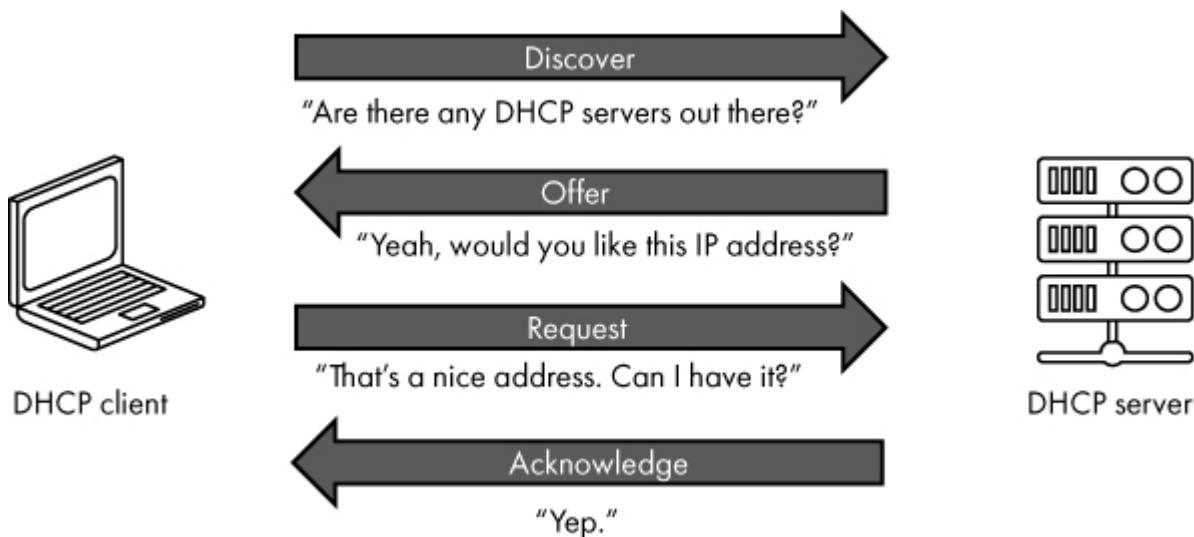


Figure 11-15: A DHCP conversation

Let's walk through Figure 11-15. When a device connects to a network, it broadcasts a message to discover any DHCP servers. A *broadcast* is a special kind of packet that's addressed to all hosts on the local network. When the DHCP server receives this broadcast, it offers an IP address to the client device. If the client wishes to accept the offered IP address, it replies to the server with a request for the offered address. The DHCP server then acknowledges the request, and the IP address is assigned to the client. The IP address is *leased* to the client, and it eventually expires if the client does not renew its lease.

NOTE

Please see Project #33 on page 258, where you can see the IP address your Raspberry Pi has leased using DHCP.

Private IP Addresses and Network Address Translation

The number of available IP addresses is limited, so most home internet service providers (ISPs) only assign a single IP address to a customer. This IP address is assigned to the device that's directly attached to the ISP's network, usually a router. However, many customers have multiple devices on their home network. Let's look at how multiple devices can share a single public IP address by leveraging private IP addresses and Network Address Translation.

Certain ranges of IP addresses are considered *private IP addresses*, addresses intended to be used on *private networks*, such as those in homes or offices, where the devices aren't directly connected to the internet. Any address that matches the pattern of 10.x.x.x, 172.16.x.x, or 192.168.x.x is a private IP address. Anyone can use these ranges of IP addresses without asking permission. The catch is that private IP addresses are nonroutable—they can't be used on the public internet. A DHCP server on a home network can assign these addresses without worrying about whether any other network is using the same addresses. Unlike public IP addresses that must be unique, private IP addresses are intended to be used simultaneously on multiple private networks. It doesn't matter if multiple networks use the same addresses, since the addresses won't ever be seen outside of the private network anyway. Private IP addresses solve the problem of an ISP only providing a single public IP address to a home or business, but how are private IP addresses useful if they aren't routable on the internet?

Network Address Translation (NAT) allows devices on a private network, often a home network, to all use the same public IP address on the internet. As packets flow through a NAT router, the router modifies the IP address information in those packets. When a packet originating from the private home network arrives at the NAT router, it modifies the source IP address field to match the public IP address, as shown in Figure 11-16.

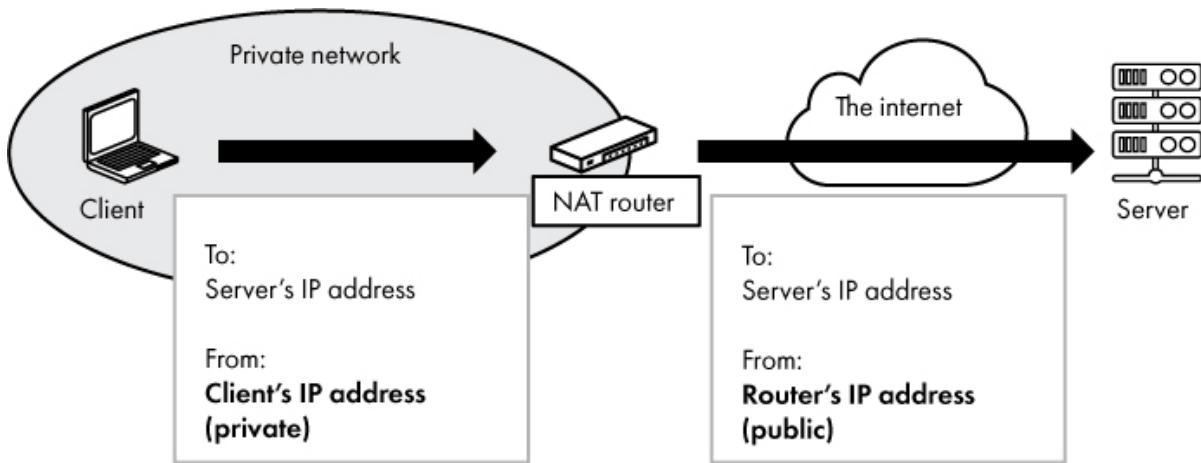


Figure 11-16: A NAT router replaces private IP addresses with its own public IP address

When a response comes back to the router, it sets the destination IP address to the private address of the host that originated the request. In this way, all traffic from the home appears to originate from the same public IP address, even if there are actually multiple devices on the private network. NAT also has the side benefit of security: the devices on the private network aren't directly exposed to the public internet, so a malicious user on the internet can't initiate a connection directly to a private device. Most routers sold to consumers for home use are NAT routers, often with built-in wireless access point capabilities as well.

Private IP addresses are valuable not only for home networks, but also for businesses that don't want their computers exposed to the public internet. Many corporate networks use a *proxy server* rather than a NAT router. A proxy server is similar to a NAT router in that it allows devices on a private network to access the internet, but a proxy server differs in that it typically operates at the application layer rather than the internet layer. Proxies also usually provide additional features such as user authentication, traffic logging, and content filtering.

NOTE

Please see Project #34 on page 259, where you can see if the IP address your device is assigned is a public IP address or a private IP address.

The Domain Name System

We've seen that hosts on the internet are identified by IP addresses. However, most users of the internet rarely, if ever, directly deal with IP addresses. Although IP addresses work well for computers, they aren't very user friendly. No one wants to remember sets of four numbers separated by periods. Fortunately, we have the *Domain Name System* (*DNS*) to make things easier for us. DNS is an internet service that maps names to IP addresses. This allows us to refer to a host by a name like *www.example.com* rather than by its IP address.

A computer's full DNS name is known as a *fully qualified domain name*, or *FQDN*. A name like *travel.example.com* is an FQDN. This name is composed of a short, local *hostname* (*travel*) and a domain suffix (*example.com*). The term *hostname* is often used interchangeably to mean either the computer's short name or the FQDN. Later in this section, we use *hostname* to mean a computer's FQDN. A *domain*, like *example.com*, represents a grouping of network resources managed by an organization. Both *example.com* and *travel.example.com* are domain names. The former represents a network domain; the latter represents a specific host on that domain.

Software needs to be able to query DNS to convert hostnames to IP addresses—this is known as *resolving* a hostname. To enable this functionality, hosts are configured with a list of the IP addresses of DNS servers. This list is usually provided by DHCP, and it typically is composed of DNS servers maintained by the internet service provider or running on the local network. When a client wants to connect to a server by name, it asks a DNS server for the IP address corresponding to that name. The server replies with the requested IP address, if it can. This is illustrated in Figure 11-17.

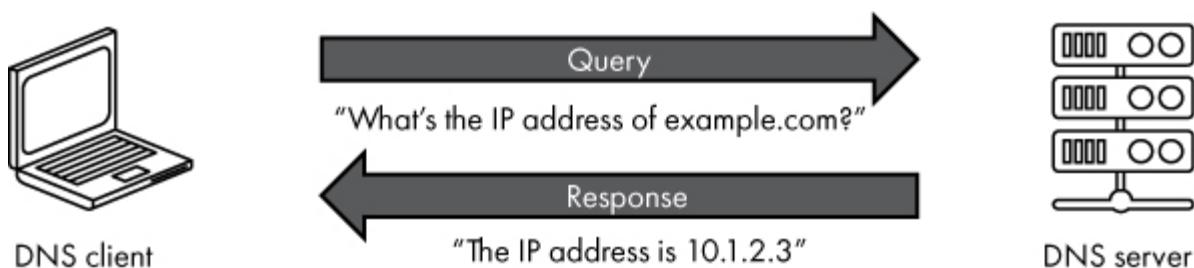


Figure 11-17: A simplified DNS query. The IP address of example.com is not intended to be accurate.

Once the client has the server's IP, it proceeds to communicate with the server using the IP address, as described earlier. I've heard DNS described as the phone book of the internet, although that analogy may fall short for some readers since phone books aren't as common as they once were!

You might assume that there's a one-to-one mapping between IP addresses and names. That actually isn't the case. A name can map to multiple IP addresses. In that scenario, different clients query DNS for a certain name, and they may all receive a different IP address as a response. This is useful for situations in which the load for a given service needs to be distributed across multiple servers. This can be done geographically, so that clients in Europe, for example, get a different IP address than clients in Asia, allowing clients in each region to connect to the IP address of a server that's physically close to them.

The reverse is possible too: multiple names can map to the same IP address. In this scenario, a query for different names may return a single IP address. This is useful when a server hosts multiple instances of the same type of service, each identified by name. This is common in web hosting, where a single server hosts multiple websites, each identified by its DNS name.

Each entry in DNS is known as a *record*. There are various kinds of records; the most basic is an *A record*, which simply maps a hostname to an IP address. Other examples are *CNAME (canonical name)* records that map one hostname to another hostname, and *MX (mail exchanger)* records used for email services.

No single organization would want to undertake the task of managing the many, many DNS records that exist today. Fortunately, this isn't needed; DNS is implemented in a way that allows for shared responsibility. A DNS name like *www.example.com* actually represents a hierarchy of records, and different DNS servers are responsible for

maintaining the records at different levels of the hierarchy. The DNS hierarchy, as applied to *www.example.com*, is illustrated in Figure 11-18.

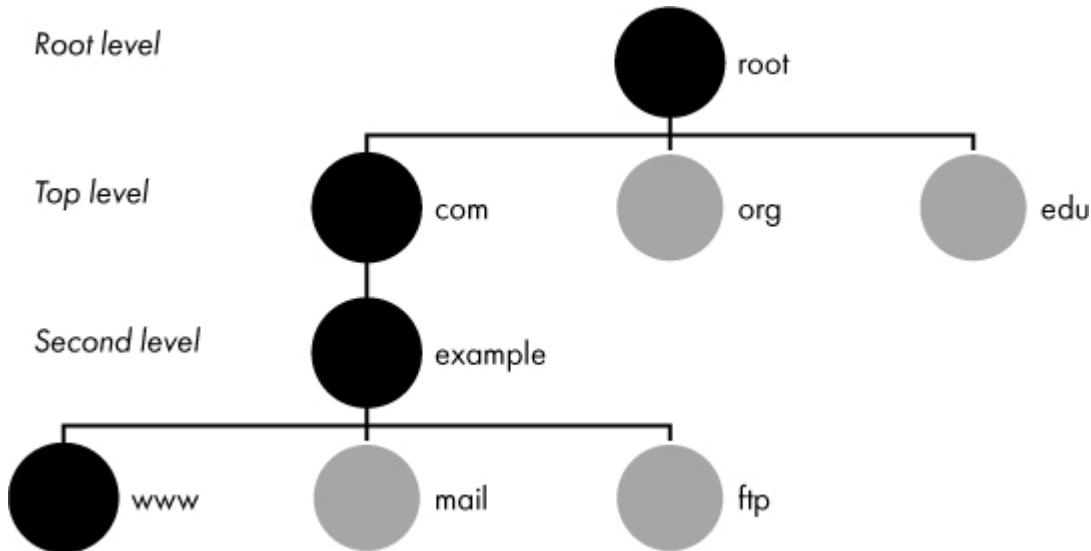


Figure 11-18: Example DNS hierarchy, highlighting *www.example.com*

At the top of this hierarchical tree is the *root domain*. The root domain doesn't get a textual representation in a DNS name like *www.example.com*, but it's an essential part of the DNS hierarchy. The root domain contains records for all the *top-level domains (TLDs)* like *.com*, *.org*, *.edu*, *.net*, and so forth. As of 2020, there are 13 root name servers worldwide, each responsible for knowing the details of all the top-level domain servers. Let's say you want to look up a record in a domain that ends with *.com*. A root server can point you to a TLD server that knows about domains under *.com*.

A top-level DNS server is responsible for knowing about all the second-level domains under its hierarchy. A top-level DNS server for *.com* could point you to the second-level DNS server for *example.com*. The DNS servers for second-level domains maintain records for hosts and third-level domains that fall under second-level domains. This means that the DNS server(s) for *example.com* are responsible for maintaining the records for hosts like *www.example.com* and *mail.example.com*. This pattern continues, allowing for nested domains.

Once a domain is registered under a top-level domain, the owner of that domain can create as many records as needed under their domain.

As mentioned earlier, when a computer needs to find the IP address for an FQDN, it sends a request to its configured DNS server. What does the DNS server do with this request? If the server has recently looked up the requested record, it may have a copy of that record stored in its cache, and it can immediately return the IP address to the client. If the DNS server doesn't have the response in cache, it may query other DNS servers as needed to get the answer. This involves starting at the root and working down the hierarchy of servers to find the record in question. Once the server has the record, it can cache it so that it can immediately respond to future queries for that record. Eventually the cached record is removed to ensure that the server always provides reasonably recent data.

NOTE

Please see Project #35 on page 260, where you can look up information in DNS.

Networking Is Computing

Let's take a moment to consider how the internet fits in with the broader picture of computing that we've already covered in this book. Networking may seem like a tangential topic, but really it isn't so far removed from computing in general. The internet is composed of hardware and software working together to allow communication between devices. Data sent over the internet boils down to 0s and 1s, represented in various forms such as voltages on a wire. From the perspective of a computer, a networking interface like a Wi-Fi or Ethernet adapter is just another I/O device. An operating system interacts with such adapters via device drivers, and the OS includes software libraries that allow applications to easily communicate over the internet. Networking devices like routers and switches are computers too, although highly specialized ones. The internet, and networking in

general, is an extension of local computing, allowing for data transfer and processing beyond the boundaries of a single device.

Summary

In this chapter we covered the internet, a globally connected set of computer networks that all use a suite of common protocols. You learned about the four layers of the internet protocol suite—the link layer, the internet layer, the transport layer, and the application layer. You saw how data travels through the internet and how devices interact at various layers. You learned how DHCP provides networking configuration data, how NAT allows devices on private networks to connect to the internet, and how DNS provides friendly names that can be used in place of IP addresses. In the next chapter you'll learn about the World Wide Web, a set of resources delivered by HTTP over the internet.

PROJECT #29: EXAMINE THE LINK LAYER

Prerequisite: A Raspberry Pi, running Raspberry Pi OS. I recommend that you flip to Appendix B and read the entire “Raspberry Pi” section if you haven’t already.

In this project, you’ll use your Raspberry Pi to check out the link layer of your local network. Let’s start with the following command, which lists the MAC address of your Ethernet adapter:

```
$ ifconfig eth0 | grep ether
```

The output should look something like the following:

```
ether b8:27:eb:12:34:56 txqueuelen 1000 (Ethernet)
```

In this example, the MAC address is `b8:27:eb:12:34:56`. That’s a hexadecimal representation of a 48-bit number. Remember, each hex character represents 4 bits, so that’s 12 characters × 4 bits = 48 bits.

The first 24 bits of a MAC address represent the vendor/manufacturer of the hardware. This number is known as an *organizationally unique identifier (OUI)* and is

managed by the *Institute of Electrical and Electronics Engineers (IEEE)*. In this case the OUI is B827EB, which is assigned to the Raspberry Pi Foundation. You can see the current OUI listings here: <http://standards-oui.ieee.org/oui.txt>.

Your Raspberry Pi's Wi-Fi adapter has its own MAC address. View it like this:

```
$ ifconfig wlan0 | grep ether
    ether b8:27:eb:78:9a:bc  txqueuelen 1000  (Ethernet)
```

On my system, the OUI (the first 24 bits of the MAC address) of the Wi-Fi adapter is the same as the OUI of the Ethernet adapter. This is because both adapters are internal Raspberry Pi hardware and use the OUI for the Raspberry Pi Foundation.

From your Raspberry Pi, you can also see the MAC address of other devices on your local network. To do this you can use a tool called `arp-scan` that attempts to connect to every computer on your local network and retrieve its MAC address.

First, install the tool:

```
$ sudo apt-get install arp-scan
```

Then run this command (that's a lowercase L at the end of the command, not the number 1):

```
$ sudo arp-scan -l
```

You should get a list of IP addresses (which we cover elsewhere in this chapter) and MAC addresses, plus a column that attempts to match the MAC prefix to the manufacturer. I got 10 results on my local network, some of which I didn't immediately recognize. You may see some duplicate results returned, indicated with DUP in the third column. The returned list typically does not include the address of the computer from which you ran the scan.

You may have some results in the third column that show as (Unknown). This means that the `arp-scan` tool wasn't able to match the OUI number to a known manufacturer, probably because the tool is using an outdated version of the OUI list. You can try to fix this by downloading the current list of OUI numbers from IEEE and then running the scan again, like this:

```
$ get-oui
$ sudo arp-scan -l
```

When I see multiple devices on my home network that I can't identify right away, I have an immediate urge to figure out what they are! As a bonus challenge for you, identify every device returned from `arp-scan`. Now, this may be impractical if you're running this tool on a network you don't control (say, at a coffee shop or library), but if you're at home, this is something you can do. You'll probably need to log on to each device on your network and dig through its settings to find its IP address or MAC

address and see if it matches one of the entries returned from `arp-scan`. Hint: use the `ifconfig` utility on Linux or Mac, or the `ipconfig` tool on Windows. On mobile devices, look at the user interface for network settings.

PROJECT #30: EXAMINE THE INTERNET LAYER

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll look at the internet layer using your Raspberry Pi. Let's begin with the following command, which lists all the network interfaces on your device and their associated IP addresses.

\$ `ifconfig`

You'll typically see three interfaces: `eth0`, `lo`, and `wlan0`. The `lo` interface is a special case; it's the *loopback* interface. It's used for processes running on the Pi that wish to communicate with each other using TCP/IP, but without actually sending any traffic over the network. That is, the traffic stays on the device. The loopback interface has an IP address of 127.0.0.1. This is a special address that is not routable and can't be used as an address on the local subnet, because any attempt to deliver messages to that address results in the messages coming right back to the sending computer. In other words, every computer considers 127.0.0.1 to be its own IP address.

As we covered in the previous project, `eth0` is the wired Ethernet interface and `wlan0` is the wireless Wi-Fi interface. If you're connected to a network on either or both of these interfaces, you should see an IP address beside the text `inet` in the `ifconfig` output. You may also see an IPv6 address listed beside `inet6`. Here's example `wlan0` output from `ifconfig`:

```
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 192.168.1.138 netmask 255.255.255.0 broadcast 192.168.1.255
      inet6 fe80::8923:91b2:13e0:ed2a prefixlen 64 scopeid 0x20<link>
```

In this output you can see that the assigned IP address is 192.168.1.138. The `netmask` value (subnet mask) is 255.255.255.0, and the `broadcast` address is 192.168.1.255.

The `ifconfig` command gives us information about the various network interfaces on the Raspberry Pi, but it doesn't tell us about how routing is configured. Let's take a look at that using the `ip route` command. I've included sample output here; your results may vary.

```
$ ip route
default via 192.168.1.1 dev wlan0 src 192.168.1.138 metric 303
```

```
192.168.1.0/24 dev wlan0 proto kernel scope link src 192.168.1.138 metric  
303
```

This command's output can be a little difficult to interpret. In short, the first line gives the default route. This is where packets should be sent when there isn't a specific route that applies. In this particular example, every packet that doesn't match a specific routing rule should be sent to 192.168.1.1. That means that 192.168.1.1 is the IP address of the local router, also known as the *default gateway*.

The next line is a routing entry that tells you that any packet sent to an IP address in the range of 192.168.1.0/24 should be sent through device `wlan0`. That's the Wi-Fi adapter on the local subnet. In other words, this routing rule ensures that communication with IP addresses on the local subnet happens directly, without going through a router.

To summarize, this output tells you that any packet sent to an IP address that matches 192.168.1.0/24 should be sent directly to the destination address via the `wlan0` interface. Any other traffic uses the default route, which sends traffic to the router at 192.168.1.1. The end result is that local subnet traffic is sent directly to the target device, while traffic to devices on other subnets, likely on the internet, is sent to the default gateway.

PROJECT #31: EXAMINE PORT USAGE

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll see which network ports are in use on a Raspberry Pi. You'll then examine ports on other computers. Let's begin with the following command that shows you the listening and established TCP sockets on your Raspberry Pi.

```
$ netstat -nat
```

Let's break down the `-nat` options used in the command. The `n` option indicates that numeric output should be used to show the port numbers. The `a` option means show all connections (both listening and established), and `t` means limit the output to TCP. On my device, I see a list like so:

Active Internet connections (servers and established)					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN
tcp	0	36			
	192.168.1.138:22		192.168.1.125:52654	ESTABLISHED	
tcp	0	0			
	192.168.1.138:22		192.168.1.125:51778	ESTABLISHED	
tcp6	0	0	:::22	:::*	LISTEN

Here you see four sockets, all related to SSH. I can tell they are related to SSH because all the sockets are using port 22. I have SSH enabled on my Raspberry Pi to allow remote terminal connections. The first and last lines show that the Pi is listening on port 22 for new incoming SSH connections using both TCP and TCP over IPv6. The middle two lines show that I have two established SSH connections to this device, both of them from my laptop (with an IP of 192.168.1.125) to the Pi (with an IP of 192.168.1.138). Note how both the established connections go to the same server port on the Pi (22), whereas the client port on my laptop varies (52654 and 51778), as they are ephemeral ports.

Run the command again, this time adding the **p** option and prefixing the command with **sudo**:

```
$ sudo netstat -natp
```

This gives you the same list, but with the process ID (PID) and program name to which the socket belongs. Any traffic sent to the socket is directed to the PID, which handles the traffic and responds as needed. On my computer I see that the program using this port is **sshd**—the daemon for SSH.

Now that you've examined which ports are in use on your Raspberry Pi, let's examine the ports on a remote computer. For this, you'll use a tool called **nmap**, which must first be installed on your Raspberry Pi:

```
$ sudo apt-get install nmap
```

Once the tool is installed, select a target host that you wish to scan. This can be either a device on your network (say your router or a laptop) or a host on the internet. Note that repeatedly scanning a host that you don't control may look suspicious to the administrators of that server, so I strongly recommend that you only scan devices that you own.

In my case, I decided to scan my default gateway, which happens to be at 192.168.1.1. The following **nmap** command scans for open TCP ports on the specified IP address. Try this on your Raspberry Pi, replacing the IP address with the address of the device you wish to scan. If you want to scan your own router, see Project #30 for a reminder of how to get the IP address of your default gateway.

```
$ nmap -sT 192.168.1.1
```

A partial listing of the results from my scan showed these ports:

PORt	STATE	SERVICE
53/tcp	open	domain
80/tcp	open	http

This tells me that the device acts not only as a router, but as a DNS server (port 53) and web server (port 80). It's normal for a home router to provide these services.

PROJECT #32: TRACE THE ROUTE TO A HOST ON THE INTERNET

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll examine the route a packet travels from your Raspberry Pi to a host on the internet. First, you need to choose a host on the internet. This can be a website like *www.example.com*, or the IP address or FQDN of any internet host you happen to know. Once you've decided on a host, enter the following command, replacing *www.example.com* with the name or IP address of the host you wish to see.

```
$ traceroute www.example.com
```

The **traceroute** tool attempts to show the routers that are encountered on a packet's journey across the internet. The output should be read line by line. Each line is sequentially numbered and shows the name (if available) and IP address of the router encountered at that step of the packet's trip. If there is no response after a short time, the output displays an asterisk (*) and moves on to the next router. You may also see more than one IP address per line, indicating multiple possible routes.

PROJECT #33: SEE YOUR LEASED IP ADDRESS

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll look at the lease information associated with your Raspberry Pi's IP address obtained from a DHCP server. Of course, this assumes that your Raspberry Pi is configured to use DHCP (which is the default) rather than a static IP address. To do this, look at the system log:

```
$ cat /var/log/syslog | grep leased
```

Expect to see output similar to the following:

```
Jan 24 19:17:09 pi dhcpcd[341]: eth0: leased 192.168.1.104 for 604800
seconds
```

Here you can see that IP address 192.168.1.104 was leased from a DHCP server for use on network interface `eth0`, the Ethernet interface on the Raspberry Pi. Your output likely shows a different IP address and perhaps a different interface, maybe `wlan0`.

By default, the `syslog` file is periodically cleared, and its contents are moved to a backup file. Because of this, you may not see a DHCP entry in your `syslog` file. You can release your current IP address, request a new one, and look again for the lease entry like so:

```
$ sudo dhclient -r wlan0
$ sudo dhclient wlan0
$ cat /var/log/syslog | grep leased
```

Replace `wlan0` with `eth0` if you want to do this for Ethernet rather than Wi-Fi.

PROJECT #34: IS YOUR DEVICE'S IP PUBLIC OR PRIVATE?

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll see if the IP address of your Raspberry Pi is public or private. If your device has a private IP address, you'll also find the public IP address that is used for your communication over the internet. As before, you can use the following utility to see your device's assigned IP address(es).

```
$ ifconfig
```

When looking for your device's assigned IP address, you'll likely see an entry for 127.0.0.1; you can ignore this one since it's used for loopback (see Project #30). As mentioned earlier, any address that matches the pattern of 10.x.x.x, 172.16.x.x, or 192.168.x.x is a private IP address. Now, even if you have a private IP address like one of these, when you access resources on the internet, you're also indirectly making use of a public IP address. This is the address that websites or other internet services see when you connect to them. If you're on a home network, that public IP address is likely assigned to your router. If you're on a business network, that public IP address may be assigned to a proxy device on the edge of your corporate network. In either case, all the network traffic from inside your local network to the internet originates from that public address.

To find the public IP address that your device uses when connecting to a device on the internet, one option is to log on to your router or proxy server and check its network configuration. If you know how to query your router or proxy server for this information, feel free to do so. However, since every model of network device is somewhat different, I won't walk you through the steps here.

A more universal option is to query an online service that can return your current public IP address. This is possible because every internet server that your device connects to knows your IP address; it's simply a matter of finding a service that's willing to tell you what IP address it sees. If you're running a web browser on your device, perhaps the simplest thing to do is query Google for something like "my IP address." This usually returns the information you want.

If you're working from a terminal, like on the Raspberry Pi, you can use the `curl` utility to make an HTTP request to a website that returns your current IP address. The following are a few examples of services that are available for this at the time of this writing:

```
$ curl http://ipinfo.io/ip
$ curl http://checkip.amazonaws.com/
$ curl http://ipv4.icmhanzip.com/
$ curl http://ifconfig.me/ip
```

Any of these should return your public IP address to the terminal window. Compare this address with the address you got earlier from `ifconfig`. If they are the same, then your device is directly assigned a public IP address. If they differ, then your device likely has a private IP assigned to it, and you're connecting to the internet through a NAT router or proxy server.

PROJECT #35: FIND INFORMATION IN DNS

Prerequisite: A Raspberry Pi, running Raspberry Pi OS.

In this project, you'll use your Raspberry Pi to query DNS records. Let's begin by looking up the IP address of a website. You'll use the `host` utility to do this. The following command returns the IP address of `www.example.com`, the hostname of the site I'm interested in. Feel free to substitute the name of another host you wish to look up.

```
$ host www.example.com
```

You should see output that gives the IP address of the host. You may also see an IPv6 address. Depending on the hostname you queried, you may get back multiple records, since a DNS name can map to multiple IP addresses. You may also learn that the name you typed is actually an alias for a different name, which in turn maps to an IP address.

DNS also allows for reverse lookups, where you specify an IP address and a hostname is returned. This doesn't always work, since DNS records need to be in place to support it. To give this a try, just use `host` with an IP address. In the following command, replace `a.b.c.d` with your public IP address that you found in Project #34, or any other public IP address you wish to query. Again, this works only for IP addresses that have DNS records in place to support reverse lookups.

```
$ host a.b.c.d
```

By default, the `host` utility uses the DNS server your device is configured to use. You may also query a specific DNS server using `host` by specifying the IP address of that server. Internet service providers include DNS services for their customers, but many free alternate DNS services are available as well. For example, at the time of this writing, Google provides a DNS server at 8.8.8.8 and Cloudflare provides a DNS server at 1.1.1.1. If you want to use the DNS server at 1.1.1.1 to look up `www.example.com`, you could enter this:

```
$ host www.example.com 1.1.1.1
```

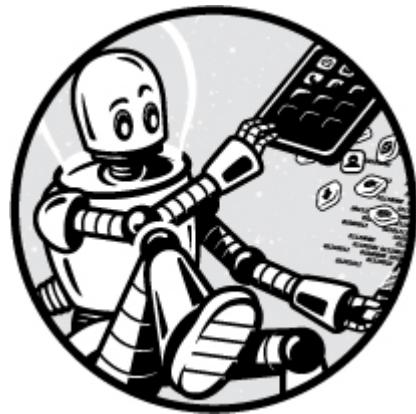
This should output IP address information as before, along with some text indicating which DNS server was used for the lookup.

If you're curious about the details of the DNS query, you can use the `-v` option with the `host` command, which provides verbose output.

```
$ host -v www.example.com
```

12

THE WORLD WIDE WEB



The previous chapter described the internet, the globally connected set of computer networks that share a suite of protocols. The World Wide Web is a system built on top of the internet, one so popular that it is often confused with the internet itself. In this chapter, we dive into the details of the web. We first look at its key attributes and related programming languages, and then we look at web browsers and web servers.

Overview of the World Wide Web

The *World Wide Web*, often just called the *web*, is a set of resources, delivered using *HyperText Transfer Protocol (HTTP)* over the internet. A *web resource* is anything that can be accessed using the web, such as a document or an image. A computer or software program that hosts web resources is called a *web server*, and a *web browser* is a type of application commonly used to access content on the web. Browsers are used to view documents known as *web pages*, and a collection of related web pages is known as a *website*. The web is distributed, addressable, and linked. Let's begin by examining each of those core attributes.

The Distributed Web

The World Wide Web is *distributed*. No centralized organization or system governs what content can be published to the web. Any computer connected to the internet can run a web server, and the owner of such a computer can make available any content they wish. That said, organizations or countries may choose to block users from accessing certain content on the web, and governments can shut down websites that host illegal content. Aside from those cases, the web is an open platform for publishing whatever people wish to publish, and no single organization controls what content is made available.

The Addressable Web

The web uses *Uniform Resource Locators (URLs)* to give every resource on the web a unique address that includes both its location and how to access it. URLs are commonly referred to as *web addresses* or just *addresses*. To illustrate how these addresses are structured, let's use a URL for a fictitious travel website, as shown in Figure 12-1. This URL identifies a page with information about traveling to the Carolinas.

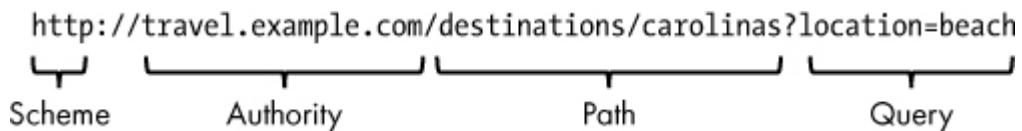


Figure 12-1: An example URL

A URL is composed of multiple parts. The URL *scheme* identifies the application layer protocol for accessing the resource. In this case, the protocol is HTTP, which we'll cover in more detail later. The colon (:) character indicates the end of the scheme portion. Following two forward slashes (//) is the *authority* portion of the URL. In this example, the authority portion contains a DNS hostname of the server(s) where the resource resides, *travel.example.com*. An IP address can be used here as well. Other information besides the host can also go in this section, such as a username (preceding the host and followed by an @ sign) or port number (following the host and prefixed with a colon). The *path*

portion of the URL is next; it specifies the location of a resource on the web server. A URL path is analogous to a filesystem path, organizing resources into a logical hierarchy. In our example, the path */destinations/carolinias* implies that the site has a collection of pages that describe travel destinations, and the particular page specified in the URL is a page about the Carolinas. We could reasonably assume that if the site had a page describing Florida as a destination, it would be found at */destinations/florida*. Finally, the *query* portion of the URL acts as a modifier to the resource returned to the client. In our example, the query indicates that the *carolinias* page should display locations at the beach. The format and meaning of the query portion of the URL varies from site to site.

A lot of information is packed into that URL, so let me restate how to read it in plain language. A website is running on a computer named *travel.example.com*. The site speaks HTTP, so use that protocol when connecting to the site. On that site there's a page called *carolinias*, part of a collection of *destinations*. The query string directs the page to only show locations that are at the beach.

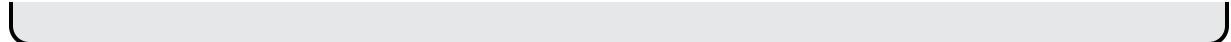
A URL does not have to include every element in the example in Figure 12-1. It may also contain some elements not included in this example. A URL that includes only the scheme and the authority is perfectly valid, such as *http://travel.example.com*. In that scenario, the website serves its default page, since no path is provided.

EXERCISE 12-1: IDENTIFY THE PARTS OF A URL

For the following URLs, identify the scheme, username, host, port, path, and query. Not all URLs include all these parts.

- *https://example.com/photos?subject=cat&color=black*
- *http://192.168.1.20:8080/docs/set5/two-trees.pdf*
- *mailto:someone@example.com*

You can check your answers in Appendix A.



A web browser typically displays the current URL in its address bar, as illustrated in Figure 12-2.

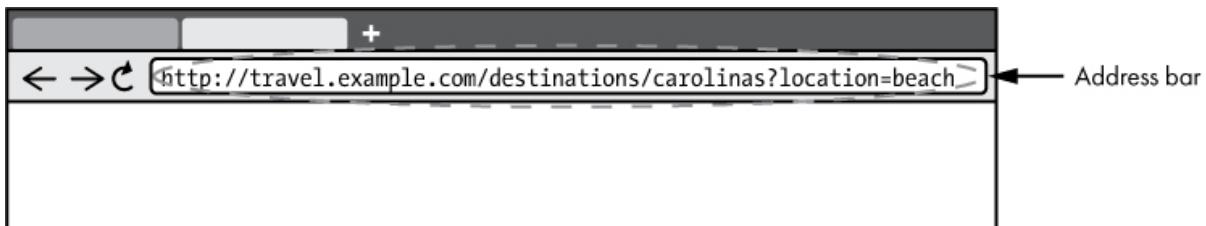


Figure 12-2: The address bar

Today, it's common for browsers to exclude the scheme, colon, and forward slashes in the address bar representation of the URL. This doesn't mean these elements of the URL are no longer used by the browser. The browser is just trying to simplify things for users. The specifics of how the URL is displayed continue to change over time, with various browsers behaving differently.

Figure 12-3 shows examples of how Google Chrome (version 77) displays URLs in its address bar.



Figure 12-3: Chrome's address bar

The top image in Figure 12-3 shows the address bar when an HTTP site is loaded. Chrome doesn't display the *http://* prefix in its address bar. Note the *Not secure* text. The lower image shows the address bar when an HTTPS site is loaded. HTTPS is the secure version of HTTP. Chrome omits the *https://* prefix but displays a padlock icon indicating this is an HTTPS site.

We've been discussing URLs in the context of web pages, but URLs extend to other resources on the web too. An image shown on a web page, for example, has its own URL, as does a script file or an XML data file. A web browser only shows the URL of the web page in its address bar, but a typical web page references various other resources by URL; the browser automatically loads those resources.

Sometimes it isn't necessary to include the scheme, hostname, or even the full path in a URL. When a URL omits one or more of these elements, it's known as a *relative URL*. A relative URL is interpreted as relative to the context in which it's found. For example, if a URL like `/images/cat.jpg` is used on a web page, the browser that loads the page assumes that the scheme and hostname of the cat photo match the scheme and hostname of the page itself.

The Linked Web

The nature of URLs, where every resource on the web has a unique address, makes it easy for one web resource to reference another. A reference from one web document to another is known as a *hyperlink*, or just a *link*. Such links are one-way; any web page can link to another page without permission or a reciprocal link. This system of pages linking to one another is what puts the "web" in World Wide Web. Documents like web pages that can be connected with hyperlinks are known as *hypertext* documents.

The Protocols of the Web

The web is delivered using *HyperText Transfer Protocol (HTTP)*, and its secure variant, *HTTPS*.

HTTP

Despite its name, HTTP isn't just for transferring hypertext; it's used for reading, creating, updating, and deleting all resources on the web. HTTP typically relies on TCP/IP. TCP ensures that data is reliably transferred, and IP handles host addressing. HTTP itself is based on a

model of *request* and *response*. An HTTP request is sent to a web server, and the server replies with a response.

Each HTTP request includes an *HTTP method*, also informally called an *HTTP verb*, that describes what kind of action the client is requesting of the server.

Some commonly used HTTP methods:

GET Retrieve a resource without modifying it.

PUT Create or modify a resource at a specific URL on the server.

POST Create a resource on the server, as a child of an existing URL.

DELETE Remove a resource from the server.

Any HTTP method can be attempted on any resource, but the server hosting a specific resource often won't allow some methods on that resource. For example, most websites do not allow clients to delete resources. Those that do allow deletion almost always require the user to log on with an account that has permission to delete content.

The most commonly used method on a typical website is `GET`. When a web browser navigates to a website, the browser performs a `GET` on the requested page. That page may include references to scripts, images, and so forth, and the browser then also uses the `GET` method to obtain those resources before the page can be fully displayed.

Each HTTP response includes an *HTTP status code* that describes the server's response. Each status code is a 3-digit number, where the most significant digit indicates the general class of response. Responses in the 100 range are informational. Responses in the 200 range indicate success. Responses in the 300 range indicate redirection. Responses in the 400s indicate an error on the client side—the request wasn't properly formed by the client. A 500 range response means the server encountered an error.

Some commonly used HTTP status codes:

200 Success. The server was able to fulfill the request.

301 Moved Permanently. The browser should redirect the request to a different URL, specified in the response.

401 Unauthorized. Authentication is required.

403 Forbidden. The user doesn't have access to the requested resource.

404 Not Found. The server didn't find the requested resource.

500 Internal Server Error. Something unexpected happened on the server.

HTTP is fairly easy to understand. It uses human-readable text to describe requests and responses. The first line of a request includes an HTTP method, the URL of the resource, and the requested version of HTTP. Here's an example:

```
GET /documents/hello.txt HTTP/1.1
```

This simply means that the client is asking the server to send it the content of */documents/hello.txt* using HTTP version 1.1. Following the request line, an HTTP request usually includes header fields that provide more information about the request and an optional message body.

Similarly, an HTTP response uses a simple text format. The first line includes a version of HTTP, a status code, and a response phrase. Here's an example of the first line of an HTTP response:

```
HTTP/1.1 200 OK
```

In this example response, the server is indicating a status code of 200 and a response phrase of *ok*. Just like HTTP requests, responses may also include header values and a message body. Figure 12-4 provides a more detailed, but still simplified, example of an HTTP request and response.

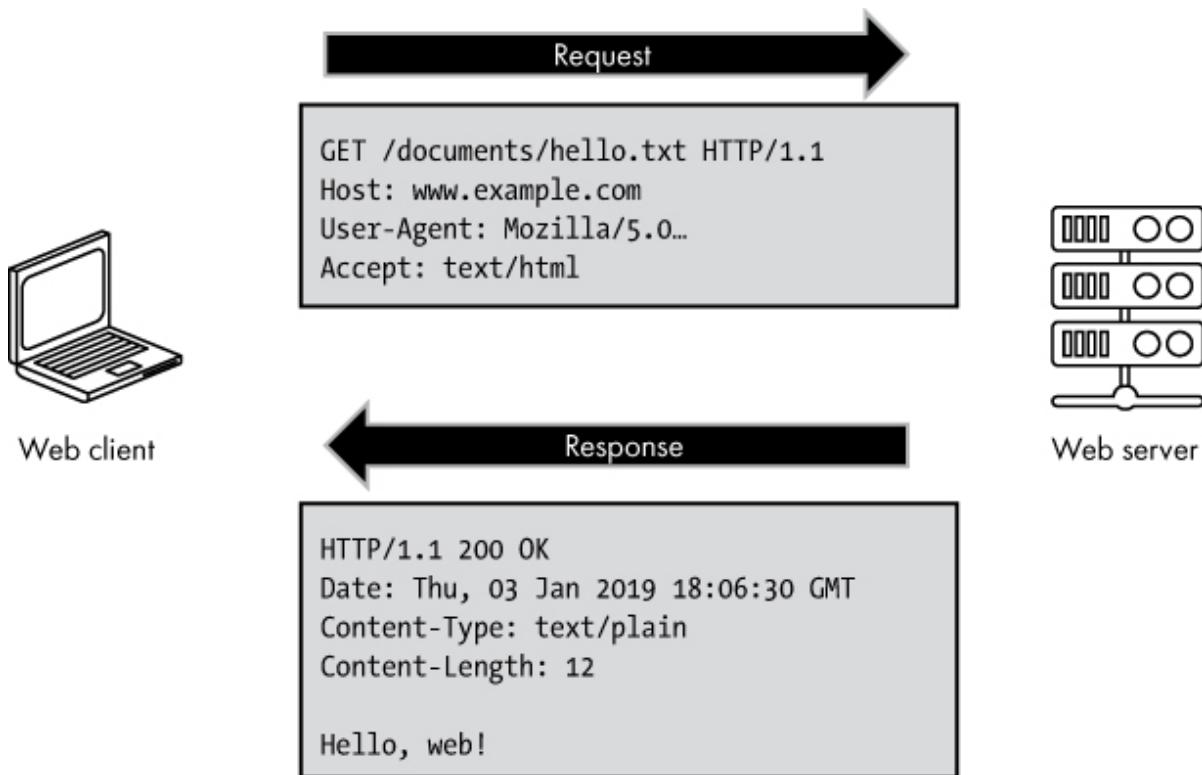


Figure 12-4: A simplified HTTP request and response

NOTE

Please see Project #36 on page 283, where you can look at HTTP network traffic.

HTTPS

A secure variation of HTTP known as *HTTPS (HyperText Transfer Protocol Secure)* is commonly used on the web to encrypt data sent over the internet. *Encryption* is the process of encoding data into a format that's unreadable. *Decryption* is the reversal of encryption, making encrypted data readable again. Cryptographic algorithms encrypt and decrypt data using a secret sequence of bytes known as a *cryptographic key*. Because keys can be kept secret, the algorithm itself can be well-known.

HTTPS uses two kinds of encryption. *Symmetric encryption* uses a single *shared key* both for encrypting and decrypting a message. *Asymmetric encryption* uses two keys (a *key pair*): a *public key* is used to encrypt data, and a *private key* is used to decrypt data. Asymmetric

encryption allows a public key to be shared freely so that anyone can encrypt and send data, whereas a private key is shared only with trusted parties who need to be able to receive and decrypt the data.

Without HTTPS, web traffic is transmitted “in the clear,” meaning it is unencrypted and can be intercepted or modified in transit by malicious parties. HTTPS helps to reduce these risks. With HTTPS, the entire HTTP request is encrypted, including the URL, headers, and body. The same is true of an HTTPS response; it’s fully encrypted. HTTPS takes an HTTP request and encrypts it using a protocol called *Transport Layer Security (TLS)*. In the past, a similar protocol called *Secure Sockets Layer (SSL)* was used, but due to security issues, it has since been deprecated in favor of TLS. When we speak of HTTPS, what we mean is HTTP encrypted with TLS.

When an HTTPS session begins, the client connects to the server with a *client hello* message with details of how it wishes to securely communicate. The server responds with a *server hello* message that confirms how the communication will occur. The server also sends a set of bytes known as a *digital certificate*, which includes the server’s public cryptographic key, used for asymmetric encryption. The client then checks if the server’s certificate is valid. If so, the client encrypts a string of bytes using the server’s public key and then sends the encrypted message to the server. The server decrypts the bytes using its private key. The server and client both use the information previously exchanged to compute a shared secret key, used for symmetric encryption. Once both client and server have the shared key, that key is used to encrypt and decrypt all communication between the client and server for the duration of the session.

HTTPS was previously only used in limited cases, for websites that dealt with particularly sensitive information. However, the web is moving to a state in which HTTPS is the norm rather than the exception. There is a growing belief that the security and privacy benefits of HTTPS make sense for most, if not all, traffic on the web. Google has encouraged this change, by marking HTTP sites as “Not

secure” in Chrome and by using the presence of HTTPS as a positive signal to its search engine, helping to boost the Google search rank of HTTPS sites.

NOTE

Please see Project #37 on page 285, where you can set up a simple web server on your local network.

The Searchable Web

For many people, the typical entry point to the web is a search. Rather than navigate to a particular URL, a user types some search terms into their browser and sees what comes up. Browser design encourages this, since browsers commonly leverage the address bar as a search box too. Even when a user wants to visit a particular site, they often perform a search for that site, then click the resulting link, rather than entering the full URL in the address bar. This is a design that enhances usability of the browser, even as it blurs the distinction between URLs and search terms, browser and search engine.

Despite the prevalence and usefulness of searching the web, the capability of searching isn’t a native feature of the web. There isn’t a standard specification for how searching should work. This means that searching, one of the key features of the web, relies on nonstandard, proprietary search engines. At the time of this writing, Google dominates the web searching space, and although there are good alternative search engines, their worldwide usage is a fraction of Google’s.

The Languages of the Web

Any content that can be saved as a file can be hosted on the web. For example, a web server can host a collection of Excel files, and they can be downloaded from the website and opened in Excel. However, a web browser is much more than just a tool for downloading files to be opened in other applications. A web browser not only downloads content, but also renders web pages. These pages can be simple

documents, or interactive web applications. To make this possible, browsers understand three computer languages, which are used to construct websites.

HyperText Markup Language (HTML) Defines the structure of a web page. In other words, it defines *what is on the page*. For example, HTML can specify that a button exists on a web page.

Cascading Style Sheets (CSS) Defines the appearance of a web page. In other words, it defines *how the page looks*. For example, CSS can specify that the aforementioned button is 30 pixels wide and blue.

JavaScript Defines the behavior of a web page. In other words, it defines *how the page functions*. For example, JavaScript can be used to add two numbers together when a button is clicked.

These three languages are used together to create the content of the web. It's worth noting that web browsers are also capable of rendering some other data types too, notably certain image, video, and audio formats, but we won't go into those in detail. Let's now dive into each of the three foundational languages of the web: HTML, CSS, and JavaScript.

Structuring the Web with HTML

HTML is a markup language that describes the structure of a web page. Note that HTML isn't a programming language. A programming language describes operations that a computer should perform, whereas a markup language describes the structure of data. In the case of HTML, the data in question represents a web page. A web page can contain various elements such as paragraphs, headings, and images. Here's an example of a simple web page described in HTML.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Cat</title>
  </head>
```

```
<body>
  <h1>Thoughts on a Cat</h1>
  <p>This is a cat.</p>
  
</body>
</html>
```

You see a number of items enclosed in less than (<) and greater than (>) signs. These are known as *HTML tags*, sets of text characters used to define the parts of an HTML document. As an example, the tag used to indicate the start of a paragraph is `<p>`. A corresponding tag is used to indicate the end of a paragraph, `</p>`. Note the slash in the end tag, differentiating it from the start tag. An *HTML element* is a portion of the page beginning with a start tag, ending with an end tag, and including the content between the tags. For example, this is an HTML element: `<p>This is a cat.</p>`. In truth, not all elements require an end tag. For example, the `img` element, used for representing an image, needs no end tag. You can see this in the previous HTML code example.

Figure 12-5 shows how the example HTML might be rendered in a web browser.



Figure 12-5: Our example web page rendered in a web browser

The document intentionally included no information about how it should be presented, so a browser uses a default font and size for the heading and paragraph. In this example, the browser also defaulted to black text on a white background—again, that wasn’t specified in the document. Since this HTML example contains no style information, different browsers can choose to render this page slightly differently.

Let’s examine the HTML code example more closely. The first line of an HTML document declares that the file is an HTML document, like so: `<!DOCTYPE html>`. After that, an HTML document is structured as a tree, with parent elements and child elements. The `<html>` tag is the top-level parent tag; everything is enclosed between `<html>` and `</html>`. You can interpret those two tags as “HTML starts here” and “HTML ends here.” This makes sense—everything in an HTML document should be HTML! The `<html>` tag also contains an attribute called `lang` that identifies the language of this document as `en`, the code for English.

The `html` element has two child elements: `head` and `body`. Elements contained in the head (between `<head>` and `</head>`) describe the document, whereas elements in the body (between `<body>` and `</body>`) make up the contents of the document.

In our example, the head contains two elements: a `meta` element describing the character set used to encode our document, and a `title`. Browsers typically show the title text on a page's tab and use it as a default name when a user adds a bookmark or favorite. Search engines use the title text when showing results. For those reasons it's important for web developers to give meaningful titles to their pages.

The body in our example includes an `<h1>` tag, which is used for a heading element. Heading tags of `<h1>` through `<h6>` are available, with `h1` intended to be used as the highest level of section headings, and `h6` as the lowest level. A paragraph follows the heading, indicated with a `<p>` tag, and after that an image is included using ``. Note that the bytes of the image itself aren't present in the HTML. Instead, the `` tag simply references an image file by a relative URL (`cat.jpg`). To fully load this page, a browser needs to make a separate HTTP request to download the image. In this example, the image URL is simply a filename, meaning it's hosted on the same server and in the same path as the document itself. If the image were hosted elsewhere, a URL with a path or a server name could be used. The `` tag also has an `alt` attribute, which provides alternate text that describes the image. This is used in cases in which the image cannot be rendered, such as when a text-only browser or a screen reader that reads the contents of the page aloud is being used.

You may have noticed that the earlier HTML code used indentation to show the nesting of various elements on the page. For example, the `<h1>` and `<p>` tags are indented to the same level, showing that they are child elements of the `<body>` tag. This is a common practice in web development to improve readability of HTML, but it's not required. In fact, whitespace beyond a single space or tab doesn't matter in an HTML document! We could remove all the extra spaces, tabs, and line breaks, leaving us with the HTML all on a single line, and the

document would render the same way in a browser. Web browsers ignore extra whitespace, so spacing out elements on a page is only helpful to developers.

NOTE

Please see Project #38 on page 287, where you can have your local website return a document structured with HTML rather than simple text.

The HTML elements we've covered so far are only a small percentage of the total elements recognized by web browsers. We won't exhaustively cover all of HTML here; it is well documented online. The specifications for HTML were previously maintained by two organizations: the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). The last major version of HTML to receive "Recommendation" status from the W3C was *HTML5*. In 2019, the two organizations agreed that ongoing development of the HTML standard will be handled principally by the WHATWG, in what is known as the *HTML Living Standard*, which is continually maintained.

Modern browsers attempt to support both current and older versions of HTML, since plenty of web content was authored with earlier HTML standards in mind. In the past, browsers introduced nonstandard HTML elements, some of which eventually became standardized, while others fell out of use and lost support. Web browser developers must balance innovation with adherence to standards, while still supporting less-than-perfect HTML that's sometimes found on the web. Web browsers are ever evolving, and different browsers sometimes render the same content differently. This means that web developers regularly test their creations on multiple browsers to ensure consistent behavior.

Styling the Web with CSS

In our earlier example HTML, we used tags that described the structure of a document, but those tags did not convey any information about

how the document should be presented. This was intentional; we want to keep structure and style separate. A division between the two allows for the same content to be rendered with different styles in different contexts. For example, most web content should be rendered differently on a large PC screen versus a small mobile screen.

Cascading Style Sheets (CSS) is the language used to describe the style of a web page. A style sheet is a list of rules. Each rule describes a style that should be applied to a certain part of the page. Each rule includes a selector, which indicates what elements on the page should have the style applied. The *cascading* term refers to the ability for multiple rules to apply to the same element. Let's look at a simple example:

```
p {  
    font-family: Arial, Helvetica, sans-serif;  
    font-size: 11pt;  
    margin-left: 10px;  
    color: DimGray;  
}  
  
h1 {  
    font-family: 'Courier New', Courier, monospace;  
    font-size: 18pt;  
    font-weight: bold;  
}
```

In this example, style rules are defined for paragraph (`p`) elements and heading 1 (`h1`) elements. When this CSS is applied to a page, all paragraphs on that page use the specified font, with a size of 11 point, a left margin of 10 pixels, and gray text. Similarly, `h1` headings use the specified bolded font with a size of 18 point. Note that `font-family` is a list of fonts, not just a single font. This means that a web browser should try to find a matching font, starting with the leftmost font and proceeding to the right until a font match is found. Not every client device has the first choice of font installed; specifying multiple fonts increases the chance that a matching font will be available.

You can apply a style sheet to a web page in a couple of ways. One option is to include the CSS rules within a `style` element on the page. For example:

```
<style>p {color: red};</style>
```

This isn't ideal, because the style and structure are now closely related. A better option is to specify the CSS rules in a separate file, also hosted on the web. This approach keeps our HTML and CSS completely separate, and it allows multiple HTML files to use the same style sheet. This way we can change a CSS rule, and it will apply to multiple pages at once. A single element in the HTML's head section can be used to apply the style sheet rules from a CSS file, like so (where `style.css` is the URL of the CSS file to apply):

```
<link rel="stylesheet" type="text/css" href="style.css">
```

If we apply this style sheet to our example cat page, we see these changes to the heading and paragraph text, as shown in Figure 12-6.



Figure 12-6: Our example web page with CSS applied

NOTE

Please see Project #39 on page 288, where you can update your cat web page with some CSS.

This CSS example is simple, but CSS allows for much more advanced styling as well. If you’re familiar with the amazing variety of visual styles to be found on the web, then you have already seen the power of CSS in action.

Scripting the Web with JavaScript

The web was originally envisioned as a means of sharing information through hypertext documents. HTML gives us that capability, and CSS gives us a method of controlling the presentation of such documents. However, the web evolved into a platform for interactive content, and JavaScript became the standard means for enabling interactivity. *JavaScript* is a programming language that enables web pages to respond to users’ actions and programmatically perform various tasks. With JavaScript, a web browser becomes not just a document reader, but a full application development platform.

JavaScript is an interpreted language; it isn’t compiled to machine code before it’s delivered to the browser. Web servers host JavaScript code in text format, and that code is downloaded by a browser and interpreted at runtime. That said, some browsers use a *just-in-time (JIT) compiler* that compiles JavaScript at runtime, leading to increased performance. Some developers *minify* their JavaScript before deploying it, removing whitespace, comments, and generally reducing the size of the script. Minifying JavaScript can improve the load time of a website. Minification isn’t the same as compilation; the minified file is still high-level code, not compiled machine code.

JavaScript has a syntax that’s similar to C and other languages that borrowed from C (such as C++, Java, and C#). However, the similarity is superficial, as JavaScript is quite different from those languages. Don’t let the name confuse you: JavaScript has little to do with Java. The language is object-oriented but fundamentally relies on *prototypes* rather than classes. That is, an existing object, rather than a class, acts a template for other objects.

JavaScript interacts with an HTML page using a browser-supplied representation of the page called the *Document Object Model (DOM)*. The DOM is a hierarchical tree structure of page elements, and it can be programmatically modified. An update to an element in the DOM causes the browser to update the element on the displayed web page. JavaScript includes methods for working with the DOM, and by using these methods, JavaScript code can both respond to events that occur on the page (such as the click of a button) and change the contents of the rendered page.

Let's look at part of a simple script that interacts with our example page. The script adds the text `Meow!` to our page's paragraph every time the cat photo is clicked (or tapped on a touchscreen).

```
document.getElementById('cat-photo').onclick = function() {
  document.getElementById('cat-para').innerHTML += ' Meow!';
};
```

The first line here adds an event handler that runs when the cat photo is clicked. The event handler code is defined on the next line, and it tells the browser to add the text `Meow!` to the paragraph. Since this is defined as an event handler, the code only runs when the image click event occurs. Note that the script references the photo and paragraph by IDs, `cat-photo` and `cat-para`, respectively. HTML elements can be given IDs; this allows us to easily reference them programmatically. Our script only works if we add these IDs to our HTML. Here is the updated HTML that references the script (named `cat.js`) and adds the needed IDs.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Cat</title>
    <link rel="stylesheet" type="text/css" href="style.css">
    <script src="cat.js"></script>
  </head>
  <body>
    <h1>Thoughts on a Cat</h1>
    <p id="cat-para">This is a cat.</p>
    
```

```
</body>  
</html>
```

Once the script code is saved as `cat.js`, and the HTML is updated as shown, then reloading the page and clicking the cat image appends `Meow!` to our paragraph. If we click the image multiple times, we end up with something like what is shown in Figure 12-7.



Figure 12-7: Our example web page after running JavaScript code to append text

NOTE

Please see Project #40 on page 289, where you can update your web page with JavaScript.

JavaScript can be used to build full applications that run in a web browser. The previous example is just a taste of what it can do. JavaScript is standardized in a specification known as *ECMAScript*.

Various browsers implement script engines that attempt to comply with all or part of the ECMAScript standard, which is updated regularly.

Structuring the Web's Data with JSON and XML

Websites aren't the only type of content available on the web. A *web service* provides data over HTTP and is intended to be interacted with programmatically. This is in contrast to a website that returns HTML (and related assets) and is intended for user consumption via a web browser. Most end users never directly interact with web services, although the websites and apps we use are often underpinned by web services.

Imagine that you run a website with information about local bands that perform in your city. The site contains a profile of each band, including band members, background, where the band will be playing, and so forth. An end user can visit your website and easily read up on their favorite musicians. Now, let's say you're approached by an app developer who wants to include the latest information from your website in their app. However, the app has its own presentation that's totally different from your web pages—the developer doesn't want to just display your web pages in the app. They need a way to get at the underlying data on your site. They could try to programmatically read your web pages and extract the relevant information, but this process is complicated and error-prone, particularly if your site's layout changes.

You could make things much easier for this developer by providing a web service that presents the data from your site in a format other than HTML. Although HTML does provide a certain structure, it's a structure that describes a document (headers, paragraphs, and so on) and provides little insight into the data types referenced in that document. HTML makes sense for a human reader, but it's difficult for software to parse. So what format should your web service use to structure data about bands? The most common general-purpose data formats in use today by web services are XML and JSON.

Extensible Markup Language (XML) has been around since the 1990s and is a popular means of exchanging data over the web. Like HTML, it's a text-based markup language, but rather than having a set of predefined tags, XML allows for custom tags that describe your data. In the case of our fictitious band information service, we might define a `<band>` tag and a `<concert>` tag. Let's look at an imaginary band described using XML:

```
<band name="The Highbury Musical Club">
  <bandMembers>
    <member name="Jane Fairfax" instrument="Piano" />
    <member name="Emma Woodhouse" instrument="Guitar" />
    <member name="Harriet Smith" instrument="Percussion" />
    <member name="Frank Churchill" instrument="Vocals" />
  </bandMembers>
  <upcomingConcerts>
    <concert location="Donwell Abbey" date="August 14, 2020" />
    <concert location="Hartfield" date="November 20, 2020" />
  </upcomingConcerts>
</band>
```

As you can see, the specific XML tags and their attributes are tailored to our needs, while the general structure of start tags, end tags, and tree hierarchy follows a pattern similar to HTML. The flexibility of XML, where tags can be arbitrarily defined, means that both the producer and the consumer of the XML need to agree on the expected tags and their meanings. This is true of HTML as well, but with HTML, all parties agree to a standard. In the case of XML, only the general format is standardized while the specific tags vary.

XML is a popular method for sharing data over the web, with many web services using XML as their primary means of representing data. However, XML is verbose and parsing it properly can be tricky.

JavaScript Object Notation (JSON), like XML, is a method of describing data in a text format. JSON avoids using markup tags and instead embraces a style that's similar to JavaScript's syntax for describing objects, hence the name. In JSON, objects are wrapped in curly braces (`{` and `}`) and arrays (collections of objects) are enclosed in brackets (`[` and `]`). Its syntax is terser than XML, which is helpful in reducing the size of data transmitted over a network. The popularity of

JSON rose in the 2010s, when it began to displace XML as the preferred data format for new web services. Here is the same imaginary band described in JSON:

```
{  
  "name": "The Highbury Musical Club",  
  "bandMembers": [  
    {  
      "name": "Jane Fairfax",  
      "instrument": "Piano"  
    },  
    {  
      "name": "Emma Woodhouse",  
      "instrument": "Guitar"  
    },  
    {  
      "name": "Harriet Smith",  
      "instrument": "Percussion"  
    },  
    {  
      "name": "Frank Churchill",  
      "instrument": "Vocals"  
    }  
  "upcomingConcerts": [  
    {  
      "location": "Donwell Abbey",  
      "date": "August 14, 2020"  
    },  
    {  
      "location": "Hartfield",  
      "date": "November 20, 2020"  
    }  
}
```

Both XML and JSON ignore extra whitespace, so just like with HTML, we can remove all extra spaces, tabs, and line breaks without affecting how the data is interpreted. Doing so produces a fairly compact rendering of data, particularly in the case of JSON.

XML and JSON are not formats meant for direct rendering in a web browser. Opening JSON or XML content in certain browsers may cause the browser to display something (perhaps a lightly formatted version of the data), but really JSON and XML aren't intended to be directly consumed by web browsers. They are meant to be read by code that in turn does something useful with the data. Perhaps that code is a smartphone app that shows information about what bands are playing

nearby, as in our example. Or maybe the code is client-side JavaScript that transforms JSON into HTML for a browser to display.

Web Browsers

Now that we've covered the languages used to describe the web, let's take a look at software on the client side of the web, the web browser. The first web browser was called *WorldWideWeb* (not to be confused with the subject of this chapter). It was developed by Tim Berners-Lee in 1990. This first browser was the client for the first web server, *CERN httpd*. In a few years *WorldWideWeb* was supplanted by *Mosaic*, a browser that helped popularize the web. The next major browser release was *Netscape Navigator*, which also had a large following. In 1995, Microsoft released their first browser, *Internet Explorer*, as a direct competitor to *Netscape Navigator*, and *Internet Explorer* became the dominant browser of its time. Today, the browser landscape has shifted dramatically, and at the time of this writing, the dominant browsers are *Google Chrome*, *Apple Safari*, and *Mozilla Firefox*.

Rendering a Page

Let's now look at the process a web browser goes through to render a page. A typical visit to a website starts with a request of the default page of a site (such as `http://www.example.com/`) or a request of a specific page on the site (such as `http://www.example.com/animals/cat.html`). A user may enter this URL directly in the address bar, or the user could arrive at this URL by following a link. In either case, the browser requests the contents at the specified URL. Assuming the URL is valid and represents a web page, the server responds with HTML.

The web browser must then take the returned HTML and generate a DOM representation of the page. The HTML may contain references to other resources, like images, scripts, and style sheets. Each of these resources has its own URL, and the browser makes separate requests for each resource, as illustrated in Figure 12-8.

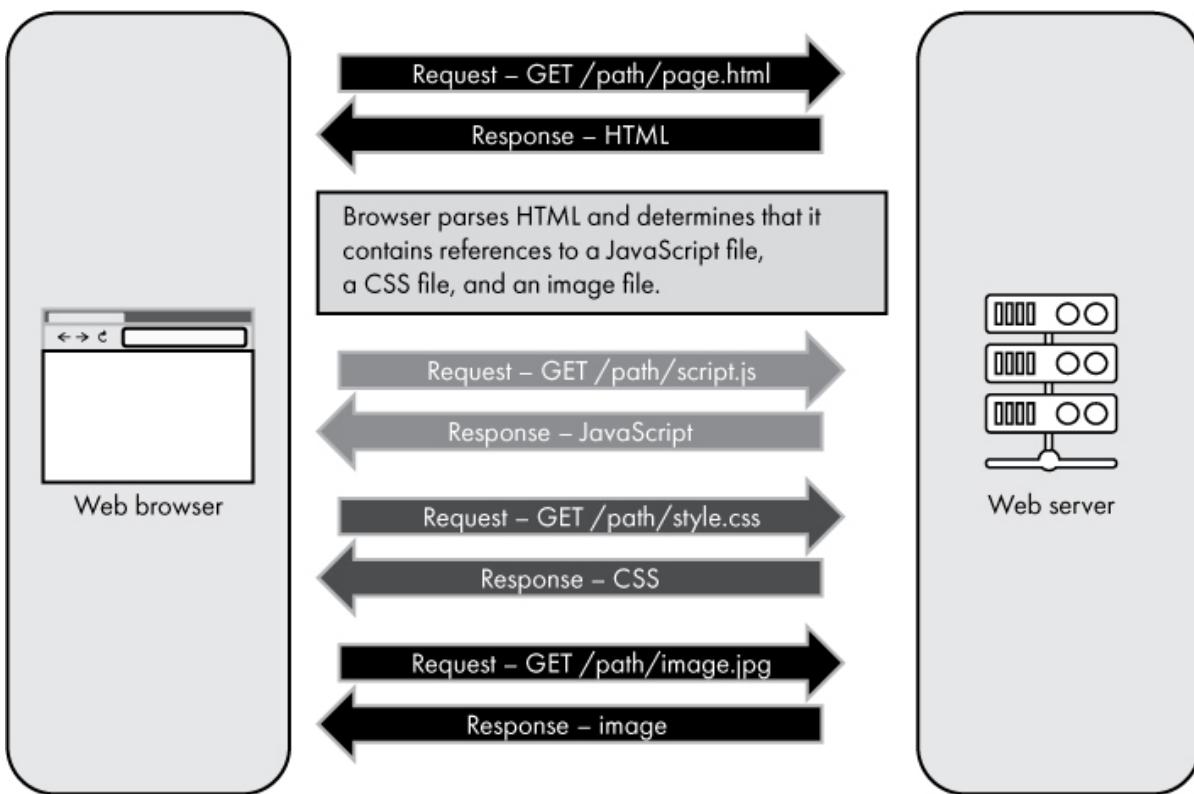


Figure 12-8: A web browser requests a page and its referenced content

Once the browser has retrieved the various resources of the page, it displays the HTML, using any specified CSS to determine the appropriate presentation. Any scripts are handed off to a JavaScript engine to run. JavaScript code may immediately make changes to the page, or it may register event handlers that run later when certain events occur. JavaScript code may also request data from a web service and use that data to update the page.

Web browsers consist of a rendering engine (for HTML and CSS), a JavaScript engine, and a user interface that ties things together. Although the user interface provides the look and feel of the browser itself (such as the appearance of the back button and address bar), it's the rendering engine and JavaScript engine that determine how websites are presented and behave (this includes things like how the page is laid out and how it responds to input). Since each rendering engine and JavaScript engine handle things slightly differently, a web page may look or act differently when viewed on different browsers.

Ideally, all browsers would render content the same way, exactly as the site developer intended, but that's not always the case. At the time of this writing, only three major rendering engines are in active development: WebKit, Blink, and Gecko.

WebKit is the rendering engine and JavaScript engine for Apple's Safari browser. It's also used in applications found in the iOS App Store, since Apple requires all iOS apps that display web content to use this engine. *Blink*, which is a fork of WebKit, is the rendering engine for the *Chromium* open source project, which also includes the *V8* JavaScript engine. Chromium is the basis for Google Chrome and Opera. In December 2018, Microsoft announced that the *Microsoft Edge* browser would also be Chromium-based; Microsoft chose to halt development of its own rendering and JavaScript engines. That leaves only one major browser that doesn't trace its roots to WebKit—Mozilla Firefox, which has its own *Gecko* rendering engine and *SpiderMonkey* JavaScript engine.

NOTE

A software fork occurs when developers make a copy of a project's source code and then make changes to that copy. This allows the original and forked projects to coexist as separate software.

The User Agent String

The formal, technical term for a web browser is a *user agent*. This term can be applied to other software as well (anything that acts on behalf of a user), but here we're talking specifically about web browsers. This term pops up in technical documentation about the web, although it's rarely used outside of formal communication. That said, one place where the term is used in practice is the *user agent string*. When a browser makes a request to a web server, it commonly includes a header value called `User-Agent` that describes the browser. As an example, here is the user agent string sent by Chrome (version 71) on Windows 10:

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/71.0.3578.98 Safari/537.36

This may seem contradictory. What does all this mean?

The first entry, `Mozilla/5.0`, is a holdover from the early days of the web. Mozilla was the user agent name for Netscape Navigator, and many sites specifically looked for “Mozilla” in the user agent string as an indicator to send the cutting-edge version of their website to the browser. At the time, other browsers wanted to get the best versions of websites too, so they identified themselves as Mozilla, even though they weren’t Mozilla at all. Fast forward to today, when essentially *every* browser identifies itself as Mozilla, and we find that portion of the user agent string fairly meaningless.

The next section in parentheses, `(Windows NT 10.0; Win64; x64)`, specifies the platform on which the browser is running.

Following that is the rendering engine, `AppleWebKit/537.36` in this case. As mentioned earlier, Chrome’s Blink engine is a fork of WebKit and still identifies itself as such. The following text, `(KHTML, like Gecko)`, is just a further elaboration on this; KHTML is a legacy engine that WebKit was based on.

Now we get to the actual browser name and version,
`Chrome/71.0.3578.98`.

Finally, we have an awkward mention of Apple’s browser `safari/537.36`, included for sites that give Safari special treatment. By including this text, Chrome attempts to ensure that those sites send it the same content that Safari would receive.

That’s a rather complicated way to identify Chrome, but other browsers do the same kind of thing to ensure compatibility with all manner of websites. This complexity is an unfortunate side effect of historically fragmented capabilities in different browsers and websites that tried to send tailored versions of their content based on the particular browser. Browsers evolved so that today there is less variation in browser capabilities. However, many websites didn’t evolve and still send content tailored for specific browsers, forcing modern browsers to continue to trick old sites into believing they are communicating with a different browser.

Web Servers

So far, we've focused primarily on the technologies used on the client side of the web. Web browsers speak a common trio of languages: HTML, CSS, and JavaScript. What about on the web's server side? What languages and technologies are used to power web servers? In short, any programming language or technology can be used on a web server, as long as that technology can communicate over HTTP and return data in a format that the client understands.

Broadly speaking, websites are designed as either static or dynamic. A *static website* returns HTML, CSS, or JavaScript that was built ahead of time. Typically, the content of the site is stored in files on the server, and the server simply returns the contents of those files without modification. This means that any required runtime processing must be implemented in JavaScript that runs in the browser. On the other hand, a *dynamic website* performs processing on the server, generating HTML when a request comes in.

In the early days of the web, nearly everything was static. Pages were simple HTML, and there was little interactivity. As time went on, developers began adding code that ran on the web server, allowing the server to return dynamic content or accept a file upload or form submission from a user. This trend continued, and it became commonplace for requests to go through server-side processing before the server would respond.

Let's look at how server-side processing on a dynamic website typically works, as illustrated in Figure 12-9. Assume that the dynamic website represented in Figure 12-9 is a blog. A browser makes a request for a blog post. When the web server receives the request for the blog post, it reads the requested URL and determines that it needs to generate HTML. Code on the server then queries a database (which may be on the web server or on another server), retrieves the relevant blog text data, formats that text as HTML, and then responds to the client with that HTML. This approach is useful because it allows the content of a site to be managed separately from the website's code, but

dynamic sites also have some drawbacks. The increased complexity on the server means more work to set things up, a slower response at runtime, a potentially heavy load on the server, and an increased risk of security problems.

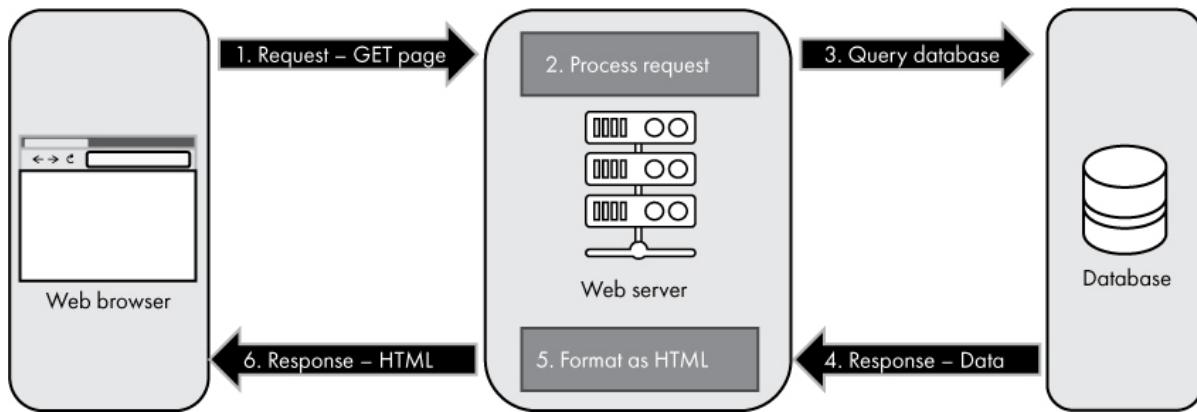


Figure 12-9: A typical dynamic website handles a request

Recently, there has been a trend to move back to static sites where possible. The flow of a page request on a static site is shown in Figure 12-10.

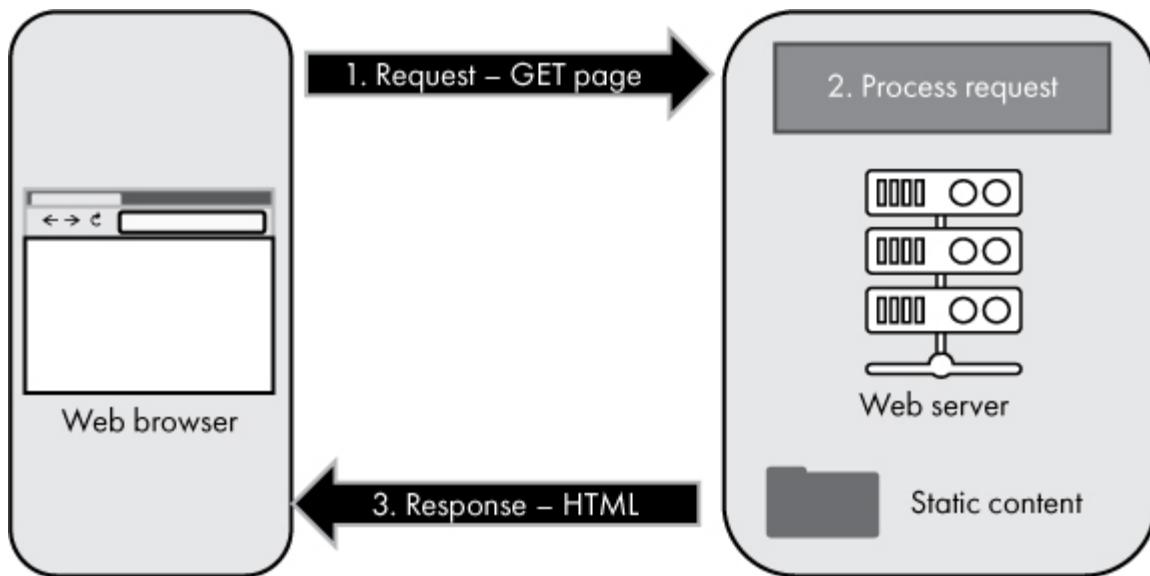


Figure 12-10: A static website handles a request

As shown in Figure 12-10, the static website's server-side processing is simplified, as compared to the dynamic site. The server-side

processing on a static site is simply a matter of returning the static file that matches the requested URL. The content has already been built; the server does not need to retrieve raw data and format it. Reducing the complexity on the server side generally means simpler, faster, and more secure sites.

It is important to understand that in this context, the terms *static* and *dynamic* are from the server's perspective, not the user's. A static site's content comes unchanged from files on the server, whereas a dynamic site's content is generated on the server. The terms aren't a description of how a user experiences the site, such as whether the site is interactive or if content is automatically updated. These experiences can be achieved using JavaScript in the browser, sometimes in conjunction with a separate web service, regardless of whether the website itself is static or dynamic.

If you're hosting a static site, all you need is web server software that can respond to requests for your static files and serve the contents of those files. No custom code required. Many software packages and online services are available for hosting static sites. Typically, the software for serving a static site is configured to point to a directory of files on the server, and when a request comes in for a certain file, the server simply returns the contents of that file. For example, if the files for the website at *example.com* reside in a directory on the server called */websites/example*, then a request for *http://example.com/images/cat.jpg* maps to */websites/example/images/cat.jpg*. The web server simply reads the matching file from its local directory and returns the bytes contained in that file to the client. The website developed in Projects #37 through #40 is an example of a static site.

If you're building a dynamic website or web service, either you can use existing software that manages your content and serves up dynamic pages, or you can write your own custom code that generates web content. Assuming you're writing custom code, you'll find things are quite different on the server side as compared to the client side of web development. Any programming language, any operating system, any

platform can be used for a web server. Anything goes, as long as the web server responds over HTTP and returns data in a format that the client understands! The client doesn't care what technologies were used to generate HTML or JavaScript; it just needs a response in a format it can handle.

Since it doesn't really matter to clients what technology is used on the web server side, many options are available to developers who wish to write code that runs on the server. Client-side web development is limited to the trio of HTML, CSS, and JavaScript, whereas server-side web development can take place in Python, C#, JavaScript, Java, Ruby, PHP, and more. Server-side web development often includes interfacing with a database of some kind. In the same way that any programming language can be used on the server, any kind of database can be used for server-side web development.

Summary

In this chapter we covered the web—a set of distributed, addressable, linked resources, delivered by HTTP over the internet. You learned how web pages are structured with HTML, styled with CSS, and scripted with JavaScript. We looked at web browsers, which are used to access content on the web, and we examined web servers—the software that hosts web resources. In the next chapter, we'll look at some trends in modern computing, and you'll have a chance to complete a final project that ties together various concepts found throughout this book.

PROJECT #36: EXAMINE HTTP TRAFFIC

In this project, you'll use Google Chrome or Chromium to examine HTTP traffic between a web browser and a web server. You can either use Chrome on a Windows PC or a Mac, or use the Chromium web browser on your Raspberry Pi. The following steps assume you're using a Raspberry Pi, but the process is similar on a Windows PC or a Mac; just use Chrome instead of Chromium.

1. If you aren't using the graphical desktop on your Raspberry Pi, switch to it now. Unlike previous projects, this project cannot be completed from a terminal window.
2. Click **Raspberry** (icon in the upper left corner) ▶ **Internet** ▶ **Chromium Web Browser**.
3. Go to a website, such as <http://www.example.com>.
4. Press the F12 key (or CTRL-SHIFT-I) to open the developer tools (DevTools), shown in Figure 12-11.

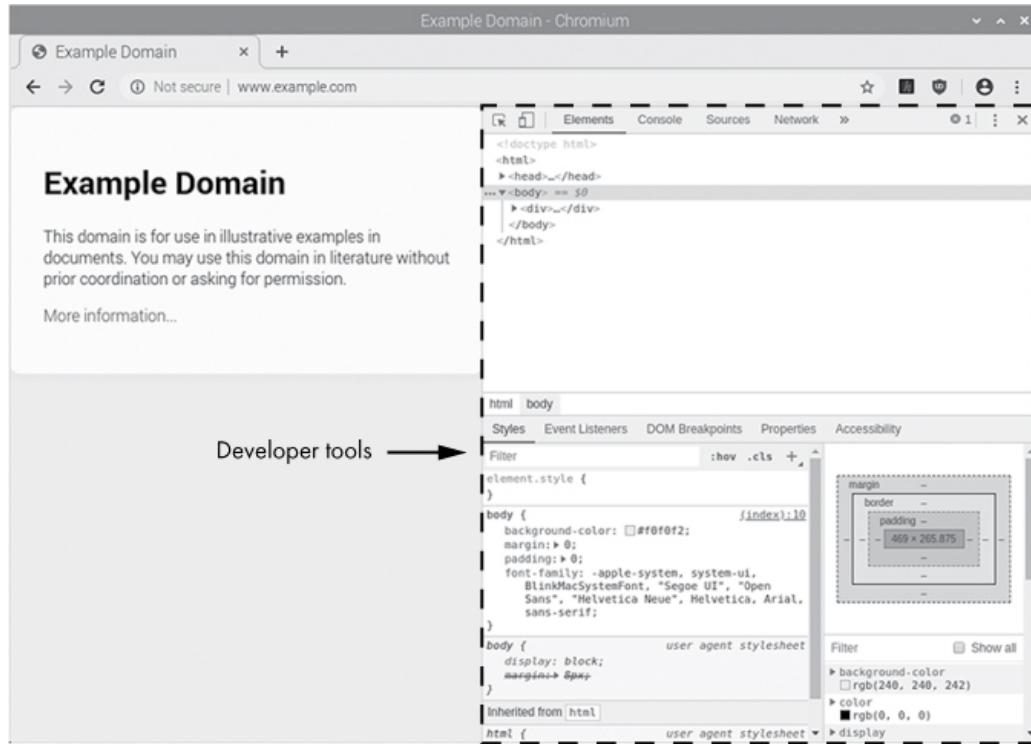


Figure 12-11: Developer tools in Chromium

5. On the DevTools menu, choose the **Network** menu item.
6. Press F5 (or hit the reload icon) to reload the page. You'll see the HTTP requests that are made to load the page you're currently visiting.
7. If you actually use www.example.com you'll likely see a fairly boring request. If you want to see something more interesting, visit a more complicated site and watch the network requests, as shown in Figure 12-12.

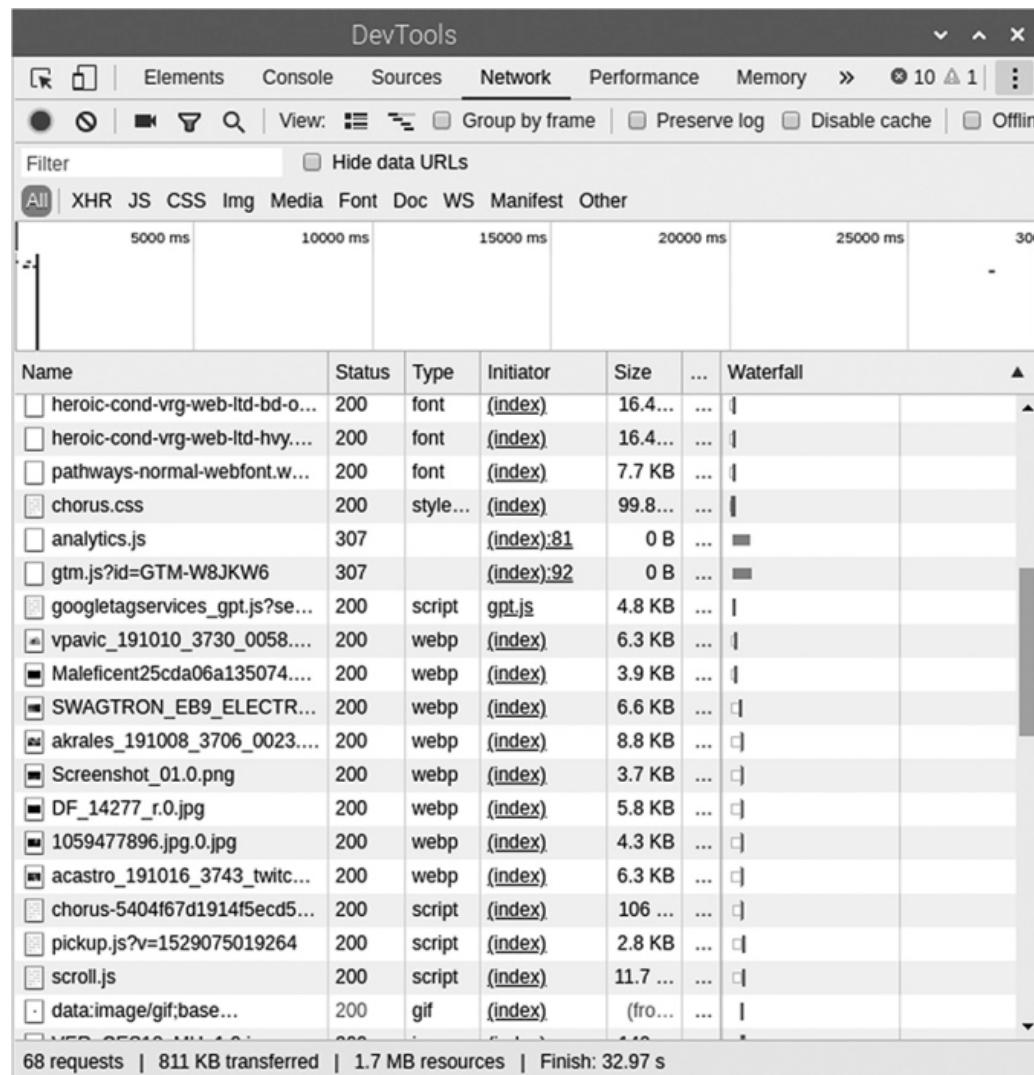


Figure 12-12: Example of HTTP traffic for a website shown in Chromium's DevTools

8. Each line represents a request to the web server. You can see the resource name that was requested, the status of the request (200 means success), and more.
9. You can click each line and see the specifics of the request, such as the headers and the content returned.

I recommend you try this on several websites to get a feel for the number of requests made for a site. You may be surprised at how much content is transferred!

PROJECT #37: RUN YOUR OWN WEB SERVER

In this project, you'll set up a Raspberry Pi to act as a web server. You'll use Python 3 to do this, so you can actually follow these steps on any device with Python 3 installed, although the steps here were written with the Raspberry Pi in mind. Our simple website will return the contents of a file when it receives a request.

Open a terminal window and create a directory that will hold the files your website will serve, and then set that new directory as your current directory.

```
$ mkdir web  
$ cd web
```

When a request is made to the root of your website, the web server software looks for a file named *index.html* and returns the contents of that file to the client. Let's create a very simple *index.html* file:

```
$ echo "Hello, Web!" > index.html
```

That command creates a text file named *index.html* with the text **Hello, Web!** in the file. You can view the contents of the text file to ensure it was created successfully by opening the file in a text editor, or you can display the contents in the terminal like so:

```
$ cat index.html
```

Once your file is in place, let's use Python's built-in web server to serve up your **Hello, Web!** message to anyone who connects.

```
$ python3 -m http.server 8888
```

This command tells Python to run an HTTP server on port **8888**. Let's test this out to see if it's working as expected. Open another terminal window on your Raspberry Pi. From this second terminal window, enter the following command to make a GET request to the root of your new website:

```
$ curl http://localhost:8888
```

The **curl** utility can be used to make HTTP GET requests, and **localhost** is a hostname that refers to the computer you're currently using. This command tells the **curl** utility to perform an HTTP GET to port **8888** on the local computer. You should see the text **Hello, Web!** returned. Also, back in the original terminal, you should see that a GET request came in.

Now, let's try connecting to your website from a web browser. From the Raspberry Pi desktop, open the Chromium web browser. In the address bar, enter **http://localhost:8888**. You should see the text from your website appear in the browser, as shown in Figure 12-13.

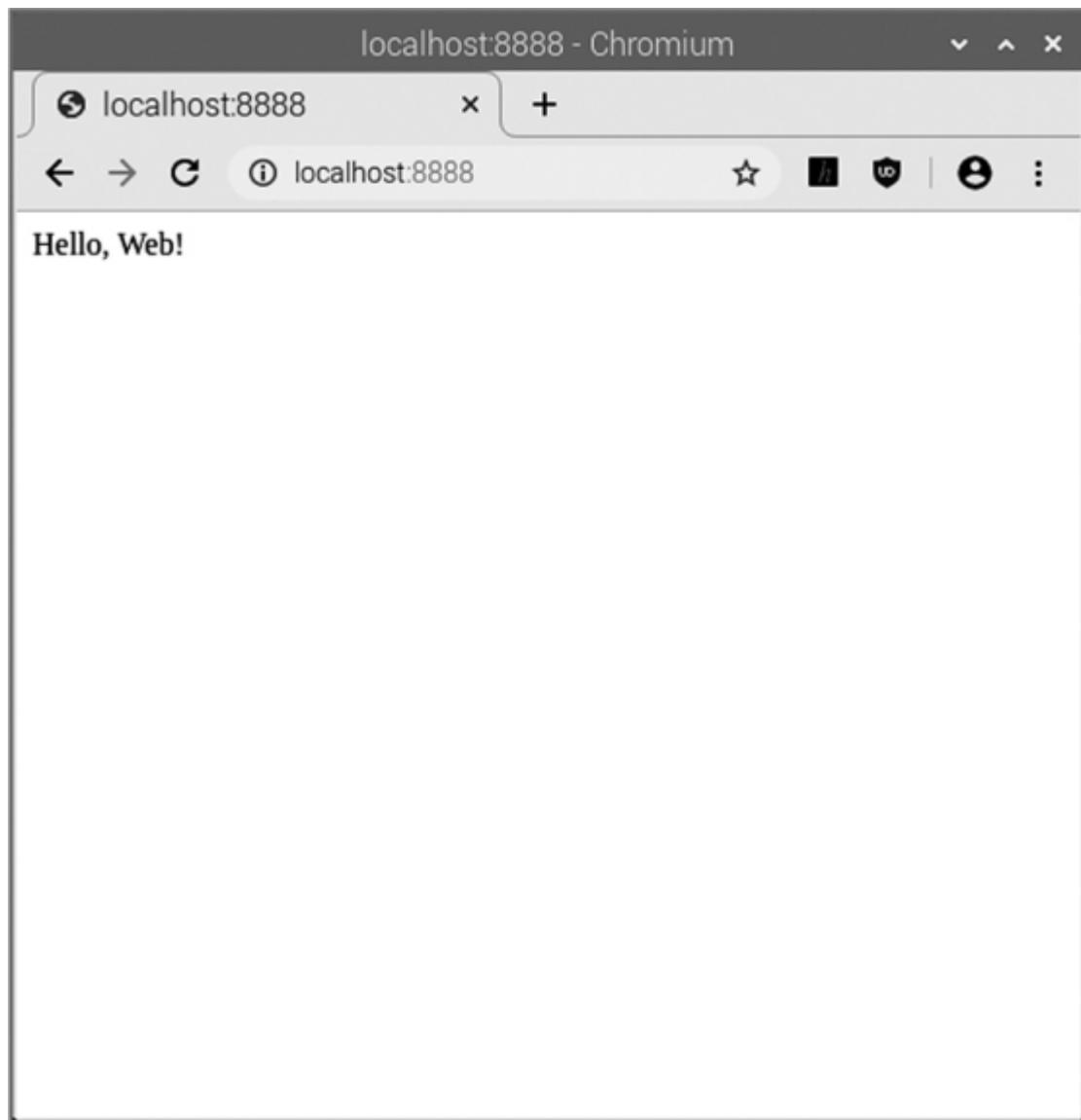


Figure 12-13: Connecting to a local web server using the Chromium browser

Now try connecting to your website from another device. For this to work, the second device has to be on the same network as your Raspberry Pi. For example, they should both be on the same Wi-Fi network. Or, if your Raspberry Pi has a public IP address (see Project #34 on page 259), then your website is available to any device on the internet! First, get your Raspberry Pi's IP address by running the following command in the second terminal window:

```
$ ifconfig | grep inet
```

This likely returns several IP addresses. You can't use `127.0.0.1` when connecting from a remote device, so choose another IP address assigned to your Raspberry Pi. Once you have the IP address, open a browser on another device. This can be a smartphone,

laptop, or really any device on your network that has a web browser. In the browser window, enter the following in the address bar: `http://w.x.y.z:8888` (replacing `w.x.y.z` with the IP address of your device). Press ENTER or the appropriate button in the browser to navigate to that address. You should see `Hello, Web!` appear in the browser.

If this didn't work for you, and your Raspberry Pi does not have a public IP address, make sure the two devices are on the same physical local network. Also, sometimes the Python web server becomes unresponsive to new requests. If the web server stops responding, you can restart it. To stop the web server, go to the terminal where the server command was executed, and press CTRL-C on the keyboard. Then restart the server by running the `python3 -m http.server 8888` command again (press the keyboard up arrow to get the last command).

Once you have your site working, try editing the `index.html` file and change the message to say whatever you want. You can use the text editor of your choice to do this. Once your `index.html` file is updated, reload the web page in a web browser to see your changes!

If you don't want other devices to be able to access your website, you can restrict things so that only requests from the Raspberry Pi itself get a response. Running the Python web server with the `--bind` option can accomplish this, like so:

```
$ python3 -m http.server 8888 --bind 127.0.0.1
```

To run the web server with the `--bind` option, you first need to stop any running instance of the web server (press CTRL-C on the keyboard).

PROJECT #38: RETURN HTML FROM YOUR WEB SERVER

Prerequisite: Project #37.

In this project, you'll update your local web server to return HTML instead of simple text. Use the text editor of your choice to open `index.html` (that was created in Project #37) and replace all the text in the file with the following HTML. This is the same HTML code that was discussed in the chapter. You don't need to worry about the indentation of each line, since extra whitespace in HTML doesn't matter.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Cat</title>
  </head>
  <body>
```

```
<h1>Thoughts on a Cat</h1>
<p>This is a cat.</p>

</body>
</html>
```

Once your file has been updated, you'll use Python's built-in web server again. If it isn't already running, start it with this command. Just be sure that your terminal window is currently in the *web* directory before running the command.

```
$ python3 -m http.server 8888
```

Now, use a web browser to connect to your web server as you did in Project #37. You should see the page rendered, but without the cat photo. If you look at the terminal window where you ran the Python web server command, you should see an attempt to get the cat photo that failed, like so:

```
192.168.1.123 - - [31/Jan/2020 17:38:56] "GET /cat.jpg HTTP/1.1" 404 -
```

The 404 error code indicates that the resource can't be found, which makes sense given that you don't have a file named *cat.jpg* in this directory! Why did the web browser even ask for a cat photo? If you look back at the HTML for the page, you see an HTML ** tag that directs the browser to render the *cat.jpg* image. The browser requests the image, but it fails to retrieve it since the file is missing.

Let's fix the missing cat image issue. You need to download an image of a cat (or an image of anything really) in JPEG format and save it as *~/web/cat.jpg*. To make this easy, you can download the image used in the chapter with the following command. Be sure that your terminal window is currently in the *web* directory before running the command.

```
$ wget https://www.howcomputersreallywork.com/images/cat.jpg
```

You should now have *cat.jpg* stored in your *web* directory. Reload the page in a web browser to see the cat image in the page. Reminder: if the web server seems stuck, restart it as described in Project #37.

It's worth noting that not only can you view the cat image in your page, but you can also request the image directly from the server, since it has its own URL. Try pointing your browser to the following URL (replacing *SERVER* with the hostname or IP address you've been using for your website): *http://SERVER:8888/cat.jpg*. You should see the cat image rendered in the browser, outside of the web page. Every resource referenced on a web page has its own URL and can be accessed directly!

PROJECT #39: ADD CSS TO YOUR WEBSITE

Prerequisite: Project #38.

In this project, you'll use CSS to style your website. First, use the text editor of your choice to create a file named *style.css* in the *web* directory. This file will contain your CSS rules. Be sure the file is named *style.css* and is saved to the *web* directory alongside your *index.html* and *cat.jpg* files. The contents of *style.css* should be the following:

```
p {  
    font-family: Arial, Helvetica, sans-serif;  
    font-size: 11pt;  
    margin-left: 10px;  
    color: DimGray;  
}  
h1 {  
    font-family: 'Courier New', Courier, monospace;  
    font-size: 18pt;  
    font-weight: bold;  
}
```

Once *style.css* has been created, open *index.html* for editing, as you did in the previous project. Leave the existing HTML in place. We just want to add a single line to the head section, as shown here:

```
<head>  
    <meta charset="utf-8">  
    <title>A Cat</title>  
    <link rel="stylesheet" type="text/css" href="style.css">❶  
</head>
```

Once you've made this update ❶ to *index.html*, start your web server (if it isn't already running), and reload the page in your web browser. You should see the style of the page update. Reminder: if the web server seems stuck, restart it as described in Project #37.

Feel free to edit *style.css* to try different styles. Maybe you want to make the paragraph font huge or a different color! Edit the style to your liking, save *style.css*, and reload the page in your browser.

If you aren't seeing your updates reflected in the browser, it may be because your web browser is loading a cached copy of your website rather than downloading the latest version. Try opening the page in a new tab or restarting the browser altogether. You can also tell your browser to bypass its local cache when reloading. To do this, navigate to the page, and then press CTRL-F5 to force the page to reload. This works on most browsers on Windows and Linux. On a Mac, you can force a refresh in Chrome and Firefox with CMD-SHIFT-R. Sometimes multiple refreshes are needed before the browser renders the latest content.

PROJECT #40: ADD JAVASCRIPT TO YOUR WEBSITE

Prerequisite: Project #39.

In this project, you'll use JavaScript to make your website interactive. First, use the text editor of your choice to create a file named *cat.js* in the *web* directory. This file will contain JavaScript code. Be sure the file is named *cat.js* and is saved to the *web* directory alongside your *index.html* and *cat.jpg* files. The contents of *cat.js* should be the following:

```
document.addEventListener('DOMContentLoaded', function() {
    document.getElementById('cat-photo').onclick = function() {
        document.getElementById('cat-para').innerHTML += ' Meow!';
    };
});
```

Once *cat.js* has been saved, open *index.html* for editing, as you did in the previous project. Leave the existing HTML in place and make the changes shown here:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Cat</title>
    <link rel="stylesheet" type="text/css" href="style.css">
    <script src="cat.js"></script>❶
  </head>
  <body>
    <h1>Thoughts on a Cat</h1>
    <p id="cat-para"❷>This is a cat.</p>
    
  </body>
</html>
```

These changes reference the script ❶ and give IDs to the paragraph ❷ and image ❸.

Once you've made this update to *index.html*, start your web server (if it isn't already running), and reload the page in a web browser. You should now be able to click (or touch) the cat photo and see the word **Meow!** appended to the paragraph. Reminder: if the web server seems stuck, restart it as described in Project #37.

13

MODERN COMPUTING



This chapter provides an overview of a few select areas of modern computing. Given the diversity and breadth in computing, I had a wide range of topics to choose from. The areas I chose are by no means an exhaustive list of the interesting things happening in computing today. Instead, they represent a handful of topics that I believe are worth your consideration. In this chapter we cover apps, virtualization, cloud computing, Bitcoin, and more. We wrap up with a final project that brings together many of the topics covered in this book.

Apps

Since the early days of computing, people have referred to software programs that are used directly by users as *applications*. This term was shortened to *app* as a convenience, and in the past, the two terms were interchangeable. However, since Apple opened the iPhone *App Store* in 2008, the word *app* has taken on a distinct meaning. Although there is no standard technical definition for what makes a software program an app, apps tend to share a number of common characteristics.

Apps are designed for end users. Apps often target a mobile device, such as a smartphone or tablet. Apps are typically distributed through an internet-based digital storefront (an *app store*), such as Apple’s App Store, the Google Play Store, or the Microsoft Store. Apps have limited access to the system on which they run, and often must declare what specific capabilities they require to operate. Apps tend to use touchscreens as their primary means of user input. The term *app*, when used alone, usually implies software installed on a device that makes direct use of the operating system’s API. In other words, the term *app* usually means a *native app*, an app built for a particular operating system. In contrast, *web apps* are apps designed with web technology (HTML, CSS, and JavaScript), and are not tied to a particular OS. Figure 13-1 provides a high-level look at native apps and web apps.

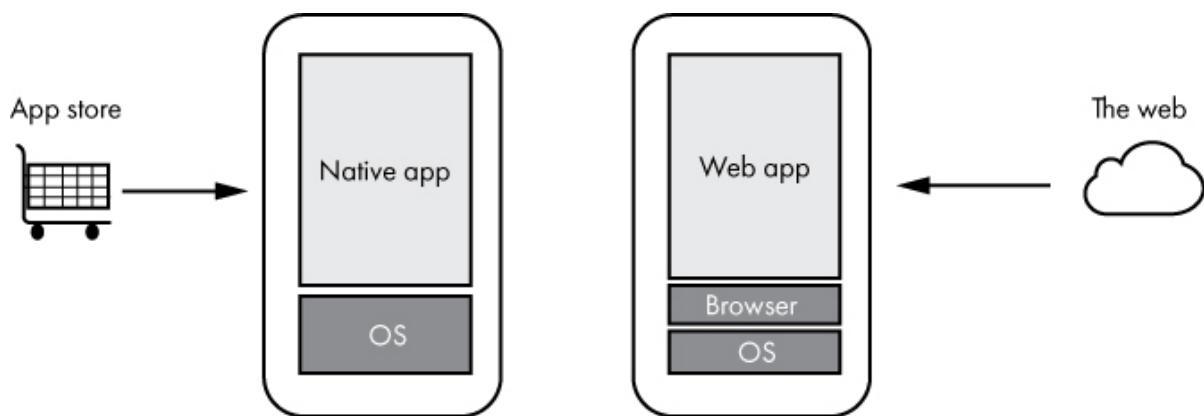


Figure 13-1: Native apps are built for a particular OS. Web apps are built using web technology.

As shown in Figure 13-1, native apps are typically installed from an app store and are designed to utilize the capabilities of a specific operating system. Web apps typically run from a website and are designed to use web technologies. Web apps run in a browser or another process that renders web content. Let’s now look at both native apps and web apps in more detail.

Native Apps

As mentioned earlier, native apps are built for a particular operating system. Apple's App Store and the similar app stores that followed it ushered in a new era of native software development, giving developers new platforms to target, new methods of distributing their software, and new ways to make money with software. The current state of native app development is largely focused on two platforms: iOS and Android. Software is certainly still developed for other operating systems, but often that software doesn't have the typical characteristics of an app (mobile-friendly, touch-input based, distributed via an app store, and so forth).

Android and iOS differ in their programming languages, APIs, and more. Therefore, writing an app that runs on both iOS and Android requires either maintaining separate codebases or the use of a *cross-platform framework* like Xamarin, React Native, Flutter, or Unity. These cross-platform solutions abstract the underlying details of each operating system API, making it possible for developers to write code that can be built to run on multiple platforms. Many native apps also rely on web services, meaning that app developers must not only write and maintain code for iOS and Android, but they must also build or integrate with web services.

Developing a cross-platform, web-connected app requires a good deal of work and expertise! In the past, developers would often focus on just one platform, such as Windows PCs or Macs. Things are certainly more complicated today for the developer targeting multiple platforms and the web. Platform competition is generally a good thing for users, but it does mean more work for developers.

Interestingly, the current state of app development could have turned out quite differently. When the iPhone was announced in January 2007, Steve Jobs (Apple's CEO at the time) had this to say about third-party app development on the iPhone:

The full Safari engine is inside of iPhone. And so, you can write amazing Web 2.0 and Ajax apps that look exactly and behave exactly like apps on the iPhone. And these apps can integrate

perfectly with iPhone services. They can make a call, they can send an email, they can look up a location on Google Maps. And guess what? There's no SDK that you need!

NOTE

An SDK (software development kit) is a collection of software used by developers to build applications for a particular platform.

Based on this quote, Apple's original plan for third-party app development was to simply let developers build app-like websites that could make use of the iPhone's capabilities. Native app development would be limited to the apps that Apple developed and included with the iPhone, such as the Camera, Mail, and Calendar apps.

At the time, using the web as a platform for application development wasn't common. Apple's position was forward-looking. Unfortunately, the underlying technologies of the web in 2007 were arguably not mature enough to position the web as a true app platform. By October 2007, Apple changed its message, announcing that Apple would allow developers to build native apps for the iPhone. Apple opened the App Store in 2008 as the only supported mechanism for distributing native iPhone apps to users.

Apple's policy reversal benefited the company, as the App Store became a source of revenue for Apple. There is a fee to register as an App Store developer, plus Apple takes a percentage of every sale. The App Store and native iPhone development also opened the door for exclusive content, apps that only worked on Apple devices.

The App Store also presents benefits to end users. A curated list of apps with ratings is helpful, and the store provides a measure of consumer trust. Apps that make it into the App Store must meet certain quality guidelines. A centralized payment service means users don't have to give their payment information to multiple companies. Apps are automatically updated, an advantage over traditional PC software, although not an advantage over the web, since web apps are also updated without user involvement.

With the success of Apple's App Store, other companies created similar digital storefronts for distributing software. The Google Play Store, Microsoft Store, and Amazon Appstore all operate on a similar model to Apple's store and provide similar benefits. Although this system has generally worked well for these companies and for end users, it has also created a complex environment for developers: multiple stores, multiple platforms, and varying technologies. Each digital marketplace has its own requirements that app developers must meet, and each store takes a percentage of sales revenue.

Web Apps

Alongside the rise of native apps, the web matured into a platform that's quite capable of running apps. A mature version of HTML known as HTML5 was introduced, and web browsers became more capable and consistent in their handling of content. Browser developers made their implementations of JavaScript compliant with the ECMAScript 5 standard, providing a better foundation for JavaScript code. Outside of browser updates, the web developer community embraced (and continues to embrace) a concept known as *responsive web design*, an approach that ensures web content renders well no matter the size of screen on which it's displayed. Using responsive design techniques, web developers can maintain a single website that works well across diverse devices, rather than creating separate websites that target different devices. Also, multiple *web development frameworks*, such as Angular and React, have been released in recent years. These frameworks make it easier for developers to write and maintain *web apps*—websites that behave like apps.

Developers have realized that modern web technology can be used to build experiences that closely resemble native apps, and many developers build websites that function as apps. Some developers have chosen to forgo native apps altogether and only build web apps. The advantages of this approach are that a web app will run on any device with a modern web browser, and the code only has to be written once. However, web apps also have some disadvantages. Web apps don't have

access to the full range of device capabilities, tend to be slower than native apps, require the user to be online, and generally aren't listed in app stores.

To address some of the disadvantages of web apps, *Progressive Web Apps (PWAs)* offer a set of technologies and guidelines that help bridge the gap between native apps and web apps. A PWA is just a website with a few extra features that help it be more app-like. A Progressive Web App must be served over HTTPS, render appropriately on mobile devices, be able to load while offline once downloaded, provide a manifest to the browser that describes the app, and transition quickly between pages. To the end user, running a PWA should feel as responsive and natural as running a native app. If a website meets the criteria for a PWA, modern web browsers give users the option of adding an icon for the PWA to their home screen or desktop. Doing so means that users can launch the web app just like they would launch a native app. The app opens in its own window, rather than in a browser window, and generally behaves like a native app.

PWAs can potentially offer great benefits to developers who wish to use web technologies for their apps but don't want to build multiple apps for different platforms. However, there are still some drawbacks to PWAs. A significant one is that PWAs don't appear in app stores. Mobile operating systems have been training users for years that apps should be obtained through app stores. Users aren't accustomed to browsing to a web page to get an app. At the time of this writing, only the Microsoft Store allows PWAs to be published directly to the store. Other platforms expect PWAs to be installed from the browser or repackaged as a native app that renders the web content. This repackaged app can then be submitted to the store. Another potential drawback is that PWAs may not look like native apps; they will usually look essentially the same on all platforms, although some might consider this a good thing. PWAs still don't have the performance of a native app or access to all the capabilities of the underlying platform, but depending on the needs of the app, this isn't necessarily an issue.

Virtualization and Emulation

When is a computer not a physical device? When it's a virtual computer, of course! *Virtualization* is the process of using software to create a virtual representation of a computer. A related technology, *emulation*, allows applications designed for a certain type of device to run on a totally different type of device. In this section we explore both virtualization and emulation.

Virtualization

A virtual computer, known as a *virtual machine (VM)*, runs an operating system just like a physical computer. In turn, applications run on that operating system. From the perspective of the application, the virtualized hardware acts like a physical computer. Virtualization enables several useful scenarios. A computer running one operating system can run another operating system in a virtual machine. For example, a computer running Windows can run an instance of Linux in a virtual machine. Virtual machines also allow datacenters to host multiple virtual servers on a single physical server. This provides a way for internet hosting companies to easily and quickly provide dedicated servers to their customers, as long as the customer is fine with a virtual server. VMs can be easily backed up, restored, and deployed.

A *hypervisor* is a software platform that runs virtual machines. There are two types of hypervisors, as illustrated in Figure 13-2.

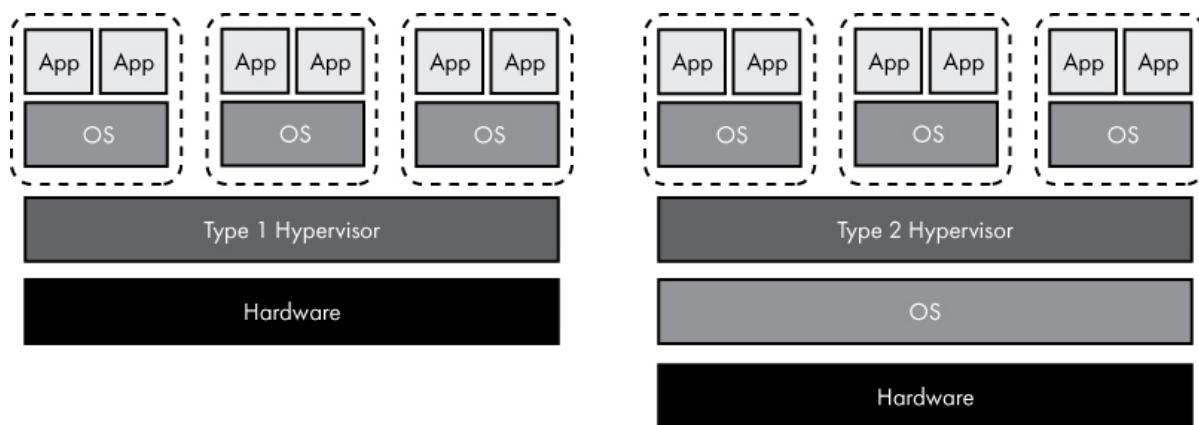


Figure 13-2: Type 1 and type 2 hypervisors

As shown on the left side of Figure 13-2, a hypervisor can interact directly with underlying hardware, actually placing the hypervisor below the kernel in the technology stack. The hypervisor talks to the physical hardware and presents virtualized hardware to the OS kernel. This is known as a *type 1 hypervisor*. In contrast, *type 2 hypervisors*, shown on the right in Figure 13-2, run as an application on an operating system. Microsoft’s Hyper-V and VMware ESX are type 1 hypervisors, whereas VMware Player and VirtualBox are examples of type 2 hypervisors.

Another popular approach for virtualization is the use of containers. A *container* provides an isolated user mode environment in which to run applications. Unlike a virtual machine, a container shares a kernel with the host OS and with other containers running on the same computer. A process running in a container can only see a subset of the resources available on the physical machine. For example, every container can be granted its own isolated filesystem. Containers provide the isolation of a VM without the overhead of running a separate kernel for each VM. In general, containers are limited to running the same operating system as the host since the kernel is shared. Some container technologies, like OpenVZ, are used to virtualize the entire user mode portion of operating systems, whereas others, like Docker, are used to run individual applications in isolated containers.

NOTE

You may recall that an operating system process was also described as a “container” in Chapter 10—this isn’t the same thing as a virtualization container.

Emulation

Emulation is the use of software to make one type of device behave like another type of device. Emulation and virtualization are similar in that they both provide a virtual environment for running software, but whereas virtualization offers up a slice of the underlying hardware, emulation presents virtual hardware that’s *unlike* the physical hardware

in use. For example, a virtual machine or container running on an x86 processor runs software compiled for x86, directly making use of the physical CPU. In contrast, an *emulator* (a program that performs emulation) running on x86 hardware can run software compiled for a completely different processor. Emulators often also provide other virtual hardware besides the processor.

For example, a complete emulator for the Sega Genesis (a video game system from the 1990s) will emulate a Motorola 68000 processor, a Yamaha YM2612 sound chip, input controllers, and every other piece of hardware found in a Sega Genesis. At runtime, such an emulator translates CPU instructions originally designed to run on a Sega Genesis to capabilities implemented in x86 code. This introduces significant overhead, since each CPU instruction must be translated, but a sufficiently fast modern computer can still emulate the much slower Sega Genesis at full speed. The result is the ability to run software intended for one platform on a completely different platform, as shown in Figure 13-3.

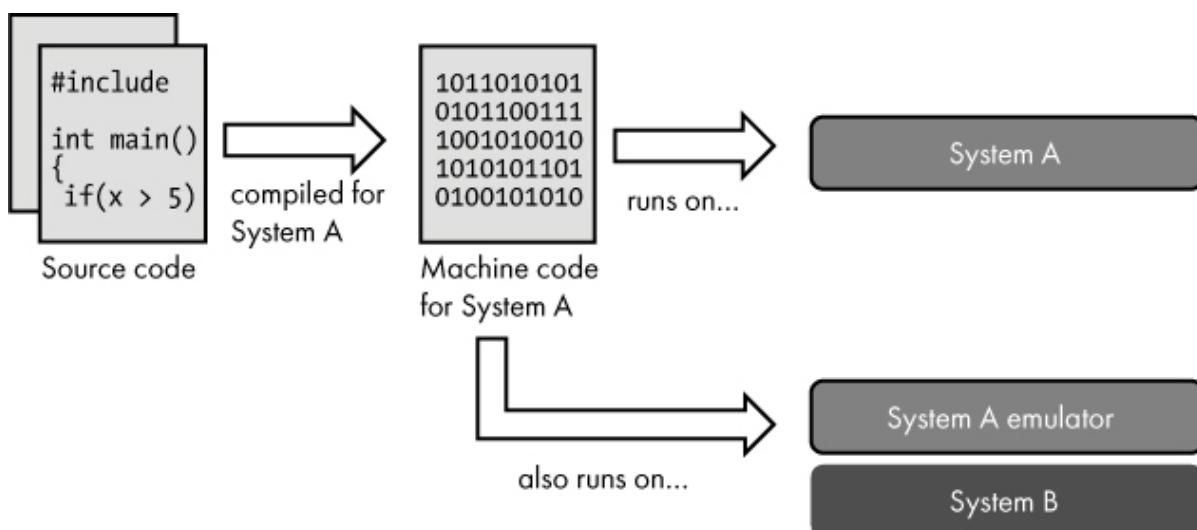


Figure 13-3: Code compiled for System A can run on an emulator for System A

Emulation plays an important role in preserving software designed for obsolete platforms. As computing platforms age, it becomes increasingly difficult to find working hardware. Emulation is commonly used by software developers who want to bring old software to a

modern platform. The original source code may be lost, or the task of modernizing it may be burdensome. In such cases, investing in an emulator allows the original compiled code to run on a new platform without modification.

PROCESS VIRTUAL MACHINES

There's another type of virtual machine that shares some traits with emulators. A *process virtual machine* runs an application within an execution environment that abstracts the details of the underlying operating system. It's similar to an emulator in that it provides a platform for execution that's decoupled from the hardware and OS on which it runs. However, unlike an emulator, a process VM isn't trying to simulate real hardware. Rather, it provides an environment designed for running platform-independent software. As we discussed in Chapter 9, Java and .NET make use of process virtual machines that run bytecode.

Cloud Computing

Cloud computing is the delivery of computing services over the internet. In this section, we'll look at various types of cloud computing, but first let's quickly review the history of remote computing.

The History of Remote Computing

Since the beginning of computing, we can observe a pendulum swing from remote, centralized computing (servers accessed from terminals), to local computing (desktop computers), and now back to remote computing (the web) accessed from smart, local devices (such as smartphones). Many applications today rely on a combination of remote computing and local computing. In the case of the web, some code runs in a browser and some code runs on a web server. The devices we carry in our pockets today are significantly more powerful than the room-sized computers from the early days of computing, yet much of what we want to do on those devices involves communicating with other

computers, so it makes sense that the responsibility of processing should be shared between local devices and remote servers.

With this reemergence of remote computing came a need for organizations to maintain servers. In the past, this meant purchasing a physical server, configuring it as needed, connecting it to the network, and letting it run in a closet somewhere. The organization had physical access to the machine and complete control over its configuration. However, maintaining a server (or a fleet of servers) can be a complex and costly endeavor. This includes the costs of purchasing and maintaining hardware, keeping up with software updates, dealing with security concerns and capacity planning concerns, managing the networking configuration, and so forth. Often the skillset and expertise required for this work doesn't align well with the purpose of an organization. Even a technology-focused company doesn't necessarily want to be in the business of maintaining servers. This is where cloud computing comes in.

Cloud computing delivers remote computing capabilities over the internet (the *cloud*). Underlying hardware is maintained by a cloud services company (the *cloud provider*), freeing the organization or user in need of such capabilities (the *cloud consumer*) from maintaining servers. Cloud computing allows for the purchase of computing services on demand, as needed. For the cloud consumer, this means releasing control of certain things and trusting a third party to deliver reliable service. Cloud computing takes many forms; let's look at some of those forms here.

The Categories of Cloud Computing

The various categories of cloud computing are typically defined by the line that divides the responsibility between the cloud provider and the cloud consumer. Figure 13-4 provides an overview of four categories of cloud computing (IaaS, PaaS, FaaS, and SaaS) and their respective divisions of responsibility. We'll look at each of these categories momentarily.

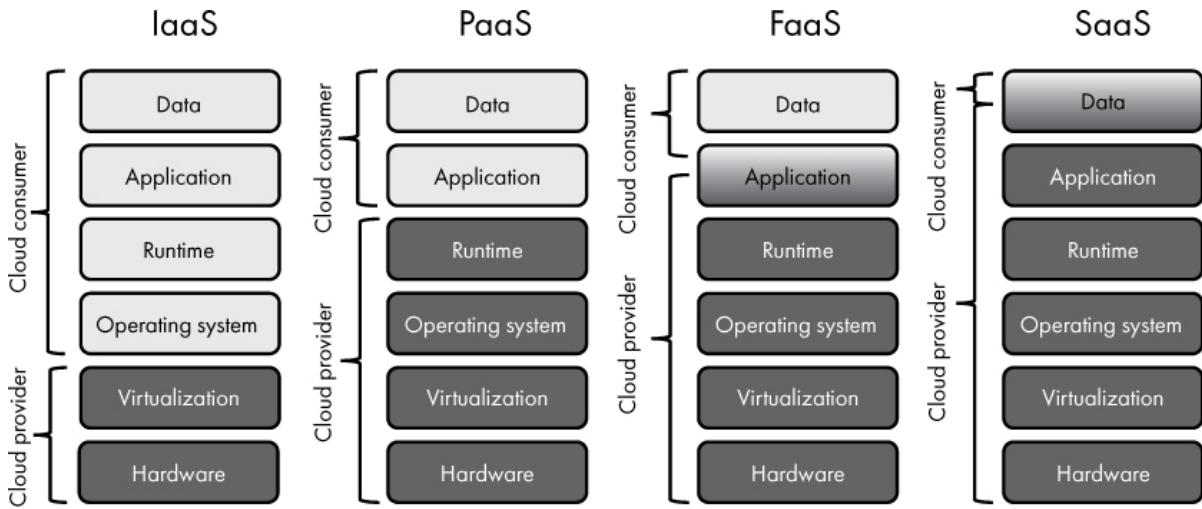


Figure 13-4: Distribution of responsibility in various types of cloud offerings

The vertical stacks in Figure 13-4 represent the components needed to run an application. No matter which category of cloud computing is used, all the components need to be present—the difference across categories lies in whether the cloud provider or the cloud consumer is responsible for managing each component. The various components in each stack should look familiar, since we've covered these topics already. However, *runtime* requires an explanation. A *runtime environment* is the environment in which an application executes, including any needed libraries, interpreters, process virtual machines, and so forth. Let's now cover the four categories of cloud computing shown in Figure 13-4, progressing from left to right.

Infrastructure as a Service (IaaS) is a cloud computing scenario in which a cloud provider manages hardware and virtualization only, allowing the consumer to manage the operating system, runtime environment, application code, and data. A consumer of IaaS typically gets an internet-connected, virtual computer to use as they see fit, typically as a server of some sort. This virtual computer is usually implemented as a hypervisor-based virtual machine or a container of the user mode portion of a Linux distribution. The consumer of an IaaS virtual server has access to the virtual computer's operating system and is able to configure it however they wish. This gives the consumer maximum flexibility, but it also means that the responsibility of

maintaining the system's software (including the operating system, third-party software, and so forth) sits squarely on the shoulders of the consumer. IaaS provides a virtual computer, and the consumer is responsible for everything that runs on that computer. Here are some examples of IaaS: Amazon Elastic Compute Cloud (EC2), Microsoft Azure Virtual Machines, and Google Compute Engine.

Platform as a Service (PaaS) gives the cloud provider more responsibility. In a PaaS scenario, the cloud provider manages not only hardware and virtualization, but also the operating system and runtime environment that the consumer wishes to use. A PaaS consumer develops an application that's targeted to run on their chosen cloud platform, taking advantage of the various capabilities that are unique to that platform. Cloud consumers of PaaS offerings don't need to concern themselves with maintaining the underlying OS or runtime environment. The cloud consumer can just focus on their application code. Although the provider does abstract away the details of the underlying system, the consumer still needs to manage what resources are provisioned by the provider to handle their application. This includes the amount of storage required and the type of allocated virtual machines. PaaS provides a managed platform for running code, and the consumer is responsible for the application that runs on that platform. Here are some examples of PaaS: Amazon Web Services Elastic Beanstalk, Microsoft Azure App Service, and Google App Engine.

Function as a Service (FaaS) takes the PaaS model one step further. It does not require the consumer to deploy a full application or to provision platform instances ahead of time. Instead, a consumer only needs to deploy their code (a function) that runs in response to certain events. For example, a developer could write a function that returns the distance to the nearest grocery store. This function could run in response to a web browser sending its current GPS coordinates to a URL. This event-driven model means that the cloud provider is responsible for an on-demand invocation of the consumer's code. The consumer no longer needs to have application code running all the time, waiting for requests. This can simplify things for the consumer.

and reduce costs, although it can mean a slower response time when a request comes in if the function code isn't already running.

FaaS is a type of *serverless computing*, a cloud computing model where consumers do not have to deal with managing servers or virtual machines. Of course, the term is a misnomer; servers are actually required to run the code, it's just that the consumer doesn't have to think about them! FaaS provides an event-driven platform for running code, and the consumer is responsible for the code that runs in response to events. Some examples of FaaS include Amazon Web Services Lambda, Microsoft Azure Functions, and Google Cloud Functions.

Software as a Service (SaaS) is a fundamentally different type of cloud service. SaaS delivers an application to the consumer that's fully managed in the cloud. Whereas IaaS, PaaS, and FaaS are for software engineering teams who want to run their own code in the cloud, SaaS delivers a complete cloud application to end users or organizations, already written. So much software runs in the cloud today that this may seem unremarkable, but it stands in contrast to a user or organization installing and maintaining software on their local devices and network. SaaS provides a complete application managed in the cloud, and the consumer is only responsible for the data they store in that application. Even management of data is partially handled by the provider, including the details of how the data is stored, backed up, and so forth. Some examples of SaaS include Microsoft 365, Google G Suite, and Dropbox.

Some of the major players in the cloud provider space are Amazon Web Services, Microsoft Azure, Google Cloud Platform, IBM Cloud, Oracle Cloud, and Alibaba Cloud.

The Deep Web and Dark Web

You have probably read news about nefarious happenings on the dark web or the deep web. Unfortunately, the two terms are often confused, but they have distinct meanings. The web can be divided into three

broad segments: the surface web, the deep web, and the dark web, as illustrated in Figure 13-5.

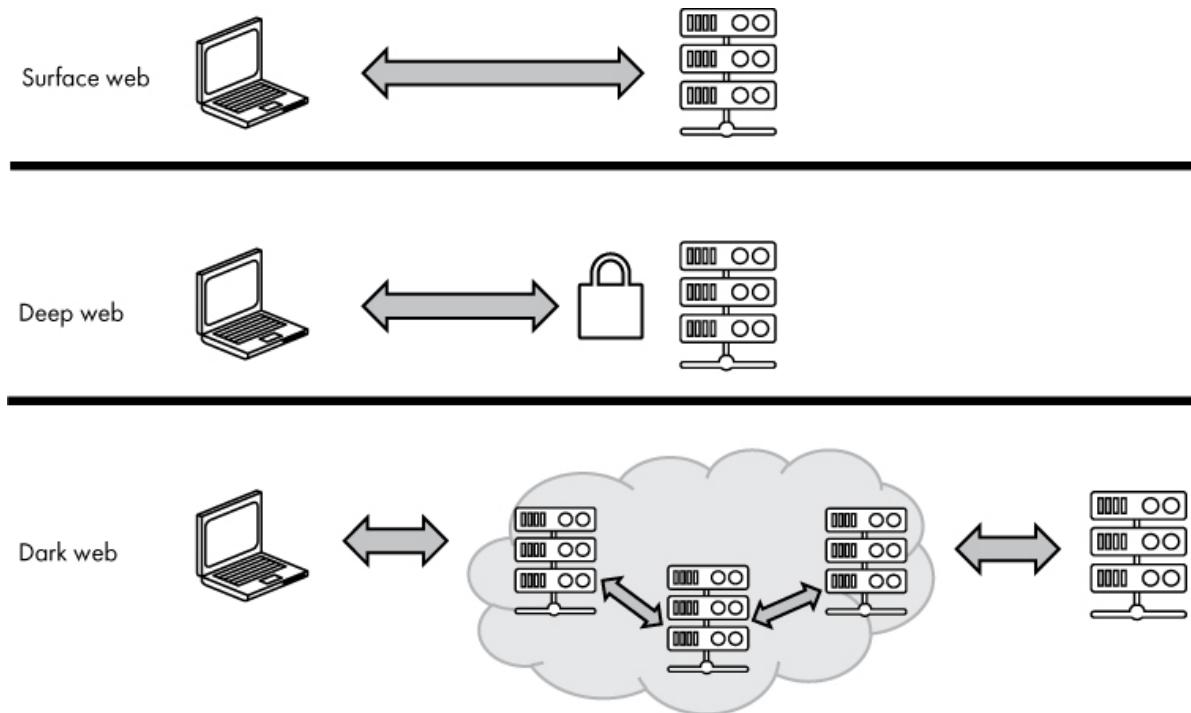


Figure 13-5: The surface web, deep web, and dark web

Content that's freely available for anyone to access is part of the *surface web*. Public blogs, news sites, and public Twitter posts are all examples of surface web content. The surface web is indexed by search engines, and sometimes the surface web is defined as content that can be found with a search engine.

The *deep web* is web content that cannot be accessed without logging in to a website or web service. Most internet users access deep web content on a regular basis. Checking your bank balance, reading your email through a website like Gmail, logging in to Facebook, looking at your personal shopping history on Amazon—these are all examples of deep web activities. The deep web is simply content that's not publicly available and generally requires a password of some sort to access. Most users don't want their email or bank balances to be publicly available, so there's a good reason why this type of content isn't public and cannot be indexed by search engines.

The *dark web* is web content that requires specialized software to access. You cannot access the dark web using only a standard web browser. The most prevalent dark web technology is *Tor (the onion router)*. Through a system of encryption and relays, Tor allows for anonymous access to the web, preventing a user's ISP from monitoring which sites are accessed, and preventing sites from knowing their visitor's IP address. Additionally, Tor allows users to access websites known as *onion services* that cannot be accessed at all without Tor—these sites are part of the dark web. Tor hides the IP addresses of onion services, making them anonymous as well. As you might expect, the anonymity of the dark web is sometimes exploited for criminal purposes. However, there are legitimate uses for the privacy afforded by the dark web, such as whistleblowing and political discussion. I recommend caution when accessing content on the dark web.

Bitcoin

A *cryptocurrency* is a digital asset intended to be used for financial transactions, as a substitute for a traditional currency like the US dollar. Users of cryptocurrencies maintain a balance of that currency, much like at a traditional bank, and can spend their currency on goods and services. Some users treat cryptocurrencies primarily as an investment rather than a means of commerce, making it more akin to something like gold for those users. Unlike traditional currencies, cryptocurrencies are typically decentralized, with no single organization controlling their use.

Bitcoin Basics

Introduced in 2009, *Bitcoin* was the first decentralized cryptocurrency, and it's the most well-known today. Since then, a large number of alternate cryptocurrencies (known as *altcoins*) have sprung up, but none have challenged the dominance of Bitcoin. Bitcoin's main unit of currency is also simply called *bitcoin*, abbreviated *BTC*.

Bitcoin and similar cryptocurrencies are based on *blockchain* technology. In a blockchain, information is grouped into data structures called *blocks*, and blocks are linked together chronologically. That is, when a new block is created, it's added to the end of a blockchain. In the case of Bitcoin, blocks hold transaction records, tracking the movement of bitcoins. Figure 13-6 illustrates the Bitcoin blockchain.

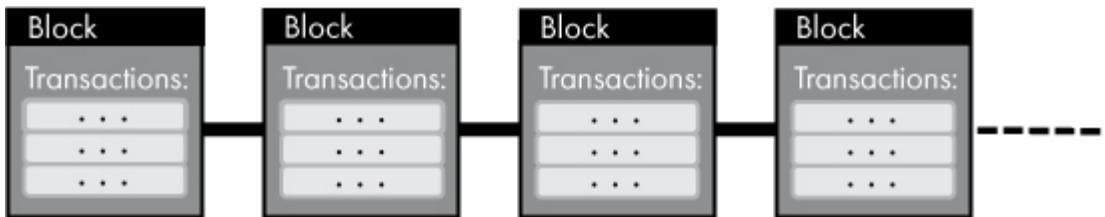


Figure 13-6: Bitcoin's blockchain links chronological blocks of transaction records.

Blockchains operate over a network such as the internet, with multiple computers working together to process transactions and update the blockchain. The computers that work together to process Bitcoin transactions are known as the *Bitcoin network*. A computer that connects to the Bitcoin network is called a *node*, and certain nodes hold a copy of the blockchain; there is no single master copy. Encryption and decryption are employed to ensure the integrity of transactions and prevent tampering with the data in the blockchain. Once written, blockchain data is immutable—it can't be changed. Bitcoin's blockchain is a public, decentralized, immutable ledger of transactions. This ledger is used to record all events that occur on the Bitcoin network, such as the transfer of bitcoins.

Bitcoin Wallets

An end user's bitcoins are stored in a *Bitcoin wallet*. However, more accurately, a Bitcoin wallet holds a collection of cryptographic key pairs, as illustrated in Figure 13-7.

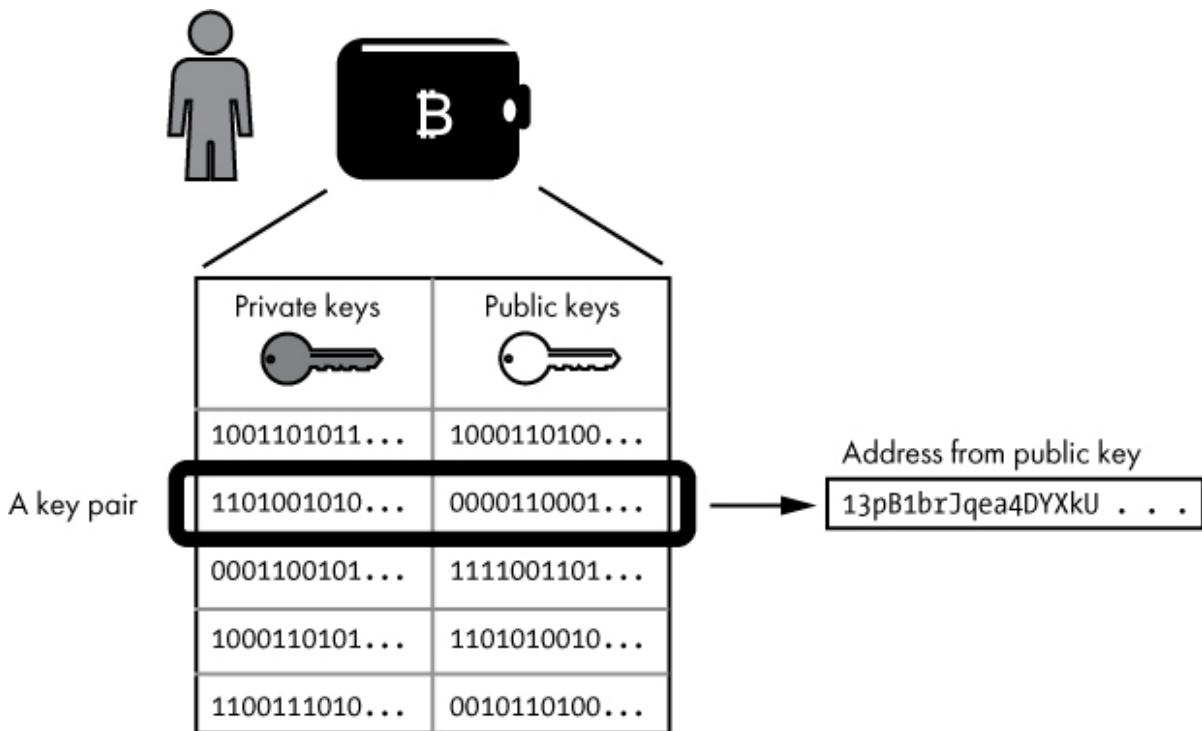


Figure 13-7: A Bitcoin wallet contains key pairs. A Bitcoin address is derived from a public key.

As shown in Figure 13-7, each key pair in a wallet consists of two numbers—a private key and a public key. The *private key* is a randomly generated 256-bit number. This number must be kept secret; anyone who knows the private key can spend the bitcoins associated with the key pair. The *public key*, which is used to receive bitcoins, is derived from the private key. When receiving bitcoins, the public key is represented as a *Bitcoin address*, a text string generated from the public key. Here's an example Bitcoin address: 13pB1brJqea4DYXkU5n44HCgBkJHa2v1.

Let's say I have one bitcoin that I want to send you. This bitcoin is associated with an address that I control. That is, I have the private key for this address. If you give me the text string representation of a Bitcoin address that you control, I can send my bitcoin to your address. You don't need to (and shouldn't) send me your private key. I'm able to send my bitcoin to you because I have the private key for my address, which allows me to spend my bitcoin. Conversely, I can't transfer any bitcoins out of your address because I don't have your private key.

Bitcoin Transactions

Let's take a closer look at how this works. A transfer of bitcoins is known as a *transaction*. To send bitcoins, wallet software constructs a transaction specifying the details of the transfer, digitally signs it with a private key, and broadcasts the transaction to the Bitcoin network. The computers in the Bitcoin network verify the transaction and add it to a new block on the blockchain. Figure 13-8 illustrates a Bitcoin transaction.

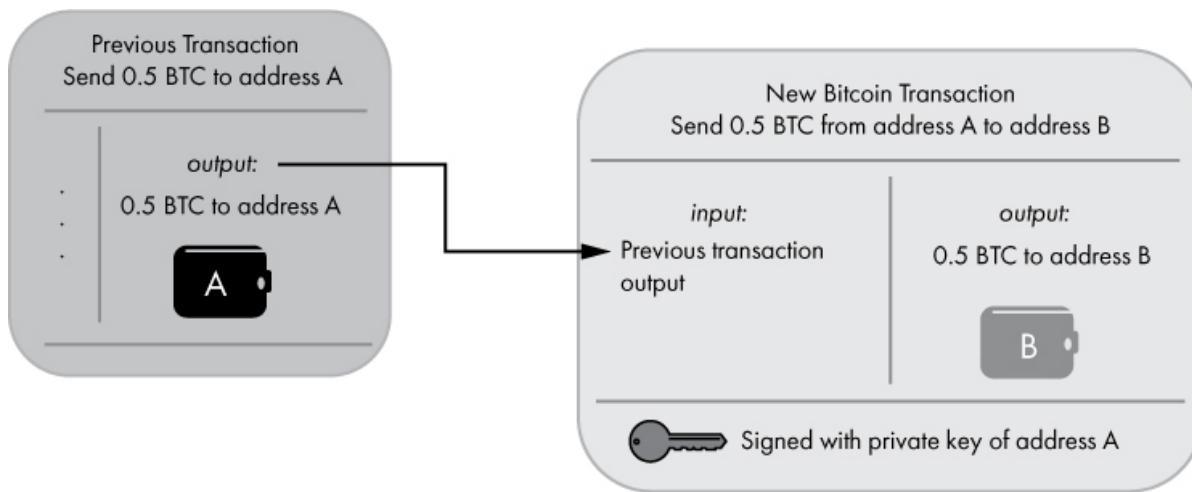


Figure 13-8: A Bitcoin transaction moves 0.5 bitcoin to address B (ignoring any transaction fee).

As shown in Figure 13-8, a transaction contains inputs and outputs, representing where the bitcoin is coming from and going to. On the left of this figure we have a previous transaction, where only the output is shown; the input of the previous transaction isn't relevant to our discussion. In the previous transaction, 0.5 BTC was sent to address A.

On the right side of Figure 13-8 we have a new transaction, which moves 0.5 BTC from address A to address B. For simplicity, this transaction has only a single input and single output. The input represents the source of the bitcoin to be transferred. You might expect this to be a Bitcoin address, but it isn't. Instead, the input is the previous transaction's output. Let's say that address A is my address, and I want to send 0.5 bitcoin to your address, address B. Now, I know that previously

0.5 BTC was sent to my address, so I can use that previous transaction's output as an input to a new transaction, allowing me to send that 0.5 bitcoin to you. The output portion of a transaction contains the address where the bitcoin is sent.

Although you can think of an address as having a balance of bitcoin, the amount of bitcoin associated with an address isn't stored in a Bitcoin wallet, nor is the balance directly stored in the blockchain. Instead, the history of transactions association with that address is stored in the blockchain, and from that history the balance for a certain address may be calculated. As a reminder, Bitcoin wallets simply contain the keys that enable Bitcoin transactions.

Bitcoin Mining

The process of maintaining the Bitcoin blockchain is known as *Bitcoin mining*. Computers from around the globe add blocks of transactions to the blockchain—these computers are called *miners*. Figure 13-9 illustrates the process of mining bitcoins.

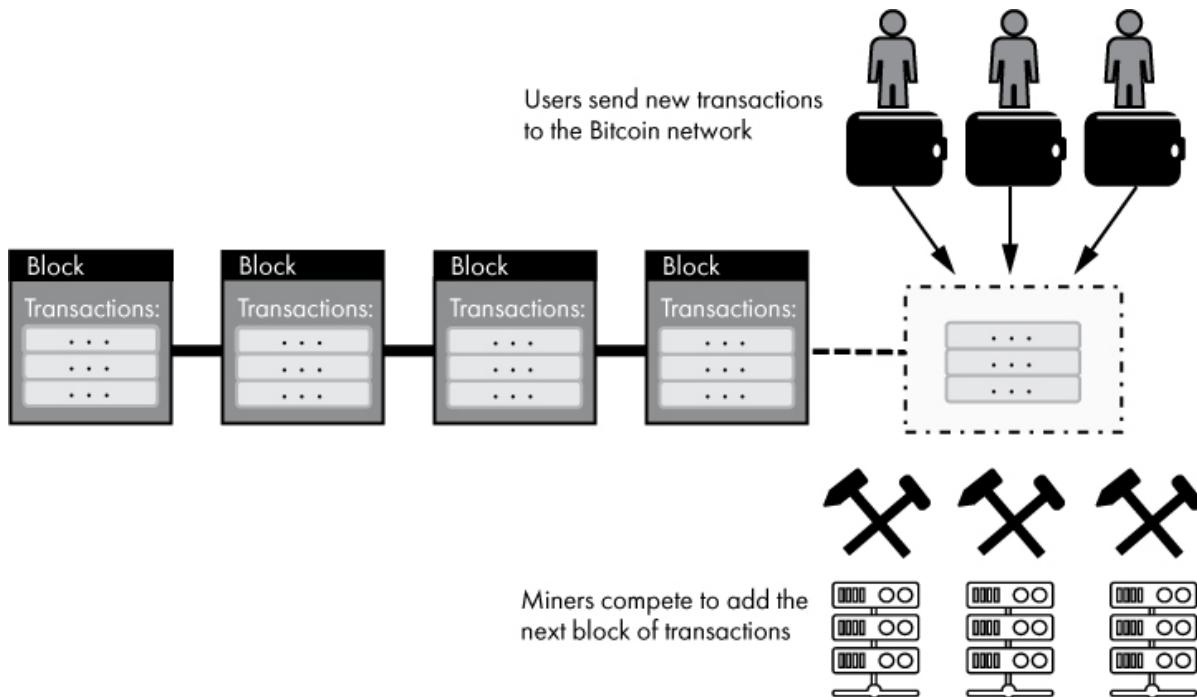


Figure 13-9: Bitcoin mining

In order to add a block of transactions to the blockchain, a miner must verify the transactions included in the block (ensuring that each transaction is syntactically correct, that the input coins haven't already been spent, and so forth), and it must also complete a computationally difficult problem. Requiring miners to solve such a problem prevents tampering with the blockchain, since altering a block would require solving the problem for the altered block and for every block that comes after it in the blockchain. This system of solving a difficult problem as a means of deterring unwanted behavior is known as *proof of work*.

Arriving at a solution to the computational problem involves a significant number of trial-and-error calculations. The solution is hard to produce but easy to verify. The first miner to complete the problem is awarded a sum of bitcoins. This is how new bitcoins are generated and introduced into the system. In this way, Bitcoin mining is similar to traditional mining—miners perform work and may “strike gold” under the right circumstances. In addition to being awarded newly minted bitcoins, the miner is also able to claim a *fee* for each transaction included in the block, which is deducted from the total amount of bitcoins sent in the transaction. Bitcoin is designed to only allow for 21 million coins to be mined in total. Once this number is reached, Bitcoin miners will no longer be awarded bitcoins, and will instead rely on transaction fees to fund their operations.

BITCOIN BEGINNINGS

The Bitcoin blockchain began when the first block, known as the *genesis block*, was mined in 2009. This block was mined by Satoshi Nakamoto, who is credited with inventing Bitcoin. “Satoshi Nakamoto” is presumed to be a pseudonym; this person’s identity is disputed at the time of this writing.

For Bitcoin mining to be profitable, the costs of operating mining hardware must not exceed the value of bitcoins awarded. Bitcoin mining hardware tends to be power-hungry, and so the electricity bill for people

mining Bitcoin can be high. Bitcoin was originally mined on regular computers, but today specialized, costly hardware is used to mine as quickly as possible (remember, the award goes to the first computer to solve the problem). These costs, plus the highly volatile price of bitcoins, mean that Bitcoin mining is not a guaranteed path to profit!

The Bitcoin blockchain is public—all transactions can be viewed by anyone. However, the blockchain contains no records of the personal identity of the people transferring bitcoins. So while an address's balance and transaction history are public, there's no easy way to tie that address to a human. For that reason, Bitcoin is attractive to those who wish to remain anonymous, such as those who run commerce sites on the dark web.

Blockchain technology is closely associated with cryptocurrencies, where it's used as a financial ledger, but blockchains can be used for other purposes as well. Any system that needs a tamper-resistant history of records could make use of a blockchain. Time will tell if Bitcoin or other cryptocurrencies are successful in the long run, but regardless, we may see blockchain technology leveraged in other novel ways.

Virtual Reality and Augmented Reality

Two technologies that have the potential to fundamentally change how we interact with computers are virtual reality (VR) and augmented reality (AR). *Virtual reality* is a form of computing that immerses a user in a three-dimensional virtual space, typically displayed by a headset. VR allows the user to interface with virtual objects via a variety of input methods, including the user's gaze, voice commands, and specialized handheld controllers. In contrast, *augmented reality* overlays virtual elements onto the real world, either through a headset or by the user looking "through" a handheld portable device, like a smartphone or tablet. VR immerses the user in another world; AR alters the real world.

Although various attempts at VR have been made for several decades, it wasn't until the 2010s that VR became more mainstream.

Google helped popularize VR in 2014 with *Google Cardboard*, named for the idea that a VR headset can be constructed from cardboard, lenses, and a smartphone. Specially-designed Cardboard apps present VR content to the user by rendering content for the left eye on half of the smartphone screen and content for the right eye on the other half of the screen, shown in Figure 13-10.

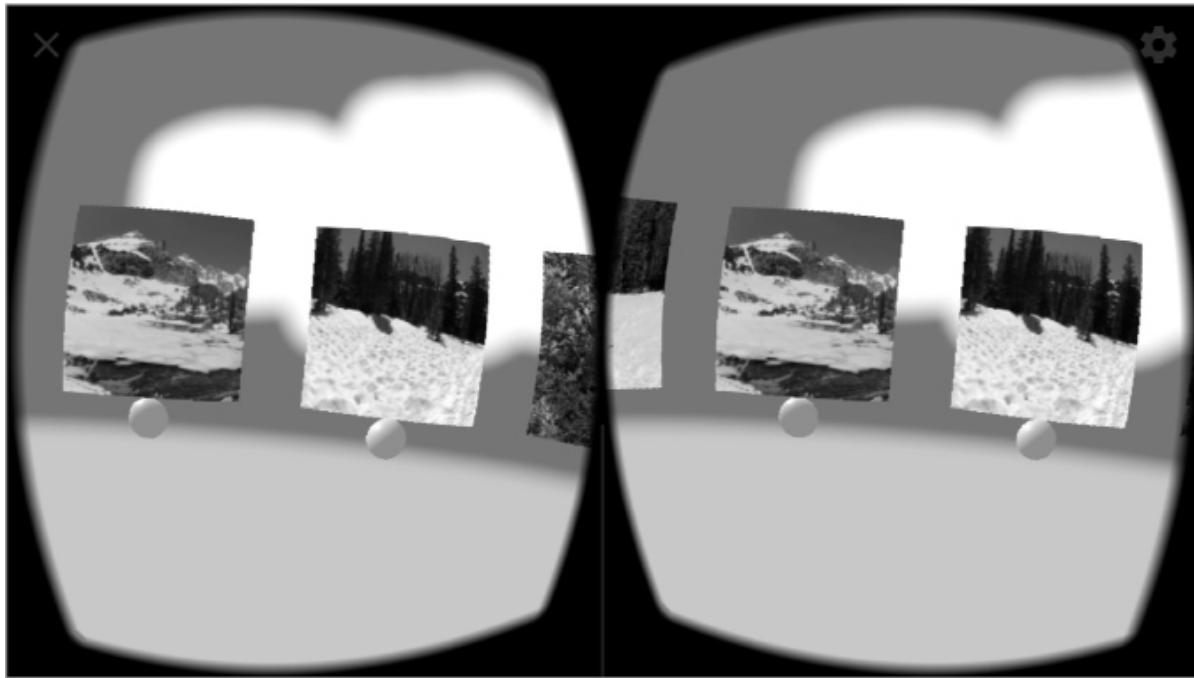


Figure 13-10: An app designed for Google Cardboard presenting in VR mode

Apps designed for Cardboard rely on the smartphone's ability to detect gyroscopic movement, allowing the display to update as the user moves their head. Such a headset is said to have *3 degrees of freedom* (*3DoF*); the headset can track limited head movement, but it cannot otherwise track movement in space. This allows a user to *look* around, but not *move* around using the headset. Cardboard also supports a basic one-button input. Cardboard is simple, but effective. It introduced VR to many users who probably wouldn't have tried it otherwise.

A more immersive experience requires *6 degrees of freedom* (*6DoF*); where the user can move around in VR by physically moving their body in real space. Some VR headsets support 6DoF, and VR controllers can

have either 3DoF or 6DoF. A 6DoF controller, held in the user's hand, can track the position of the controller in VR space, allowing for more natural interactions with the VR environment.

The consumer market has seen a number of VR solutions released since Google Cardboard. Some rely on smartphones (Samsung Gear VR, Google Daydream). Others use a personal computer for processing, with a connected VR headset and controllers (Oculus Rift, HTC Vive, Windows Mixed Reality). Still others are standalone devices, not requiring a smartphone or PC (Oculus Go, Oculus Quest, Lenovo Mirage Solo). In general, the PC-connected solutions provide the highest graphical fidelity, and are also the most expensive, particularly when considering the cost of the required computer.

As mentioned earlier, augmented reality, or AR, is a similar but distinct technology. While VR attempts to completely immerse the user in a virtual world, AR overlays virtual elements onto the real world. This can be accomplished using a mobile device, where a rear-facing camera is used to observe the real world while simulated elements are overlaid on what is seen by the camera. Advanced AR techniques allow software to understand the physical elements in a room so that overlaid virtual elements can interact seamlessly with the environment. AR is implemented in basic form in mobile apps, but it's more fully realized in dedicated devices such as Google Glass, Magic Leap's headset, and Microsoft HoloLens. Such AR devices are worn on the head and superimpose computer generated graphics on a user's field of view. Users are able to interact with virtual elements using various methods, such as voice commands or hand tracking.

The various VR and AR technologies (referred to together as *XR*) present multiple platforms for software developers to target. Many VR developers rely on existing game engines that are typically used for building 3D games, such as the Unity game engine or the Unreal game engine. These engines are familiar to game developers already, and they make it relatively easy for the developers to build their software for multiple VR platforms. Web developers can develop VR and AR content using JavaScript APIs known as *WebVR* and *WebXR*. Of the two,

WebVR came first and was focused on VR specifically. WebXR followed, with support for both AR and VR.

The Internet of Things

Traditionally we think of servers as providing services on the internet, and users interacting with those servers via internet-connected personal computing devices, such as PCs, laptops, and smartphones. In recent years we've seen the growth of new types of devices connecting to the internet—speakers, televisions, thermostats, doorbells, cars, lightbulbs, you name it! This concept of connecting all kinds of devices to the internet is known as the *Internet of Things (IoT)*.

Costs and physical size of electronic components are decreasing, Wi-Fi and cellular internet access are widespread, and consumers expect their devices to be “smarter.” All of this has contributed to the trend of connecting everything to the internet. IoT devices typically don’t operate without some kind of web service supporting them, so the rise of cloud computing has also furthered the spread of the Internet of Things. For consumers, IoT devices are prominent in the “smart home,” where all types of home appliances can be monitored and controlled. In business, IoT devices can be found in manufacturing, healthcare, transportation, and more.

Although these types of connected devices bring clear benefits, these devices also introduce risks. Security of such devices is a particular area of concern. Not every IoT device is well secured against attacks from malicious parties. Even if the data on the device isn’t of interest to an attacker, the device can act as a foothold in an otherwise well-defended network, or it can be used as a launch point for a remote attack against a different target. Particularly for consumers, an IoT device seems innocuous enough, and security concerns often aren’t top of mind when connecting such a device to a home network.

Privacy is another risk presented by IoT devices. Many of these devices, by their nature, collect data. That data is often sent to a cloud

service for processing. How much should end users trust the organizations that operate these services with their personal data? Even a well-intentioned organization can be a victim of a data breach, and user data may be exposed in unexpected ways. Devices like smart speakers must be listening all the time, waiting for verbal commands. This presents a risk of accidental recording of private conversations. Modern day readers of George Orwell's novel *1984* may find a certain irony in seeing consumers of today willingly trading privacy for convenience.

Another risk with IoT devices is that their full functionality often depends on a cloud service. If a device's internet connection goes down, that device may temporarily become less useful. A greater concern is that a device's manufacturer will likely someday permanently turn off the service that supports the device. At that point, the smart device will revert to a dumb device!

NOTE

Please see Project #41 on page 311, where you can use what you've learned about hardware, software, and the web to build a network-connected "vending machine" IoT device.

Summary

In this chapter we covered a variety of topics related to modern computing. You learned about apps, both native and web-based. You explored how virtualization and emulation allow computers to run software on virtualized hardware. You saw how cloud computing provides new platforms for running software. You learned how the surface web, deep web, and dark web differ, and how cryptocurrencies like Bitcoin enable decentralized payment systems. We touched on virtual reality and augmented reality and how they enable unique user interfaces for computing. You learned about IoT, and had an opportunity to build an internet-connected "vending machine."

As we near the end of this book, let's review some major computing concepts and see how they fit together. Computers are binary digital

devices, where everything is represented as a 0 or 1, on or off. Binary logic, also known as Boolean logic, provides the foundation for computing operations. Computers are implemented using digital electrical circuits where voltage levels represent binary states—a low voltage is 0, and a high voltage is 1. Digital logic gates are transistor-based circuits that enable Boolean operations such as AND and OR. Such logic gates can be arranged to create more complex circuits, such as counters, memory devices, and addition circuits. These types of circuits provide a conceptual foundation for computer hardware: a central processing unit (CPU) that executes instructions, random access memory (RAM) that stores instructions and data while powered, and input/output (I/O) devices that interact with the outside world.

Computers are programmable; they can perform new tasks without changing their hardware. Instructions that tell a computer what to do are known as software or code. CPUs execute machine code, whereas software developers typically write source code in a higher-level programming language. Computer programs typically run on an operating system—software that communicates with computer hardware and provides an environment for the execution of programs. Computers communicate using the internet, a globally connected set of computer networks that all use the TCP/IP protocol suite. A popular use of the internet is the World Wide Web, a set of distributed, addressable, linked resources, delivered by HTTP over the internet. All of these technologies provide an environment where modern computing innovations can flourish.

I hope this book has given you a fuller understanding of how computers work. We covered a great deal of ground, and yet we only scratched the surface of most topics. If some particular area grabbed your attention, I'd encourage you to continue learning about that subject—read about it online, take a class, watch videos, or buy another book! There's a wealth of knowledge about computing to be discovered.

PROJECT #41: USE PYTHON TO CONTROL A VENDING MACHINE CIRCUIT

Prerequisites: Project #7 (on page 105) and #8 (on page 107) where you built a vending machine circuit. A Raspberry Pi, running Raspberry Pi OS. I recommend that you flip to Appendix B and read the entire “Raspberry Pi” section on page 341 if you haven’t already.

In this project, you’ll use what you’ve learned about hardware, software, and the web to build a network-connected “vending machine” IoT device. Back in Chapter 6 you built a vending machine circuit using push buttons, an LED, and digital logic gates. For this project, you’ll update that device. You’ll keep the buttons and LED, but you’ll replace the logic gates with Python code running on a Raspberry Pi. This will allow you to add capabilities in software easily, such as the ability to connect to the device over the network.

For this project, you’ll need the following components:

- Breadboard
- LED
- Current-limiting resistor to use with your LED; approximately 220Ω
- Two switches or pushbuttons that fit a breadboard
- Jumper wires, including 4 male-to-female wires
- Raspberry Pi

GPIO

Besides its tiny size and low cost, the Raspberry Pi has another feature that sets it apart from a typical computer—its GPIO pins. Each *general-purpose input/output* (GPIO) pin can be designated as an electrical input or output. When a pin acts as an input, code running on the Raspberry Pi can read the pin as high at 3.3V, or as low at 0V. The Raspberry Pi even has internal pull-up and pull-down resistors, enabled through software, so you no longer have to add such resistors to your input buttons. When a pin acts as an output, it can be set to high (3.3V) or low (0V), all controlled through software. Some pins are always set to ground, 5V, or 3.3V. The pins are referenced in software by number. Figure 13-11 shows the GPIO pin designations.

As you can see in Figure 13-11, the GPIO numbers don’t correspond to the pin numbers. The pins, shown in the gray box, are simply numbered 1 through 40, starting in the upper left and ending in the lower right. For example, the second pin down on the left side is GPIO 2 and pin number 3. When you reference these GPIO pins in code, you need to use the GPIO number rather than the pin number.

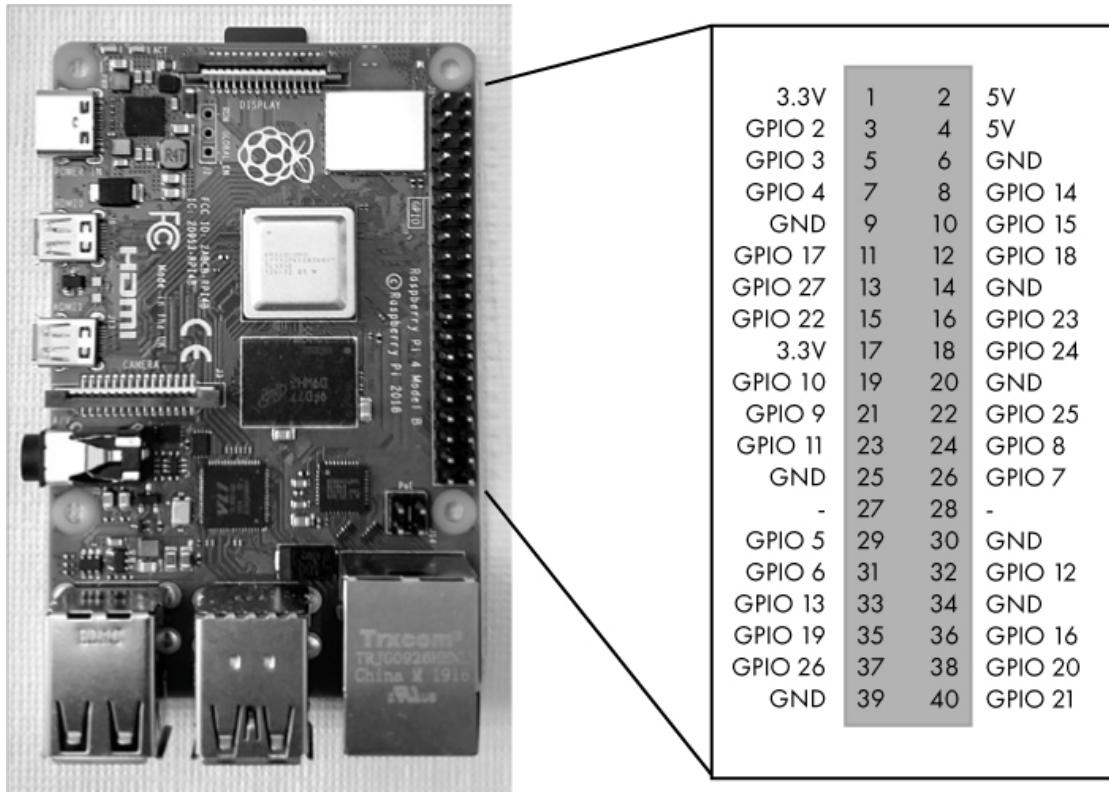


Figure 13-11: Raspberry Pi GPIO pins

BUILD THE CIRCUIT

Before writing any code, connect your circuit components to a breadboard and to the Raspberry Pi GPIO pins as shown in Figure 13-12.

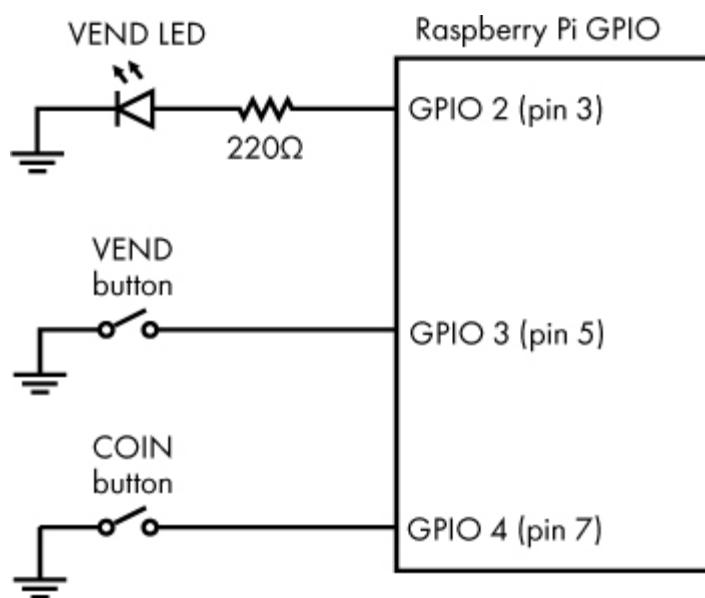


Figure 13-12: Raspberry Pi vending machine circuit diagram

I'd recommend powering off your Raspberry Pi before connecting anything to the GPIO pins. For the connections between the GPIO pins and the breadboard, use a male-to-female jumper wire. You can connect the female end of the wire to the GPIO pin, and the male end of the wire to the breadboard.

If you use the pin numbers shown in Figure 13-12, the VEND LED, VEND button, and COIN button are connected to three consecutive GPIO pins on the Raspberry Pi. You also need to connect one of the GND pins (I'd recommend pin 9) to the breadboard's negative power column, so you can easily connect the buttons and LED to ground.

You may have noticed that this circuit's input switches are wired differently from the switches you used in Project #7 (on page 105) and #8 (on page 107). In those projects, you connected a switch to 5V on one side and to a pull-down resistor/input pin on the other side. The circuit was designed so that an open switch was expected to be a low voltage, whereas a closed switch was expected to be a high voltage. Here, things are just the opposite—a closed switch is low, and an open switch is high. Internally, the Raspberry Pi pulls the GPIO pin high when the switch is open (or nothing is connected).

Figure 13-13 shows this circuit built on a breadboard.

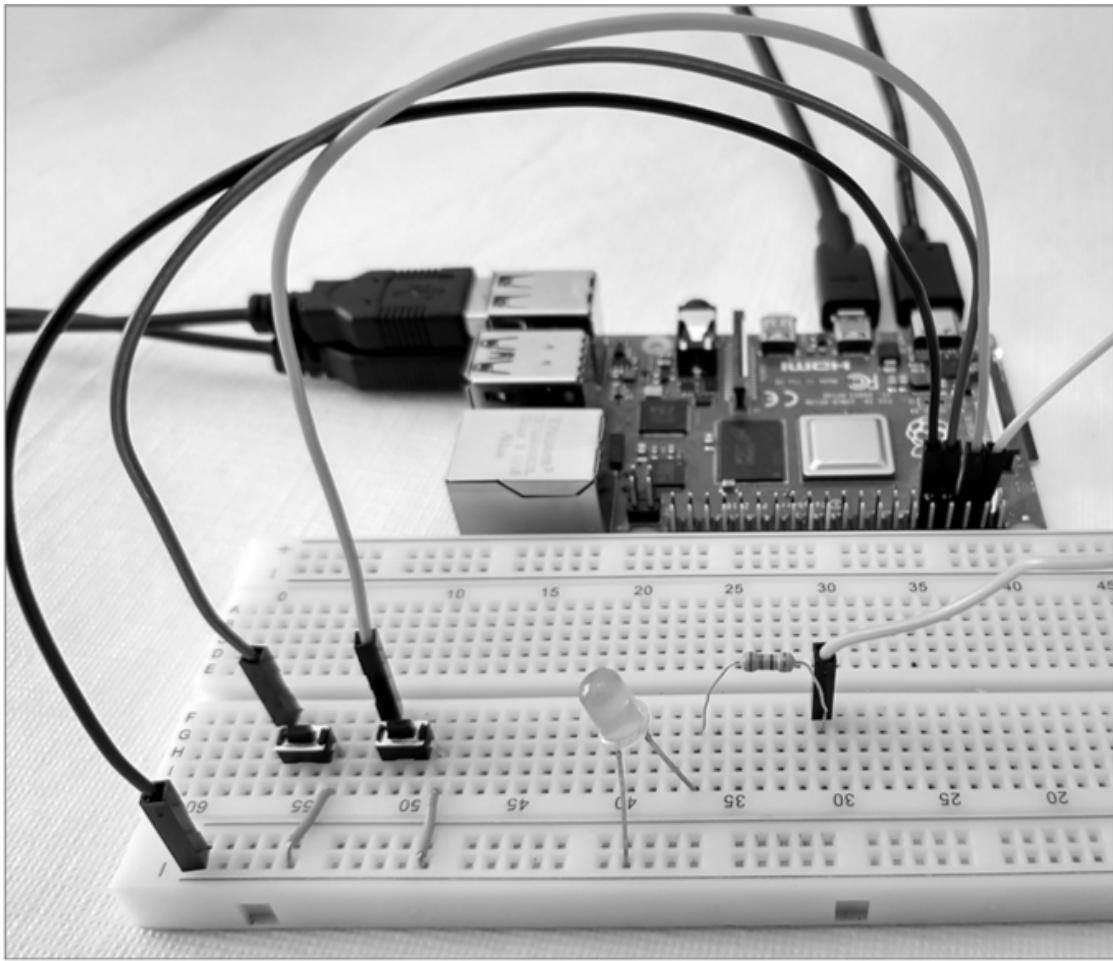


Figure 13-13: Raspberry Pi vending machine circuit on a breadboard

Once your circuit is connected and you have verified your connections, power on your Raspberry Pi. You may see the LED turn on, and that's fine, since you haven't run any code yet to set the LED to a particular state.

TEST YOUR CIRCUIT

Before entering the vending machine code, let's write a simple program to test that the circuit is properly connected and to give you a feel for working with GPIO in Python. For interacting with the GPIO pins, we're going to use *GPIO Zero*, a Python library that makes it easy to work with physical devices such as buttons and LEDs. Use the text editor of your choice to create a new file named *gpiotest.py* in the root of your home folder. Enter the following Python code into your text editor. Indentation matters in Python, so make sure you indent appropriately.

```
from time import sleep①
from gpiozero import LED, Button②
```

```
button = Button(3)❸
led = LED(2)❹

while True:❺
    led.off()
    button.wait_for_press()
    led.on()
    sleep(1)
```

This simple program imports the `sleep` function ❶ and the `LED` and `Button` classes from the `GPIO Zero` library ❷. It then creates a variable named `button` that represents the physical button on GPIO 3 ❸. Similarly, an `led` variable is created to represent the LED connected to GPIO 2 ❹. The program then enters an infinite loop ❺ that turns the LED off, waits for the button to be pressed, and then turns the LED on for one second before going through the loop again.

Once the file is saved, you can run it using the Python interpreter like so:

```
$ python3 gpiotest.py
```

When you start the program, nothing should happen at first, except perhaps the LED may turn off if it was previously on. If you press the button connected to GPIO 3, the LED should turn on for one second, and then turn off. You can repeat this as long as the program is running.

Our simple program didn't include any graceful way to exit, so to end the program, press `CTRL-C` on the keyboard. When you exit the program in this way, the Python interpreter shows you a "Traceback" of the recent function calls—this is normal.

If the program doesn't work as expected, double-check the code you entered and review "Troubleshooting Circuits" on page 340.

A VENDING MACHINE PROGRAM

In Project #7 (on page 105) and #8 (on page 107), the vending machine's logic was controlled by an SR latch, an AND gate, and a capacitor. You can now replace all of that with a program on the Raspberry Pi. This new design also gets rid of the COIN LED. Previously, the COIN LED turned on if a coin had been inserted. With this new design, the program instead prints the count of coin credits. Each time a coin is inserted, the credit count should go up one, and each time a vend operation occurs the credit count should go down by one.

The requirements for this device are the following:

- Pressing the COIN button increases the credit count by one.
- Pressing the VEND button simulates vending an item. If the credit count is greater than 0, the VEND LED briefly turns on and the credit count decreases by

- one. If the credit count is 0, nothing happens when the VEND button is pressed.
- Every actionable button press, whether COIN or VEND, causes the program to print the current number of credits.

Use the text editor of your choice to create a new file named *vending.py* in the root of your home folder. Enter the following Python code into your text editor:

```
from time import sleep❶
from gpiozero import LED, Button

vend_led = LED(2)❷
vend_button = Button(3)
coin_button = Button(4)
coin_count = 0❸
vend_count = 0

def print_credits():❹
    print('Credits: {}'.format(coin_count - vend_count))

def coin_button_pressed():❺
    global coin_count
    coin_count += 1
    print_credits()

def vend_button_pressed():❻
    global vend_count
    if coin_count > vend_count:
        vend_count += 1
        print_credits()
        vend_led.on()
        sleep(0.3)
        vend_led.off()

coin_button.when_pressed = coin_button_pressed❾
vend_button.when_pressed = vend_button_pressed

input('Press Enter to exit the program.\n')❿
```

First, the code imports the `sleep` function, `LED` class, and `Button` class ❶, all of which are used later in the program. Next, three variables are declared that represent the physical components attached to GPIO pins—`vend_led` on GPIO 2, `vend_button` on GPIO 3, and `coin_button` on GPIO 4 ❷. The variable `coin_count` is declared to track the number of times the COIN button has been pressed, and the variable `vend_count` tracks the number of times that a vending operation has occurred ❸. These two variables are used to calculate the number of credits.

The `print_credits` function ❹ prints the number of available credits, which is simply the difference between `coin_count` and `vend_count`.

The `coin_button_pressed` function ❸ is the code that runs when the COIN button is pressed. It increments `coin_count` and prints the number of credits. The `global coin_count` statement allows the global variable `coin_count` to be modified within the `coin_button_pressed` function.

The `vend_button_pressed` function ❹ is the code that runs when the VEND button is pressed. If there are credits remaining (`coin_count > vend_count`), then the function increments `vend_count`, prints the number of credits, and turns the LED on for 0.3 seconds.

Setting `coin_button.when_pressed = coin_button_pressed` ❺ associates the `coin_button_pressed` function with `coin_button` on GPIO 4 so that the function runs when the button is pressed. Similarly, `vend_button_pressed` is associated with `vend_button`.

Finally, we call the `input` function ❻. This function prints a message to the screen and waits for the user to press the ENTER key. This is a simple way to keep the program running. Without this line of code, the program would reach its end and stop running before the user had a chance to interact with the buttons.

Once the file is saved, you can run it using the Python interpreter like so:

```
$ python3 vending.py
```

When you start the program, you should immediately see `Press Enter to exit the program` displayed to the terminal window. At this point try pressing the COIN button, which is connected to GPIO 4. You should see the program print `Credits: 1`. Next try pressing the VEND button. The LED should briefly light up, and the program should print `Credits: 0`. Try pressing the VEND button again—nothing should happen. Try pressing COIN and VEND multiple times and make sure the program works as expected. When you’re finished testing the program, press ENTER to end the program.

As you can see, a Raspberry Pi, or similar device, can replicate in software the same logic that we previously implemented in hardware. However, a software-based solution is much easier to modify. New features can be added by changing a few lines of code rather than adding new chips and wiring. A Raspberry Pi is actually overkill for what we wanted to do here; the same thing could be accomplished with a less capable computing device at even lower cost, but the principle is the same.

AN IOT VENDING MACHINE

Let’s say that the operator of the vending machine wants to be able to check the machine’s status remotely, over the internet. Since you’re using a Raspberry Pi for your vending machine’s logic, you can take things a step further and make this an IoT vending machine! You can add a simple web server to the program, allowing someone to connect to the device’s IP address from a web browser and see how many times a coin has been inserted and how many times a vending operation occurred.

Python makes this relatively easy, because it includes a simple web server library, `http.server`. You just need to construct some HTML that includes the data you want to send and write a handler for incoming GET requests. You also need to start the web server when the program begins.

Use the text editor of your choice to edit your existing `vending.py` file in the root of your home folder. Start by inserting the following import statement as the first line of the file (leaving all the existing code intact, just shifted down a line):

```
from http.server import BaseHTTPRequestHandler, HTTPServer
```

Next, remove the entire `input` line at the bottom of the file and add this code to the end of the file:

```
HTML_CONTENT = """
<!DOCTYPE html>
<html>
  <head><title>Vending Info</title></head>
  <body>
    <h1>Vending Info</h1>
    <p>Total Coins Inserted: {0}</p>
    <p>Total Vending Operations: {1}</p>
  </body>
</html>
"""

class WebHandler(BaseHTTPRequestHandler):❶
    def do_GET(self):❷
        self.send_response(200)❸
        self.send_header('Content-type', 'text/html')❹
        self.end_headers()
        response_body = HTML_CONTENT.format(coin_count,
vend_count).encode()❺
        self.wfile.write(response_body)

    print('Press CTRL-C to exit program.')
    server = HTTPServer(('', 8080), WebHandler)❻
    try:❽
        server.serve_forever()❾
    except KeyboardInterrupt:
        pass
    finally:
        server.server_close()❿
```

`HTML_CONTENT` is a multiline string that defines the HTML code that the program sends over the network. This block of HTML code represents a simple web page with a `<title>`, a `<h1>` heading, and two `<p>` paragraphs that describe the state of the vending machine. Specific values in these paragraphs are represented as placeholders `{0}` and `{1}`.

These values are filled in by the program when it runs. Since this is HTML, the spacing and line breaks within this string don't matter.

The `WebHandler` class ❶ describes how the web server handles incoming HTTP requests. It inherits from the `BaseHTTPRequestHandler` class, meaning that it has the same methods and fields as `BaseHTTPRequestHandler`. However, this just gives you a generic HTTP request handler; you still need to specify how your program will respond to specific HTTP requests. In this case, the program only needs to respond to HTTP GET requests, so the code defines the `do_GET` method ❷. This method is invoked when a GET request comes to the server, and it replies with the following:

- A 200 status code indicating success ❸
- A `Content-type: text/html` header that tells the browser to expect the response to be HTML ❹
- The HTML string that was defined earlier, but with the two placeholders replaced by the values of `coin_count` and `vend_count` ❺

A web server instance is created using the `HTTPServer` class ❻. Here you specify that the server name can be anything and that the HTTP server listens on port 8080 ('', 8080). This is also where you specify to use the `WebHandler` class for inbound HTTP requests.

The web server starts with `server.serve_forever()` ❼. This is placed in a `try/except/finally` block ➋ so that the server continues running until a `KeyboardInterrupt` exception occurs (generated by CTRL-C). When this happens, `server.server_close()` is called to clean up, and the program ends ❼.

Once the file is saved, you can run the file using the Python interpreter like so:

```
$ python3 vending.py
```

The program should behave as it did before when you press the COIN or VEND buttons. However, now you can also connect to the device from a web browser and see data about the vending machine. To do this, you need a device on the same local network as the Raspberry Pi, unless your Pi is directly connected to the internet with a public IP address, in which case any device on the internet should be able to connect to it. If you don't have another device, you can launch a web browser on the Raspberry Pi itself and let the Raspberry Pi act as both the client and server.

You need to find the IP address of your Raspberry Pi. We did this in Project #30 on page 255 if you want to review the details, but this is the command you want to use:

```
$ ifconfig
```

Once you have the IP address of your Raspberry Pi, open a web browser on the device you want to use as your client. In the address bar, enter the following: <http://IP:8080>, replacing IP with your Raspberry Pi's IP address. The end result should

look something like this: `http://192.168.1.41:8080`. Once you've entered this into the browser's address bar, you should see the web page load with the count of coins and vending operations. Each time you request this page, you should see the Python program print information about the request to the terminal. Once the web page is loaded, it won't automatically reload, so if you press the COIN or VEND buttons additional times and want to see the latest values, you need to refresh your browser's view of the page. To stop this program, use CTRL-C on the keyboard.

Recall from Chapter 12 that websites are either static or dynamic. The site you ran in the Chapter 12 projects was static—it served content that was built ahead of time. In contrast, the vending machine site in this chapter is dynamic. It generates an HTML response when a request comes in. Specifically, it updates the coin and vending values in the HTML content before responding.

As a bonus challenge, try modifying the program to also display the “credits” value on the web page. This value should match the last credits value that was printed to the terminal.

A ANSWERS TO EXERCISES



Here you'll find questions and answers for the exercises included in this book. Some questions don't have a single right answer; for those, I've included an example answer. You'll get the most out of these exercises if you come up with an answer yourself before you read the solution found here!

1-2: Binary to Decimal

Exercise: Convert these numbers, represented in binary, to their decimal equivalents.

Solution:

$$10 \text{ (binary)} = 2 \text{ (decimal)}$$

$$111 \text{ (binary)} = 7 \text{ (decimal)}$$

$$1010 \text{ (binary)} = 10 \text{ (decimal)}$$

1-3: Decimal to Binary

Exercise: Convert these numbers, represented in decimal, to their binary equivalents.

Solution:

3 (decimal) = 11 (binary)

8 (decimal) = 1000 (binary)

14 (decimal) = 1110 (binary)

1-4: Binary to Hexadecimal

Exercise: Convert these numbers, represented in binary, to their hexadecimal equivalents. Don't convert to decimal if you can help it! You can use Table 1-5 to help you. The goal is to move directly from binary to hexadecimal.

Solution:

10 (binary) = 2 (hexadecimal)

11110000 (binary) = F0 (hexadecimal)

1-5: Hexadecimal to Binary

Exercise: Convert these numbers, represented in hexadecimal, to their binary equivalents. Don't convert to decimal if you can help it! You can use Table 1-5 to help you. The goal is to move directly from hexadecimal to binary.

Solution:

1A (hexadecimal) = 0001 1010 (binary)

C3A0 (hexadecimal) = 1100 0011 1010 0000 (binary)

2-1: Create Your Own System for Representing Text

Exercise: Define a way to represent the uppercase letters A through D as 8-bit numbers, and then encode the word *DAD* into 24 bits using your system. Bonus: Show your encoded 24-bit number in hexadecimal too. Table A-1 presents an example answer; there's no single right answer.

Solution:

Table A-1: A Custom System for Representing A–D with Bytes

Character	Binary
A	00000001
B	00000010
C	00000011
D	00000100

DAD using this system would be 00000100 00000001 00000100 (spaces added for clarity). Written as hexadecimal: 0x040104.

2-2: Encode and Decode ASCII

Exercise: Using Table 2-1, encode the following words to ASCII binary and hexadecimal, using a byte for each character. Remember that there are different codes for uppercase and lowercase letters.

Solution:

Text Hello

Binary 01001000 01100101 01101100 01101100 01101111

Hexadecimal 0x48656C6C6F

Text 5 cats

Binary 00110101 00100000 01100011 01100001 01110100
01110011

Hexadecimal 0x352063617473

Note how encoding “5 cats” gave us 0b00110101 as the binary representation of the character 5. This is different from the number 5, which is 0b101. The character represents the symbol (5) we use for the number five, while the number represents a quantity. From the same encoding of “5 cats,” note that even the space character, which you might think of as empty, still requires a byte to represent.

Exercise: Using Table 2-1, decode the following words. Each character is represented as an 8-bit ASCII value with spaces added for clarity.

Solution:

Binary 01000011 01101111 01100110 01100110 01100101
01100101

Text Coffee

Binary 01010011 01101000 01101111 01110000

Text Shop

Exercise: Using Table 2-1, decode the following word. Each character is represented as an 8-bit hexadecimal value with spaces added for clarity.

Solution:

Hexadecimal 43 6C 61 72 69 6E 65 74

Text Clarinet

2-3: Create Your Own System for Representing Grayscale

Exercise: Define a way to represent black, white, dark gray, and light gray.

Solution: If we go with a 2-bit system, the four unique values for a 2-bit number are 00, 01, 10, and 11. Each of those four binary numbers can then be mapped to a color: black, white, dark gray, and light gray—the specific mapping is up to you, since you are designing your own system. Table A-2 presents an example answer; there's no single right answer.

Table A-2: A Custom System for Representing Grayscale

Color	Binary
black	00
dark gray	01
light gray	10
white	11

2-4: Create Your Own Approach for Representing Simple Images

Exercise: Part 1 Building upon your previous system for representing grayscale colors, design an approach for representing an image composed of those colors. If you want to simplify things, you can assume that the image will always be 4 pixels by 4 pixels, like the one in Figure 2-1.

Solution: Assume the image will always be 4 pixels by 4 pixels, and therefore we need to represent 16 pixels, one color per pixel. Using the previously defined system for representing grayscale colors in Table A-2, we need 2 bits to represent the color of each pixel. So to represent our full image of 16 pixels, with 2 bits per pixel, we need $16 \times 2 = 32$ total bits.

When we encode the data as binary, in what order should we represent the 16 pixels? This decision is somewhat arbitrary, but for this example, let's order our data from left to right, top to bottom, as shown in Figure A-1.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure A-1: Order of pixels in example image format

Using the approach shown in Figure A-1, when we encode our data in binary, the first 2 bits are the color of pixel 1, and the next 2 bits are

the color of pixel 2, and so on. We then use the color codes defined in the previous exercise to define the color of each pixel. For example, if pixel 1 is white, pixel 2 is black, and pixel 3 is dark gray, the first 6 bits in our image data are 110001.

Exercise: Part 2 Using your approach from part 1, write out a binary representation of the flower image from Chapter 2, Figure 2-1.

Solution: Here, I provide an example of how this can work by applying the example approach from part 1 to this problem. To help visualize this, Figure A-2 overlays the numbered image grid on the flower image.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure A-2: A 4X4 image grid overlaid on a flower image

Now that we have assigned each pixel in the grid a number, we can refer to Table A-2 to apply a 2-bit value to each square, progressing from square 1 to 16. The end result is the following binary sequence that represents the grayscale flower image:

1111110111101101111110101100101

NOTE

I wrote a simple web page that simulates this particular system of 16 pixels and 2-bit grayscale. Try it here: <https://www.howcomputersreallywork.com/grayscale/>.

2-5: Write a Truth Table for a Logical Expression

Exercise: Table 2-7 shows three inputs for a logical expression. Complete the truth table output for the expression (A OR B) AND C.

Solution:

Table A-3: (A OR B) AND C Truth Table Solution

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

3-1: Using Ohm's Law

Excercise: Take a look at the circuit in Figure A-3. What is the current, I ?

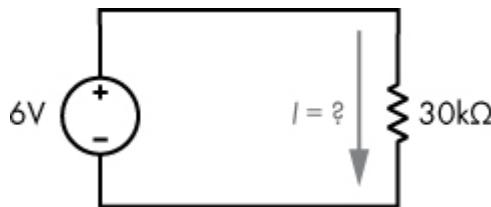


Figure A-3: Find the current using Ohm's law.

Solution: Ohm's law tells us that current is voltage divided by resistance. So I is 0.2 millamps, as shown here:

$$I = \frac{6V}{30,000\Omega} = .0002 A = 0.2 \text{ mA}$$

3-2: Find the Voltage Drops

Exercise: Given the circuit in Figure 3-11, what is the current, I ? What is the voltage drop across each resistor? Find the labeled voltages: V_A , V_B , V_C , and V_D , each measured as relative to the negative terminal of the power supply.

Solution: The total resistance is $24\text{k}\Omega + 6\text{k}\Omega + 10\text{k}\Omega = 40\text{k}\Omega$. This influences the current through the circuit, which we can calculate using Ohm's law: $10\text{V} / 40\text{k}\Omega = 0.25 \text{ mA}$, as shown in Figure A-4.

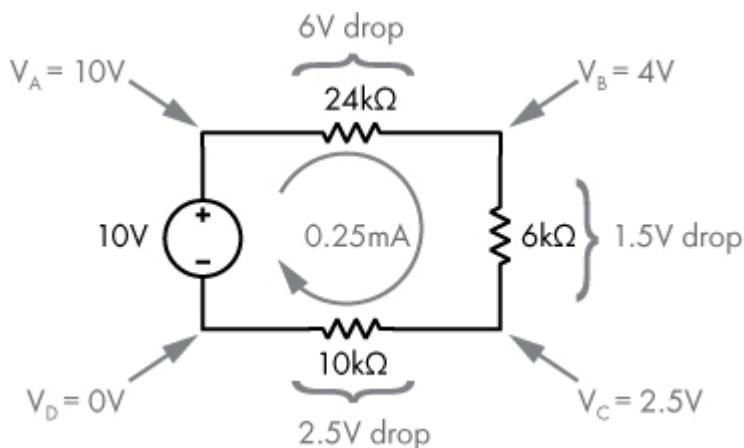


Figure A-4: Voltage drops around a circuit

Now calculate the voltage drop across the $24\text{k}\Omega$ resistor using Ohm's law: $V = 0.25\text{mA} \times 24\text{k}\Omega = 6\text{V}$. That means that V_B will be 6V less than V_A . So $V_B = 10\text{V} - 6\text{V} = 4\text{V}$. The $6\text{k}\Omega$ resistor drops $0.25\text{mA} \times 6\text{k}\Omega = 1.5\text{V}$. Therefore, $V_C = V_B - 1.5\text{V} = 2.5\text{V}$. That leaves 2.5V to drop across the $10\text{k}\Omega$ resistor, which we can deduce from Kirchhoff's voltage law or calculate using Ohm's law.

4-1: Design a Logical OR with Transistors

Exercise: Draw a circuit diagram for a logical OR circuit that uses transistors for inputs A and B. Adapt the circuit in Figure 4-4 that uses mechanical switches, but use NPN transistors instead.

Solution: Figure A-5 shows a solution for implementing a logical OR with NPN transistors.

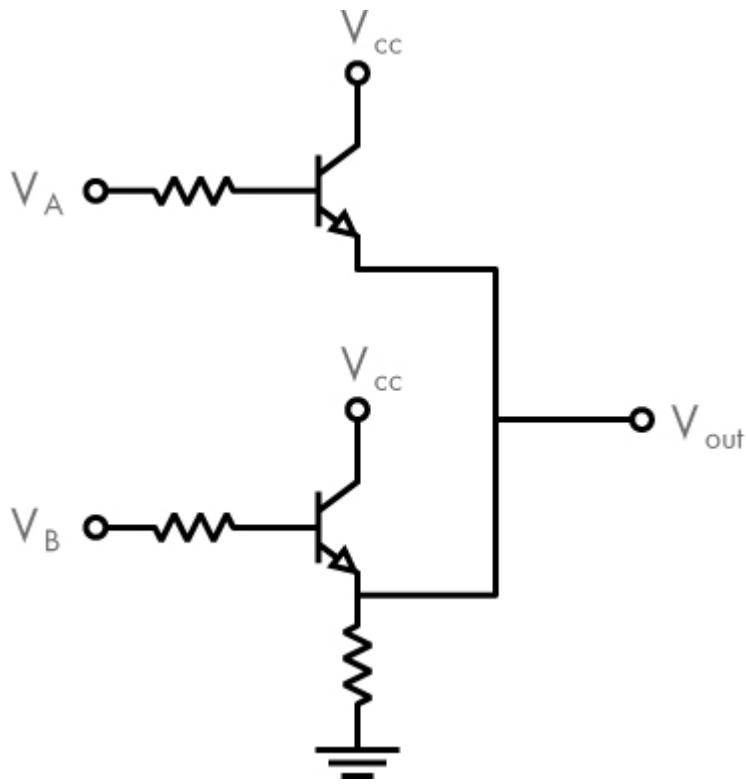


Figure A-5: Logical OR implemented with NPN transistors

4-2: Design a Circuit with Logic Gates

Exercise: In Chapter 2, Exercise 2-5, you created the truth table for $(A \text{ OR } B) \text{ AND } C$. Now build on that work and translate that truth table and logical expression into a circuit diagram. Draw a logic gate diagram (similar to the one in Figure 4-11) for the circuit using logic gates.

Solution: Figure A-6 shows a solution for implementing $(A \text{ OR } B) \text{ AND } C$.

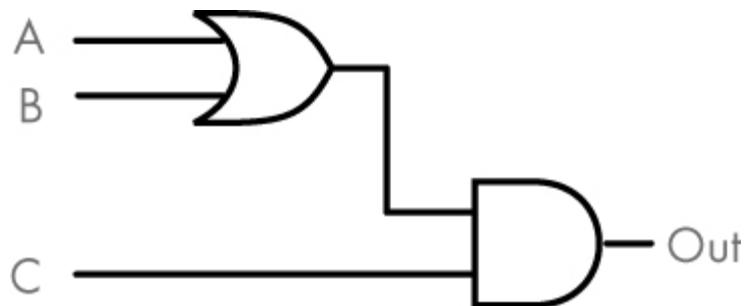


Figure A-6: Logic gate diagram for $(A \text{ OR } B) \text{ AND } C$

5-1: Practice Binary Addition

Exercise: Try the following addition problems.

Solution: The leading 0s in the answers are optional.

$$0001 + 0010 = 0011$$

$$0011 + 0001 = 0100$$

$$0101 + 0011 = 1000$$

$$0111 + 0011 = 1010$$

5-2: Find the Two's Complement

Exercise: Find the 4-bit two's complement of 6.

Solution: See Figure A-7.

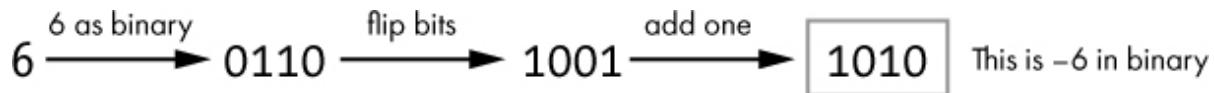


Figure A-7: Finding the two's complement of 6

5-3: Add Two Binary Numbers and Interpret as Signed and Unsigned

Exercise: Add 1000 and 0110. Interpret your work as signed numbers. Then interpret it as unsigned. Do the results make sense?

Solution: See Figure A-8.

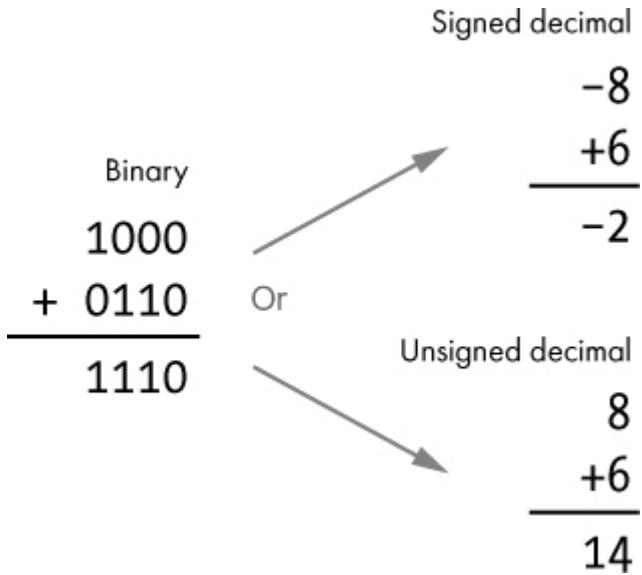


Figure A-8: Add 1000 and 0110

7-1: Calculate the Required Number of Bits

Exercise: Using the techniques described in Chapter 7, determine the number of bits required for addressing 4GB of memory. Remember that each byte is assigned a unique address, which is just a number.

Solution: Looking back at Chapter 1 as a reference on SI-prefixes, 1GB of memory is 2^{30} or 1,073,741,824 bytes. So 4GB is 4 times that number, or 4,294,967,296 bytes. If we take $\log_2(4,294,967,296)$ we get 32. So with 32 bits, we can represent a unique address for every byte in 4GB of memory.

If your calculator or application does not provide a \log_2 function, note that

$$\log_2(n) = \frac{\log(n)}{\log(2)}$$

With that bit of information, you can find $\log_2(4,294,967,296)$ by taking $\log(4,294,967,296)$ and dividing it by $\log(2)$. This should give you a result of 32.

We can also arrive at this solution using a different approach. Since memory addresses are assigned starting with 0 rather than 1, the range of memory addresses for 4GB of memory is from 0 to 4,294,967,295 (1 less than the number of bytes). In hex, 4,294,967,295 is 0xFFFFFFFF. That is 8 hex digits, and since each hexadecimal symbol represents 4 bits, we easily see that $4 \times 8 = 32$ bits are required.

8-1: Use Your Brain as a CPU

Exercise: Try running the following ARM assembly program in your mind, or use pencil and paper:

Address	Assembly
0001007c	subs r3, r0, #1
00010080	ble 0x10090
00010084	mul r0, r3, r0
00010088	subs r3, r3, #1
0001008c	bne 0x10084
00010090	---

Assume an input value of $n = 4$ is initially stored in $r0$. When the program gets to the instruction at `00010090` you've reached the end of the code, and $r0$ should be the expected output value of 24. I recommend that for each instruction, you keep track of the values of $r0$ and $r3$ before and after the instruction completes. Work through the instructions until you reach the instruction at `00010090` and see if you got the expected result. If things worked correctly, you should have looped through the same instructions several times; that's intentional.

Solution: Once you've worked through this exercise, look at Table A-4 to see each step of running the assembly code. Each row in the table represents an execution of a single instruction. For each instruction, we track the values of $r0$ and $r3$. An arrow (\rightarrow) means the register value changed from the value on the left to the value on the right. In the Notes column, I use an equals sign to mean “is set to” rather than as a mathematical check of equality. For example, $r0 = r3 \times r0$ means “ $r0$ is set to the product of $r3$ and $r0$.”

Table A-4: Factorial Assembly Code, Step by Step

Address	Instruction	r0	r3	Notes
		4	?	You want to calculate the factorial of 4, so set r0 = 4 before the code runs. r3 is initially unknown.
0001007c	subs r3, r0, #1	4	? → 3	$r3 = r0 - 1 = 4 - 1 = 3$
00010080	ble 0x10090	4	3	$r3 > 0$, so don't branch; instead, continue to 10084.
00010084	mul r0, r3, r0	4 → 12	3	$r0 = r3 \times r0 = 3 \times 4 = 12$
00010088	subs r3, r3, #1	12	3 → 2	Decrement r3.
0001008c	bne 0x10084	12	2	r3 is not 0, so branch to 10084.
00010084	mul r0, r3, r0	12 → 24	2	$r0 = r3 \times r0 = 2 \times 12 = 24$
00010088	subs r3, r3, #1	24	2 → 1	Decrement r3.
0001008c	bne 0x10084	24	1	r3 is not 0, so branch to 10084.
00010084	mul r0, r3, r0	24 → 24	1	$r0 = r3 \times r0 = 1 \times 24 = 24$
00010088	subs r3, r3, #1	24	1 → 0	Decrement r3.
0001008c	bne 0x10084	24	0	r3 is 0, so don't branch; instead, continue to 10090.
00010090		24	0	We are finished with the algorithm, and the result can be found in r0, which is now equal to 24, as expected.

Hopefully this table matches the outcomes you saw when you tried this on your own. Now that we've walked through the code for $n = 4$, consider the following questions:

1. If we calculate the factorial of 1 by initially setting $r_0 = 1$, what happens?
2. The mathematical definition of *factorial* says that the factorial of 0 is 1. Does our algorithm work for that scenario? What specific result do we get if we initially set $r_0 = 0$?
3. You may have noticed that the expected result of 24 was stored in r_0 on the next-to-last iteration through the code. That is, the program loops an additional time, but this has no bearing on the value of r_0 . Why do you think the code was written this way?
4. Given that we are using 32-bit registers, is there a practical upper limit for n ? That is, can a value of n be provided where the result will be too large to fit in a 32-bit register?

Here are the answers to these questions:

1. The first `subs` instruction sets $r_3 = 0$, and the following `b1e` instruction jumps to `0x10090`, since r_3 is 0. At this point our result in r_0 is still 1, which is the expected output.
2. No, our algorithm won't work. The first `subs` instruction sets r_3 to -1, and the following `b1e` instruction jumps to `0x10090`, since r_3 is negative. At this point our result in r_0 is still 0, which is not the expected output.
3. The factorial of n is the product of the positive integers less than or equal to n . Staying true to this definition means multiplying r_0 by 1, even though doing so doesn't change the final result. That means one extra loop through the code while r_3 is equal to 1. We could improve the efficiency of the code by skipping this multiplication by 1, but I left it in place to stay true to the mathematical definition of a factorial.

4. The maximum value that a 32-bit integer can represent is $2^{32} - 1 = 4,294,967,295$. Or if we need to represent negative numbers too, the largest value is 2,147,483,647. So if we try to calculate a factorial where the result is larger than about 4 billion (or 2 billion), we get an inaccurate result. It turns out that $n = 12$ is the largest value of n that we can use. The factorial of 13 is over 6 billion, which is too large to fit in a 32-bit integer.

9-1: Bitwise Operators

Exercise: Consider the following Python statements. What will be the values of `a`, `b`, and `c` after this code executes?

```
x = 11
y = 5
a = x & y
b = x | y
c = x ^ y
```

Solution: Figure A-9 shows how the bitwise operations of AND, OR, and XOR work when applied to the values of 11 and 5.

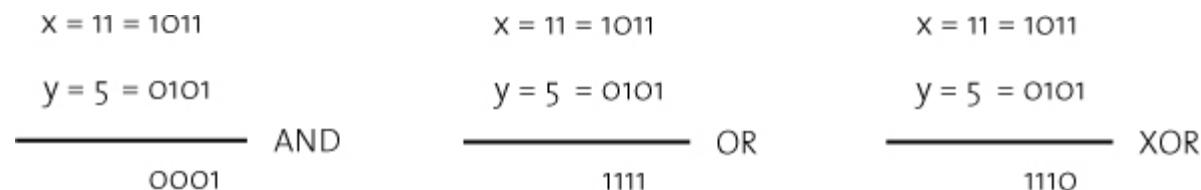


Figure A-9: Bitwise operations on two values

So, the value of `a` is 1. The value of `b` is 15 (1111 binary). The value of `c` is 14 (1110 binary).

9-2: Run a C Program in Your Mind

Exercise: Try running the following function in your head or use pencil and paper. Assume an input value of $n = 4$. When the function returns, the returned result should be the expected value of 24. I recommend that, for each line, you keep track of the values of `n` and `result` before and

after the statement completes. Work through the code until you reach the end of the `while` loop and see if you get the expected result.

Note that the condition of the `while` loop (`--n > 0`) places the decrement operator (`--`) before the variable `n`. This means that `n` is decremented *before* its value is compared to 0. This happens each time the `while` loop condition is evaluated.

```
// Calculate the factorial of n.
int factorial(int n)
{
    int result = n;

    while(--n > 0)
    {
        result = result * n;
    }

    return result;
}
```

Solution: Before you read on, I strongly recommend you attempt to complete this exercise! You'll learn more if you do this yourself. Once you've worked through this exercise, look at Table A-5 to see each step of running our example C code. An arrow (\rightarrow) means the variable value changed from the value on the left to the value on the right.

Table A-5: Factorial C Code, Step by Step

Statement	Result	<code>n</code>	Notes
<code>int factorial(int n)</code>	?	4	We want to calculate the factorial of 4, so set <code>n</code> = 4 as an input to our function.
<code>int result =</code>	$? \rightarrow 4$	4	Initially, set <code>result</code> to the value of <code>n</code> .
<code>n;</code>			
<code>while(--n > 0)</code>	4	4 \rightarrow 3	Decrement <code>n</code> .
		3	<code>n > 0</code> , so run the body of the <code>while</code> loop.
<code>result =</code>	4 \rightarrow 12	3	<code>result = 4 × 3</code>
<code>result * n;</code>	12		
<code>while(--n > 0)</code>	12	3 \rightarrow 2	Decrement <code>n</code> .
		2	<code>n > 0</code> , so run the <code>while</code> loop body again.

Statement	Result	n	Notes
result = 12 → 2			result = 12×2
result * n; 24			
while(--n > 0) 24	2 → 1		Decrement n. n > 0, so run the while loop body again.
result = 24 → 1			result = 24×1
result * n; 24			
while(--n > 0) 24	1 → 0		Decrement n. n = 0, so exit the while loop.
return result; 24	0		We are finished with the function, and the calculated value can be found in result, which is now equal to 24, as expected.

Hopefully this table matches the outcomes you saw when you tried this on your own.

11-1: Which IPs Are on the Same Subnet?

Exercise: Is IP address 192.168.0.200 on the same subnet as your computer? Assume your computer has an IP address of 192.168.0.133 and a subnet mask of 255.255.255.224.

Solution: As we found in Chapter 11, the network ID of your computer's subnet is 192.168.0.128. Assuming two devices are on the same subnet, they'll share a subnet mask and network ID. A logical AND of the other computer's IP address and our subnet mask gives us a network ID.

IP = 192.168.0.200	= 11000000.10101000.00000000.11001000
MASK = 255.255.255.224	= 11111111.11111111.11111111.11000000
AND = 192.168.0.192	= 11000000.10101000.00000000.11000000 = The network id

The network ID of the other computer (192.168.0.192) does not match the network ID of your computer (192.168.0.128), so they are on different subnets. This means that communication between these hosts needs to go through a router.

11-2: Research Common Ports

Exercise: Find the port numbers for common application layer protocols. What are the port numbers for Domain Name System (DNS), Secure Shell (SSH), and Simple Mail Transfer Protocol (SMTP)? You can find this information online with a search, or by looking at the IANA registry, here: <http://www.iana.org/assignments/port-numbers>. The IANA listings sometimes use an unexpected term for the service name. For example, DNS is simply listed as “domain.”

Solution:

- DNS: 53
- SSH: 22
- SMTP: 25

12-1: Identify the Parts of a URL

Exercise: For the following URLs, identify the scheme, username, host, port, path, and query. Not all URLs include all these parts.

Solution:

https://example.com/photos?subject=cat&color=black

scheme *https*

host *example.com*

path *photos*

query *subject=cat&color=black*

http://192.168.1.20:8080/docs/set5/two-trees.pdf

scheme *http*

host *192.168.1.20*

port *8080*

path *docs/set5/two-trees.pdf*

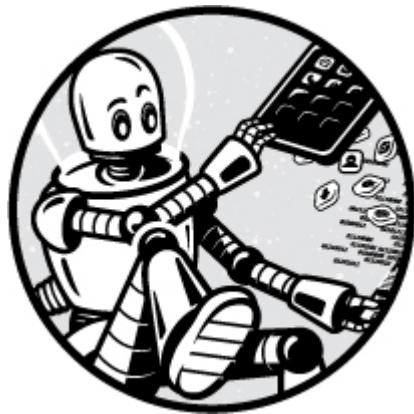
mailto:someone@example.com

scheme *mailto*

username *someone*

host *example.com*

B RESOURCES



This appendix contains information to help you get started with the projects. We cover how to find electronic components for purchase, how to power digital circuits, and how to set up a Raspberry Pi.

Buying Electronic Components for the Projects

Working with electronics and programming in a hands-on way helps you learn the concepts in this book, but trying to obtain various components can be intimidating. This section helps you find the electronic components you'll need for the projects.

Here's the full list of all the components used in the projects, in case you want to order everything at once:

- Breadboards (at least one 830-point breadboard. You can get by with only one breadboard if you plan to tear down each circuit between exercises. If you wish to keep your circuits intact, you need more than one.)
- Resistors (an assortment of resistors. Here are the specific values used: $47\text{k}\Omega$, $10\text{k}\Omega$, $4.7\text{k}\Omega$, $1\text{k}\Omega$, 470Ω , 330Ω , 220Ω .)

- Digital multimeter
- 9-volt battery
- 9-volt battery clip connector
- Pack of 5mm or 3mm red LEDs (light-emitting diodes)
- Two NPN BJT transistors, model 2N2222 in TO-92 packaging (also known as PN2222)
- Jumper wires designed for use in a breadboard (both male-to-male and male-to-female)
- Pushbuttons or slide switches that fit in a breadboard
- 7402 integrated circuit
- 7408 integrated circuit
- 7432 integrated circuit
- Two 7473 integrated circuits
- 7486 integrated circuit
- 220 μ F electrolytic capacitor
- 10 μ F electrolytic capacitor
- 5-volt power supply (See the section “Powering Digital Circuits” on page 336 for details.)
- Raspberry Pi and related items (See the section “Raspberry Pi” on page 341 for details.)
- Recommended: Alligator clips (These can make it easier to connect a battery to a breadboard or your multimeter to a circuit.)
- Optional: Wire stripper (You may need one to strip away plastic from the ends of wires and expose the copper.)

Although this list calls out specific counts for certain components, for some parts, you probably want to buy a few more than the stated number in case of damage or in case you wish to experiment. I recommend a few spare transistors and one spare of each integrated circuit.

7400 Part Numbers

Finding an appropriate 7400 series integrated circuit (IC) can be challenging, since these chips are identified with part numbers that include more detail than just the 74xx identifier. The 7400 series has a number of subfamilies, each with its own part numbering scheme. Plus, manufacturers add their own prefixes and suffixes to the part numbers. This can be confusing at first, so let's look at an example. I recently wanted to purchase a 7408 IC, but the part number I actually ordered was an SN74LS08N. Let's break down that part number in Figure B-1.

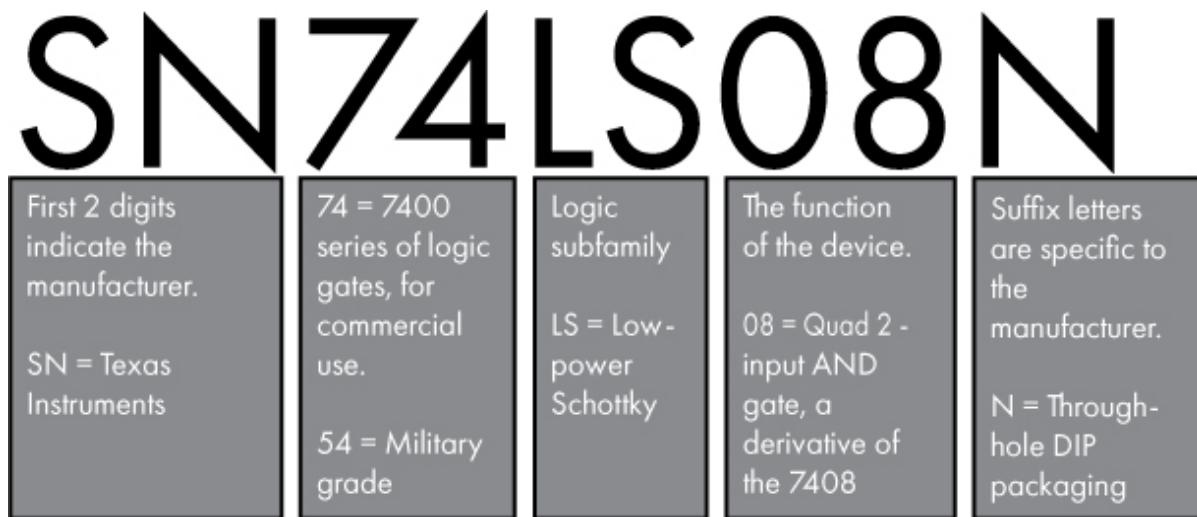


Figure B-1: Interpreting a 7400 series part number

So the SN74LS08N is a 7408 AND gate manufactured by Texas Instruments, in the low-power Schottky subfamily, packaged in through-hole packaging. No need to worry about the details of “low-power Schottky,” other than to know that it’s a common subfamily of parts that works for our purposes.

For the projects in this book, you want to ensure that you use parts that are compatible with each other. The projects assume that you are using parts that are compatible with the original 7400 logic levels (5V). Considering what is readily available, I’d recommend that you buy parts in the LS or HCT series. So if you want a 7408, you could buy an **SN74LS 08N** or **SN74HCT 08N**. Generally speaking, you should be able to mix LS and HCT parts in the same circuit. The prefix letters, SN in this example, don’t matter when it comes to compatibility; you

don't need to buy parts from a specific manufacturer. The suffix does matter, since it indicates the package type. Be sure to get parts that fit on a breadboard—N parts work well.

Shopping

If you happen to have a local electronics store nearby that can supply these parts, then I'd suggest you visit it. The store employees can help ensure that you get what you need. However, I've found that such stores are increasingly rare; you may not be able to get everything you need locally.

Your next option is to shop online. To make things easier for you, I've put together a web page with links to parts you need from various stores: <https://www.howcomputersreallywork.com/parts/>. Or if you want to shop around yourself, Table B-1 is a list of some of the popular stores where you can get parts; I've included notes on each. I'm not endorsing these shops in particular; you may find parts elsewhere. Of course, the status of these online stores could change after this book is published.

Table B-1: Shopping Online for Electronic Components

Store	Comments
Adafruit https://www.adafruit.com/	Adafruit has a great selection of electronics parts; however, the last time I checked, they don't carry individual 7400 series logic gates.

Store	Comments
Amazon https://www.amazon.com/	Amazon carries popular items like the Raspberry Pi, but ordering integrated circuits often either isn't an option or is costly. Some items are only available in packs of 50 or 100, for example. You probably don't need 100 transistors, but if you are a Prime member, it actually might be more cost-effective to buy 100 transistors from Amazon than to buy 5 from other stores, once you factor in shipping.
Digi-Key Electronics https://www.digikey.com/	Digi-Key is targeted at professionals rather than hobbyists, and the site can be a bit intimidating, but you can find integrated circuits here, including the 7400 series logic circuits.
Mouser Electronics https://www.mouser.com/	Mouser is similar to Digi-Key in that it targets professionals and carries integrated circuits.
SparkFun https://www.sparkfun.com/	As with Adafruit, you can find lots of electronics here, but no 7400 logic gates, at least when I last checked.
Texas Instruments https://store.ti.com/	Texas Instruments manufactures 7400 series logic circuits, and they also sell them directly through their website. I've found that their prices are reasonable, and shipping options are good.

A number of websites are dedicated to helping people find electronic parts. These sites provide a user interface that's easy to navigate, and they allow for price comparison across multiple retailers. Two that I've found useful are Octopart (<https://octopart.com>) and Findchips (<https://www.findchips.com>).

Powering Digital Circuits

7400 series logic chips need 5V, so using a 9-volt battery isn't an option for powering these integrated circuits. Let's look at some options for powering your 7400 circuits.

USB Charger

Many smartphone chargers made since 2010 have a micro-USB connector. Around 2016, USB Type-C (or just USB-C) connectors began to become more common. Fortunately, USB of either variety provides 5V DC. So a USB charger, like the one shown in Figure B-2, is great for powering your 7400 series integrated circuits. If you're like me, you may have a bunch of old micro-USB phone chargers around your home already.



Figure B-2: A micro-USB phone charger

However, there's a challenge; a micro-USB connector doesn't plug into a breadboard, at least not without some help! A good option for using a USB charger with a breadboard is to buy a *micro-USB breakout board*, like the one shown in Figure B-3. This lets you plug your USB charger into the breakout board, and the breakout board plugs into your breadboard. Adafruit, SparkFun, and Amazon all carry these. Some soldering may be required. These boards typically have five pins, but for this purpose, you only need to concern yourself with the VCC (5V) pin and GND (ground) pin. When connecting this to a breadboard, remember to orient the pins in such a way that they are not connected to each other, as shown in Figure B-3.

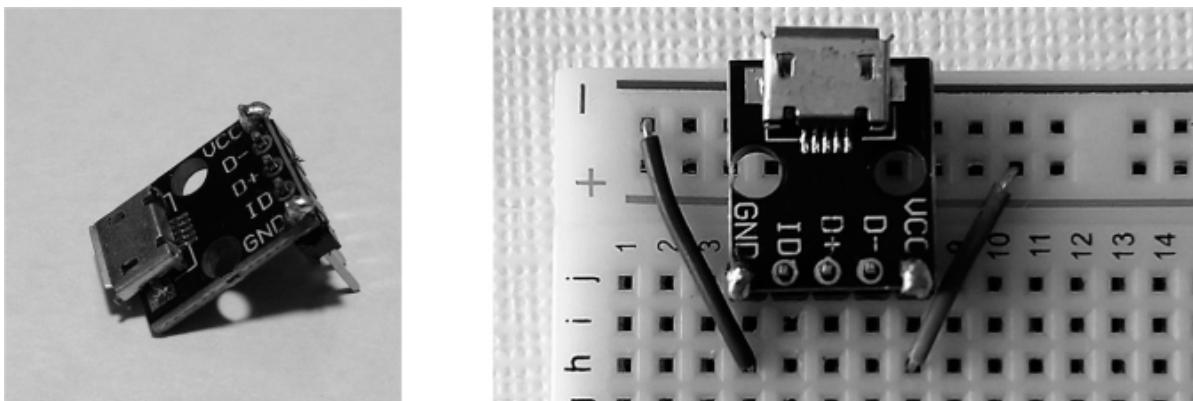


Figure B-3: A micro-USB breakout board, inserted in breadboard (right)

Breadboard Power Supply

Another option is to buy a breadboard power supply, like DFRobot DFR0140 or YwRobot Power MB V2 545043. These handy devices plug into your breadboard and are powered by a wall DC power supply with a 2.1mm barrel jack. The source DC supply should provide a voltage between 6V and 12V (be sure to verify the specific voltage allowed for the particular board you use). These 2.1mm DC power supplies are fairly common for powering consumer electronics—you may have several already—and this type of board just makes it easy to convert the voltage to 5V and connect it to a breadboard. Figure B-4 shows one of these common DC power supplies with a 2.1mm barrel jack along with a breadboard power supply.

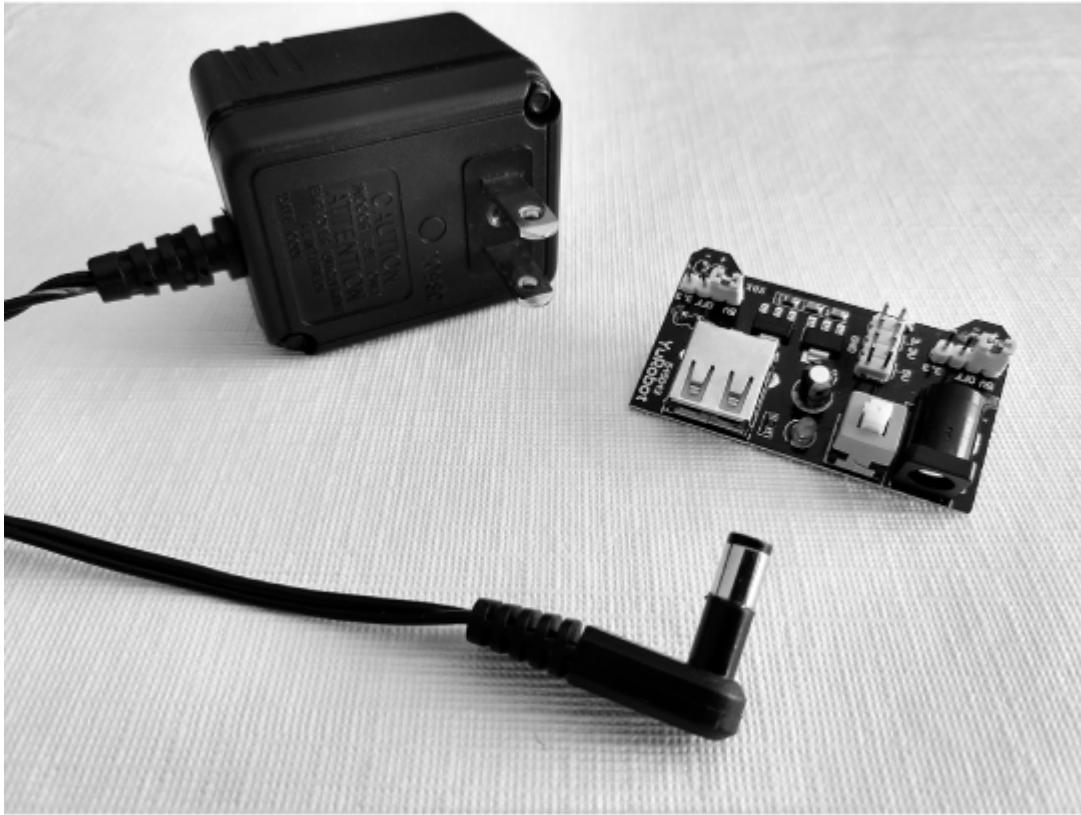


Figure B-4: A 2.1mm barrel jack power supply and a breadboard power supply

One word of caution: I've personally seen these boards experience a failure in their voltage regulator, causing the board to output a voltage higher than 5V. When connecting one of these power supplies, don't assume that the output voltage is actually 5V. Test the output voltage before connecting your circuit! Using a lower input DC voltage should help reduce this risk as well, so I'd recommend that you use a 9-volt or lower DC power supply, given an allowed range of 6V to 12V. These boards also have the option to output 3.3V instead of 5V, controlled with a jumper setting on the board, so make sure the jumpers are in the correct place.

Power from a Raspberry Pi

If you're going to buy a Raspberry Pi for the projects starting in Chapter 8, you're in luck; it has the side benefit of doubling as a 5-volt power supply! The GPIO pins on the Pi have various functions, but for this purpose, all you need to know is that pin 6 is ground and pin 2

supplies 5V. You can connect those pins to your breadboard as your power supply. See Figure 13-11 on page 312 for a GPIO pin diagram. There's no need to even install any Raspberry Pi software, because as soon as the Pi powers on, the 5-volt pin will turn on. Just connect your Pi to power. As a bonus, pin 1 can supply 3.3V if needed. To be clear, if you do this, you aren't using any of the computing power of the Pi; it is just acting as a 5-volt power supply. There is a limit to how much current the Raspberry Pi can supply. The power adapter you use for the Pi will have a maximum current rating, and the Pi itself will draw some current, maybe 300mA when idle. This probably goes without saying, but if you go with this option, be careful that you connect your circuit properly; you don't want to accidentally damage your Raspberry Pi!

Figure B-5 shows a Raspberry Pi being used as a power supply.

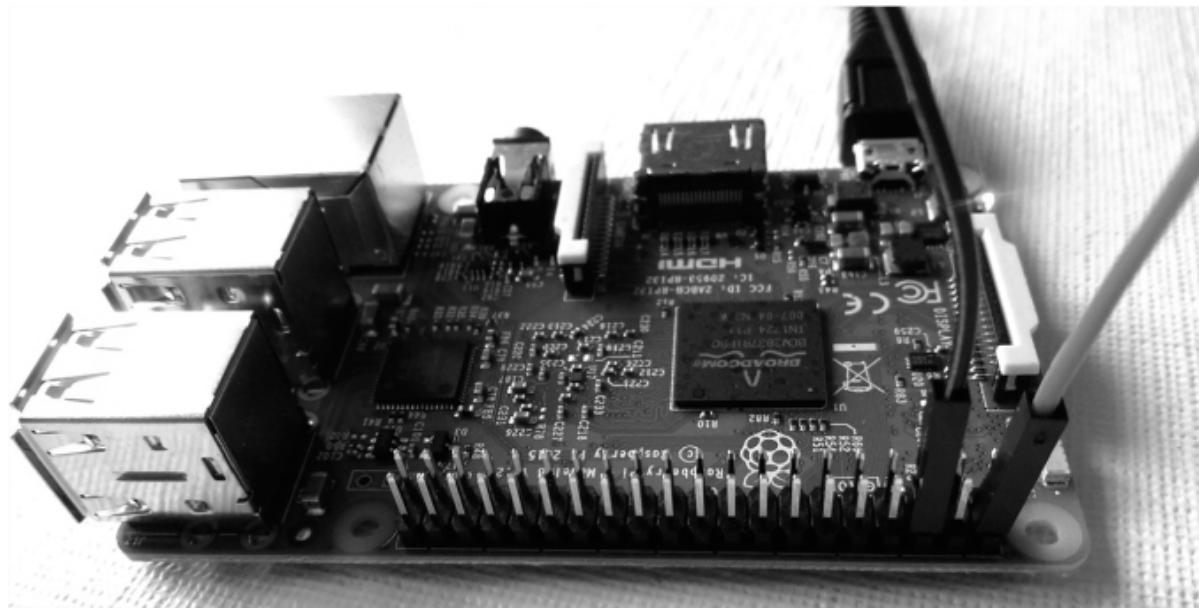


Figure B-5: Using a Raspberry Pi as a power supply

AA Batteries

You can also use AA batteries to power a digital circuit. A single AA battery supplies 1.5V, so three AA batteries can be connected in series to provide 4.5V. Although this voltage is less than the recommended voltage for 7400 series components, it should work for the circuits in this book, although your results may vary. You can buy a battery holder

for three AA batteries and connect its output wires to your breadboard, as shown in Figure B-6.



Figure B-6: Using three AA batteries to power a circuit on a breadboard

Troubleshooting Circuits

Sometimes you build a circuit, expecting things to work a certain way, but the result is something else entirely. Maybe the circuit doesn't appear to do anything, or maybe it behaves in a way you did not intend. Don't worry, this happens to everyone who builds circuits! It's easy to make a wiring mistake or to have a loose connection that throws everything off. Troubleshooting and diagnosing problems in a circuit is

a valuable skill that can actually help you expand your understanding of how things work. Here I share some troubleshooting approaches that I use when my circuits don't work as expected.

If any component in your circuit is too hot to touch, immediately disconnect the circuit from its power source. Wiring mistakes can lead to a component overheating. This often causes damage to the component if it's left connected for more than a few seconds.

Your primary tool for circuit troubleshooting is a multimeter. With a multimeter, you can easily check the voltage at various points around a circuit. Ask yourself, "What do I expect the voltage to be at this point or at that point in my circuit?" For 5-volt digital circuits, the expected voltages are usually going to be approximately 0V or approximately 5V. If your meter shows an unexpected voltage anywhere in the circuit, ask yourself, "What things could influence this voltage?" and check those things.

For digital circuits, I usually take a "work backward" approach, starting with the component that is misbehaving. Confirm that its output voltage is wrong, and then check its inputs. Is one of those inputs also an unexpected voltage? If so, move back to the component that feeds that input, and check its output. Repeat until you find the source of the problem.

When checking voltages, I've found that the simplest approach is to connect your black/negative/COM lead to a ground point in your circuit and leave it there. If there isn't an obvious place to connect the lead to ground, just add a jumper wire to a ground point on the breadboard, then use an alligator clip to connect that jumper wire to your COM lead. With your COM lead anchored to ground, you can easily use your positive lead, usually colored red, to poke around at various points in the circuit and check voltages relative to ground.

The other thing I regularly check with a multimeter during troubleshooting is resistance. Sometimes I know the expected resistance between two points, and I want to verify that resistance value. If there's more than one path connecting the points you are measuring, be sure

you know the expected resistance so you can correctly interpret your measurement.

Usually I'm checking resistance to simply ensure that two points are connected, in which case I expect a resistance of approximately 0Ω . Conversely, sometimes I want to ensure that two points are *not* connected, and then I look for a very high resistance, an open circuit. Some meters also include a *continuity check* feature, where the meter emits an audible tone if two points are connected. If you are just checking for connectivity, sometimes this is preferable to checking resistance.

Some specific things to verify when troubleshooting:

Breadboard power Does your breadboard have the appropriate voltage along the long power columns? The positive voltage column should equal the voltage from your source (say, a 9-volt battery or a 5-volt supply). Be sure to check both sides of the breadboard if both sides are in use.

Breadboard connections Verify that your wiring on the breadboard is sound. Are wires fully inserted, or do you have loose connections? Double-check the alignment of connections on the breadboard; is your wire in the right row? Is anything extra connected in the row you are checking?

Resistors Are your resistors the correct values? If needed, remove each one from the circuit and verify with a multimeter.

LEDs Are your LEDs oriented properly? The shorter lead should be connected closer to ground.

Capacitors If your capacitor is polarized, ensure you have the positive and negative leads properly oriented. Also check the capacitance value.

Integrated circuits Are your ICs properly connected to ground and to positive voltage? Is the chip fully seated in the breadboard, placed across the gap in the center? Check that your IC is aligned

correctly by looking for the notch. Are you using the right part number?

Digital input switches/buttons When using pull-down resistors, is one side of the switch connected to positive voltage, and the other side connected through a pull-down resistor to ground? Is the digital input pin on the related chip connected to the same side of the switch as the pull-down resistor?

Raspberry Pi

The Raspberry Pi is a tiny, inexpensive computer. It was developed to promote teaching of computer science, and it has gained a following among technology enthusiasts. It's our computer of choice for this book, so here we cover the basics of setting up and using a Raspberry Pi.

Why Raspberry Pi

Before we go into the details of how to configure your Raspberry Pi, I'd like to explain why I chose the Raspberry Pi for this book. Some of the projects require a computer of some sort to interact with. Now, you may be saying to yourself, "I already own a computer; why do I need another?" Yes, since you are reading a book on computing you probably already own a computer, or maybe several! However, not everyone owns the same kind of computer, and some types of computing devices are better suited than others for education. Additionally, some of the projects in this book deal with low-level details of computing, so everyone following along needs the same kind of device.

The Raspberry Pi was a natural choice since it's inexpensive (about \$35) and designed with computer education in mind. My goal isn't for you to switch to the Raspberry Pi as your primary computer or to make you a Raspberry Pi expert. Instead, we use the Raspberry Pi to learn core concepts that you can then apply to any computing device. The Raspberry Pi uses an ARM processor, and we'll be running *Raspberry Pi*

OS (previously called *Raspbian*) on it, a version of Linux optimized for the Raspberry Pi.

Parts Needed

First things first, you are going to need to obtain a Raspberry Pi and some accessories. Here's what you need:

- **Raspberry Pi.** These are usually about \$35 and can be purchased online. At the time of this writing, the latest model is the Raspberry Pi 4 Model B, and the exercises in this book were tested with this version and with the Raspberry Pi 3 Model B+. If a newer model is released, it likely will be acceptable as well, given the Raspberry Pi's track record of backward compatibility. The Raspberry Pi 4 Model B is available in multiple memory configurations (1GB, 2GB, 4GB, and 8GB)—any of them are fine for this book.
- **USB-C power supply (only for Raspberry Pi 4).** The Raspberry Pi 4 uses a USB-C power supply. The power supply needs to provide 5V and at least 3A. Certain USB-C power supplies are incompatible with some Raspberry Pi 4 devices, so I recommend you purchase a USB-C power supply specifically designed for the Raspberry Pi 4.
- **Micro-USB power supply (only for Raspberry Pi 3).** Unlike the Raspberry Pi 4, the Raspberry Pi 3 is powered by a micro-USB power adapter, like the ones that many smartphones use. If you have a smartphone charger already, it might work with the Pi. Just be sure that the connector is micro-USB. The standard voltage output from such chargers is 5V, but the maximum amount of current they supply varies. For the Raspberry Pi 3, it's recommended that you use a power supply that can provide at least 2.5A. The current requirements vary depending on what you have connected to the Pi. So check your smartphone charger to see how much current it can supply; you may need to purchase a micro-USB power supply designed for the Pi.

- **MicroSD card, 8GB or larger.** The Raspberry Pi doesn't come with any storage, so you need to add it yourself via a microSD card. These cards are commonly used in smartphones and cameras, so you may have an extra one lying around already. The process of installing Raspberry Pi OS will erase existing data, so be sure to back up anything you have stored on your microSD card.
- **USB keyboard and USB mouse.** Any standard USB keyboard and USB mouse will do.
- **Television or monitor that supports HDMI.** All modern televisions and many computer monitors support HDMI connections.
- **HDMI cable.** The Raspberry Pi 3 uses a standard, full-sized HDMI cable, but the Raspberry Pi 4 has a micro-HDMI port. Assuming your display device accepts full-sized HDMI input, this means that for a Raspberry Pi 4 you need a micro-HDMI to HDMI cable or adapter.
- **Optional: Raspberry Pi case.** This isn't required, but it is nice to have. Note that the Raspberry Pi 3 and Raspberry Pi 4 have different physical layouts, so they need differently shaped cases.

See the “Shopping” section of “Buying Electronic Components” earlier in this appendix for help getting these parts.

Setting Up a Raspberry Pi

The Raspberry Pi website (<https://www.raspberrypi.org>) contains detailed setup guides that walk through setting up a Raspberry Pi. I don't cover all the details here since the online documentation already exists, and it changes over time. Instead let me give you a brief overview of the steps required.

You have several options for installing Raspberry Pi OS on your Raspberry Pi. If you have a computer with a microSD card reader/writer, the simplest option is to use the *Raspberry Pi Imager*. Here's how to use this tool to get going quickly with your Raspberry Pi:

1. Insert your microSD card into your computer.
2. Download the Raspberry Pi Imager from <https://www.raspberrypi.org/downloads>.
3. Install and run the Raspberry Pi Imager on your computer.
4. Choose the operating system: Raspberry Pi OS (32-bit).
5. Choose the SD card you want to use.
6. Click “Write” and Raspberry Pi OS will be copied to your microSD card.
7. Remove the microSD card from your computer.
8. Insert the microSD card into the Raspberry Pi.
9. Connect the Raspberry Pi to a USB keyboard, USB mouse, and monitor or TV using HDMI, and finally to the power supply.
10. The Raspberry Pi should boot into Raspberry Pi OS.

Another good option for installing Raspberry Pi OS is to use the Raspberry Pi *New Out of Box Software (NOOBS)*. To use NOOBS, download it from <https://www.raspberrypi.org/downloads> and copy it to a blank microSD card. If you don’t have another computer available to do this, you can buy a microSD card with a copy of NOOBS preloaded. In either case, once you have NOOBS on the microSD card, insert the card into your Raspberry Pi and power it on. Follow the on-screen instructions to install Raspberry Pi OS.

NOTE

At the time of this writing, a 64-bit version of Raspberry Pi OS is available as a beta release. However, the projects in this book were tested with the 32-bit Raspberry Pi OS, so I recommend that you stick with the 32-bit version for the projects.

Using Raspberry Pi OS

Once your Raspberry Pi is set up, I recommend you take a little time to familiarize yourself with the user interface of Raspberry Pi OS. If you’ve used a Mac or a Windows PC before, the Raspberry Pi OS desktop

environment should be somewhat familiar. You can open applications in windows, move those windows around, close them, and so forth.

That said, most of the projects in the book won't require you to use any of the Pi's graphical applications. Nearly everything can be done from a terminal, and most projects require at least some use of the terminal, so let's take a minute to get familiar with it. From the Raspberry Pi OS desktop, you can open a terminal window by clicking **Raspberry** (icon in the upper-left corner) ▶ **Accessories** ▶ **Terminal** as shown in Figure B-7.

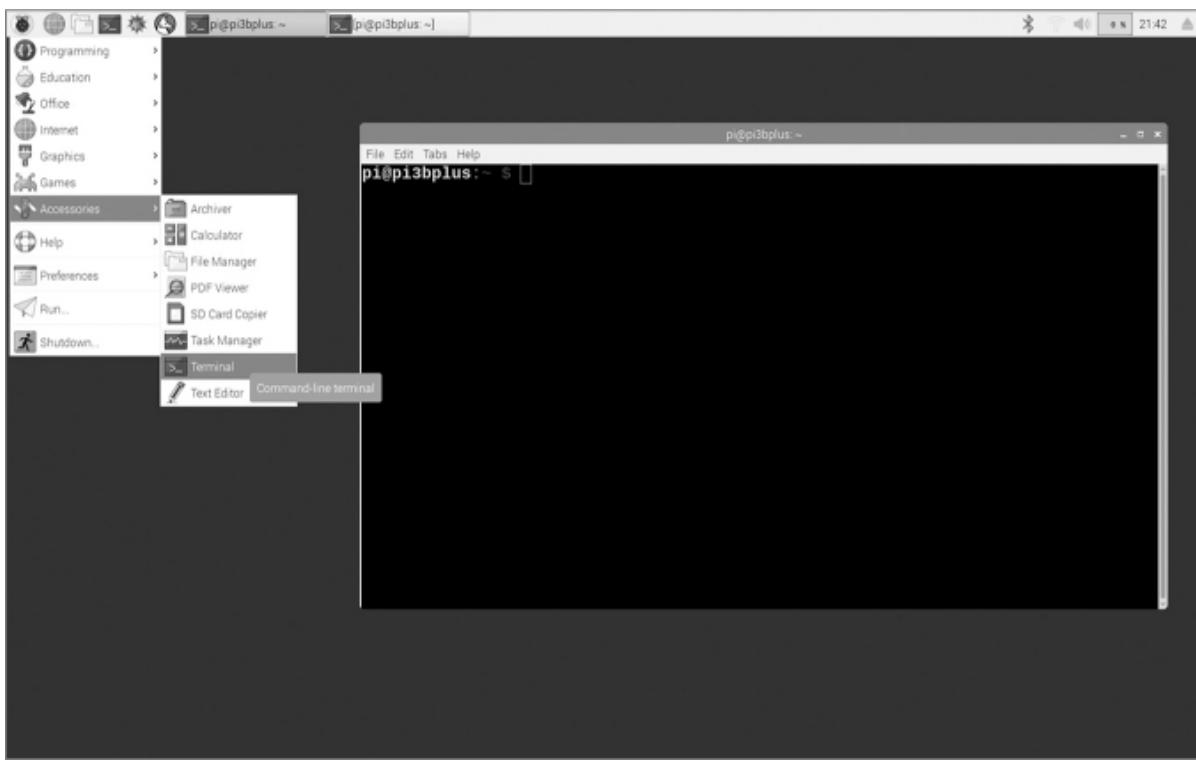


Figure B-7: Opening the Raspberry Pi terminal

The terminal is a *command line interface (CLI)*, where everything you do is accomplished by typing in commands. Raspberry Pi OS, like all versions of Linux, has excellent support for the CLI, and you can do just about anything from a terminal if you know the right commands. By default, the terminal on Raspberry Pi OS runs a shell called *bash*. A *shell* is a user interface for an operating system, which can be either graphical

(like the desktop) or command line based. The initial text in the bash command line should look something like this:

```
pi@raspberrypi:~ $
```

Let's examine each part of that text string:

pi This is the username of the currently logged-on user. The default user's name is "pi."

raspberrypi Separated from the username by an @ sign, this is the name of the computer.

~ This indicates the current directory (folder) you are working in. The ~ character has a special meaning; it refers to the current user's home directory.

\$ The dollar sign is the CLI prompt, an indicator that you can type your commands here.

In this book, when I list commands that should be typed in a terminal, I prefix the line with a \$ prompt. As an example, this command lists the files in the current directory:

```
$ ls
```

To run a command, you don't need to type the dollar sign; just type the text following it, and then press ENTER. If you want to run a command that you previously entered, you can press the up arrow on the keyboard to cycle through previously issued commands.

If you prefer to work in a terminal, you can set up your Raspberry Pi to boot directly to a command line rather than to the desktop:

Raspberry ▶ Preferences ▶ Raspberry Pi Configuration ▶ System tab ▶ Boot ▶ To CLI. Once you've made this configuration change, the next start of the system goes directly to the CLI rather than the desktop. While in a CLI-only environment, if you want to start the desktop environment, just run the following command:

```
$ startx
```

As a terminal user, another option is to control your Raspberry Pi from a different computer or even from a phone by using *Secure Shell* (*SSH*) over the network. The end result of this approach is that your Pi can run anywhere on your network, even without a monitor or keyboard attached, and you can use the keyboard and monitor of another device to control it. To pull this off, you must enable SSH on the Pi (**Raspberry ▶ Preferences ▶ Raspberry Pi Configuration ▶ Interfaces tab ▶ SSH ▶ Enable**), and then run an SSH client application on a second device. I won't include detailed steps for setting this up, but there are plenty of guides online for how to do this.

When you are finished working with your Pi for a while, you'll want to shut it down gracefully to avoid data corruption, rather than just turning it off. From the desktop, you can shut down the system with **Raspberry ▶ Shutdown... ▶ Shutdown**. Or from a terminal, you can use this command to halt the system:

```
$ sudo shutdown -h now
```

You'll know the system has completely shut down when the attached monitor no longer displays anything and the activity light on the Raspberry Pi board stops blinking. You can then unplug your Pi.

Working with Files and Folders

The projects in this book regularly direct you to create or edit text files and then run some terminal commands on them. Let's talk about working with files and folders in Raspberry Pi OS, both from a command line and from the graphical desktop. Operating systems organize the data on a storage device, like the microSD card in your Raspberry Pi, with a filesystem. A *file* is a container of data, and a *folder* (also known as a *directory*) is a container of files or other folders. The filesystem's structure is a *hierarchy*, a tree of folders. On Linux systems, the root of that hierarchy is represented with */*. The root is the topmost folder—all other folders and files are “under” the root.

A folder directly under the root folder would be represented like this: `/<filename>`. A text file in that folder would look something like this: `/<filename>/<filename>.txt`. Note the `.txt` file extension, the last part of the filename. It is convention to end a filename with a period followed by a few characters to indicate what kind of data is in the file. In the case of text files, “`txt`” is often used. The use of file extensions is not a requirement, but it is a common practice, helpful for keeping your data organized.

Every user in Raspberry Pi OS has a home folder to work in. The default user in Raspberry Pi OS is named `pi`, and the `pi` user’s home folder is located in `/home/pi`. While you are logged on as the `pi` user, that same home folder can also be referred to with the `~` character. Let’s say you create a folder in your home folder called `pizza`. Its full path would be `/home/pi/pizza`, or when logged in as `pi`, you can refer to it as `~/pizza`. Let’s try creating a `pizza` folder from a terminal window, using the `mkdir` command, short for “make directory.” Don’t forget to press ENTER after typing the command.

```
$ mkdir pizza
```

From a terminal, you can see your newly created folder by using the `ls` command:

```
$ ls
```

When you type `ls` and press ENTER, you should see the `pizza` folder alongside a set of other folders that were already present in your home folder, such as `Desktop`, `Downloads`, and `Pictures`.

The terminal isn’t the only way to view the files in a folder. You can also use the File Manager application, which you can launch from **Raspberry ▶ Accessories ▶ File Manager**. As shown in Figure B-8, the File Manager application opens with a default view of your home directory.

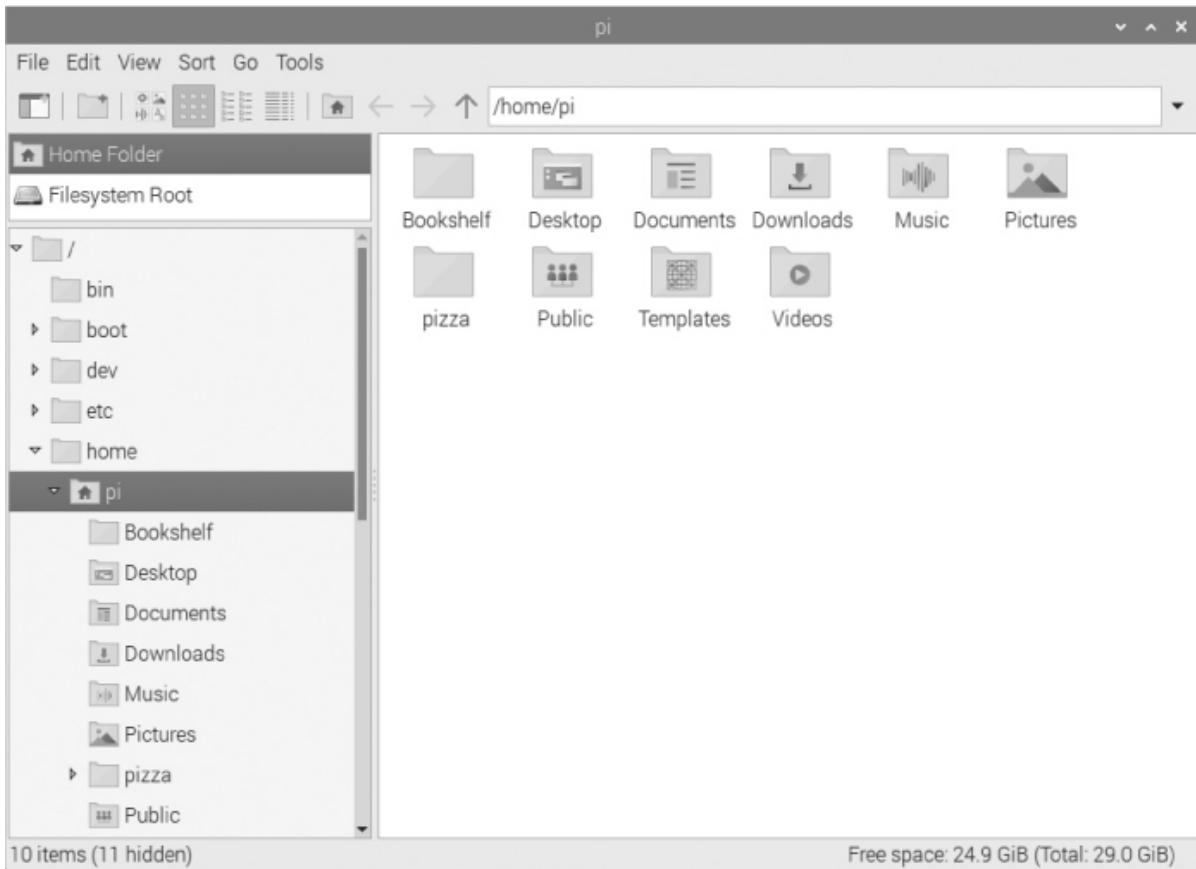


Figure B-8: Raspberry Pi OS File Manager

The left-hand side of File Manager shows the full filesystem hierarchy of folders, with the currently selected folder highlighted—*pi* in this case. The address bar at the top showing */home/pi* indicates the current folder. Now, try double-clicking the *pizza* folder; it should be empty. Let's jump back to the terminal window and create some files in this folder. First, let's change folders with the **cd** command (for change directory), so that our current folder is the *pizza* folder. Then let's create two empty files using the **touch** command. Finally, we'll list the contents of the directory with **ls**, expecting to see the two new filenames listed.

```
$ cd pizza
$ touch cheese.txt
$ touch crust.txt
$ ls
```

Note that when you changed to the *pizza* folder, your bash prompt should have changed as well. It should now include `~/pizza` before the \$, indicating the current folder. Now look at the File Manager application window; it should also show the two new files under the *pizza* folder, as shown in Figure B-9.

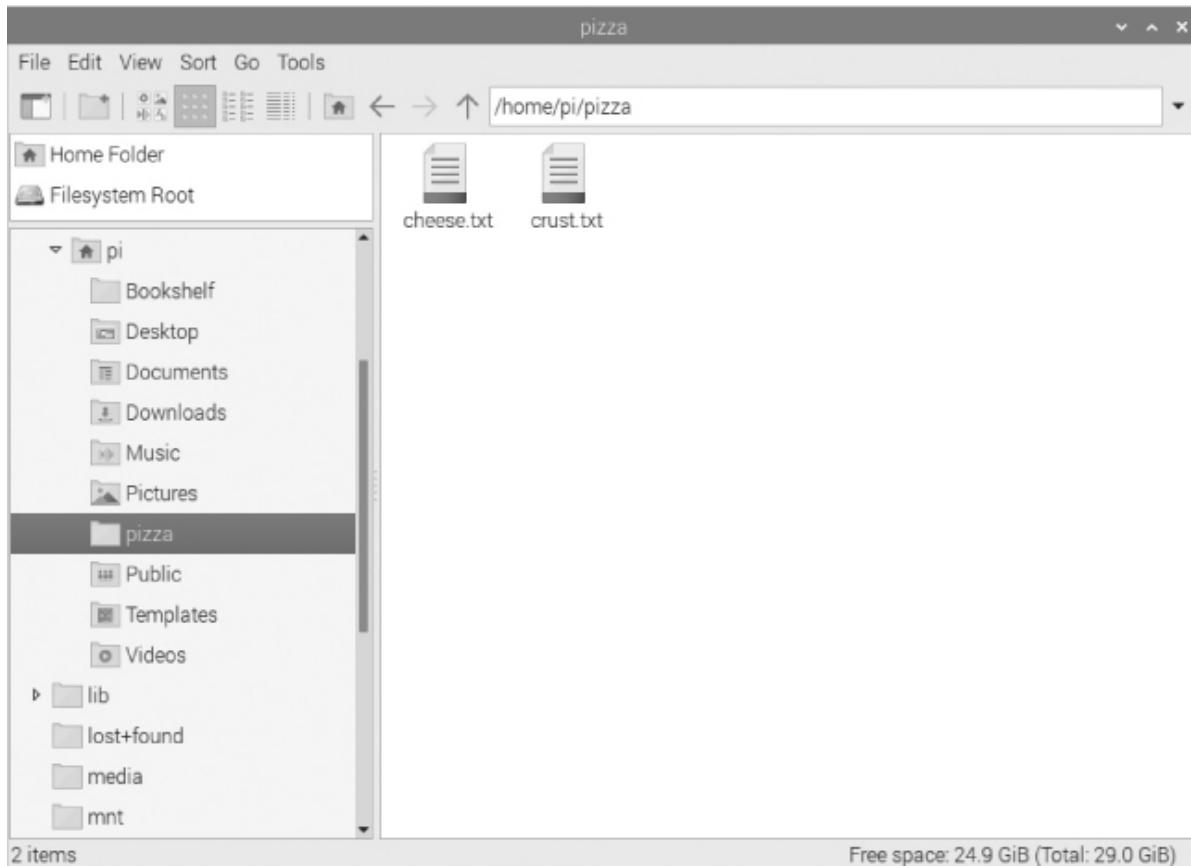


Figure B-9: Raspberry Pi OS File Manager: files in the *pizza* folder

Now we have two empty files in the *pizza* folder. Let's add text content to the files. First, we'll edit *cheese.txt* using a command line text editor called `nano`.

```
$ nano cheese.txt
```

With the `nano` editor window open in the terminal, you can type text that will be saved to *cheese.txt*. Keep in mind that `nano` is a command line application—you can't use the mouse. You need to use the arrow keys to move the cursor. Try typing some text, as shown in Figure B-10.



Figure B-10: Using nano to edit cheese.txt

After typing in some text in `nano`, press CTRL-X to exit `nano`. The editor prompts you to save your work (“Save modified buffer?”). This may seem like an odd question, but don’t let the “buffer” term throw you off—`nano` is just asking if you want to save the text you entered to the file. Press `y`, then hit ENTER to accept the suggested filename (`cheese.txt`).

I often use `nano` since it works from a terminal, but you may prefer to edit your text files using a graphical text editor. The Raspberry Pi OS desktop environment includes a handy text editor which you can launch from **Raspberry ▶ Accessories ▶ Text Editor**. At the time of this writing, this opens an editor called Mousepad. Let’s try modifying `cheese.txt` further in this text editor. First, we need to open the file by performing the following within the text editor: **File ▶ Open ▶ Home ▶ pizza ▶ cheese.txt ▶ Open (button)**, as shown in Figure B-11.

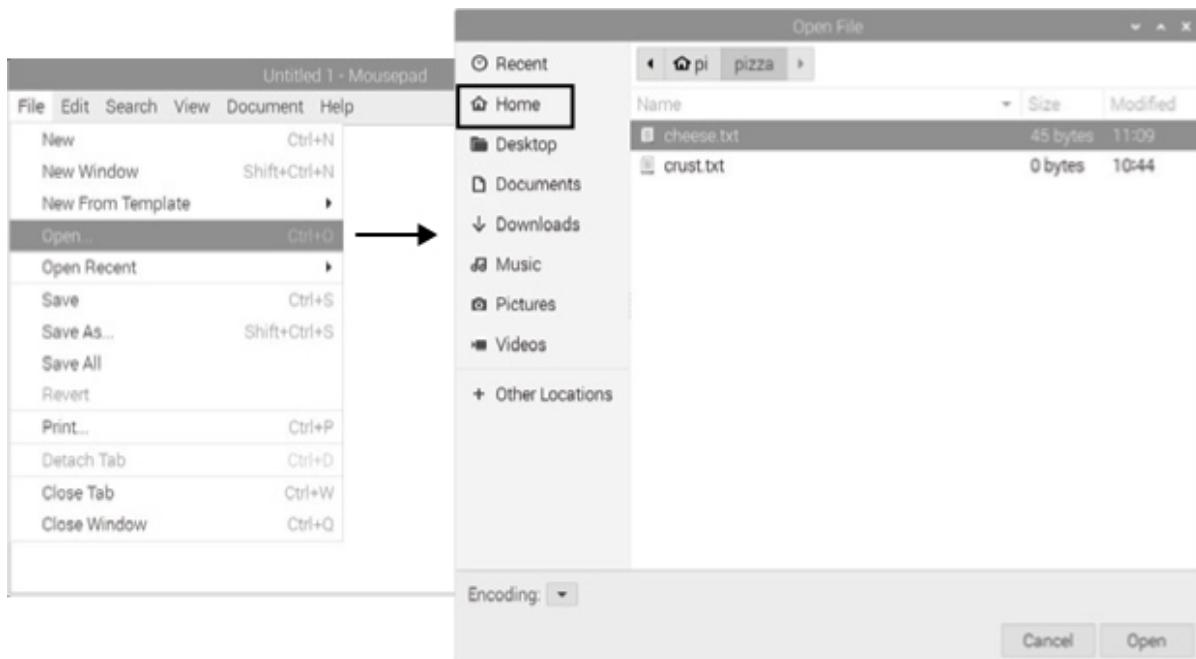


Figure B-11: Opening cheese.txt in the text editor

Once you have the file open in the text editor, you should see the text you typed earlier. You can edit the text as you wish. Then save your changes with **File ▶ Save**.

In addition to editing existing files, you can also use the text editor to create new files. Just launch a new editor window (**Raspberry ▶ Accessories ▶ Text Editor**) and click **File ▶ Save**. This will prompt you to save your file in the folder of your choice, with a name of your choosing. You can edit the text of your new file and save your changes as needed by using **File ▶ Save** again. Or if you want to save an existing file with a new name, choose **File ▶ Save As...** instead.

If you prefer to use `nano` to create a new file, from a terminal window first change to the folder where you want to save your file (if needed), and then type `nano filename`, as shown in this example:

```
$ nano new-file.txt
```

Type your text, and when you exit `nano`, you'll be prompted to save this new file.

We've just covered ways to view, edit, and create text files in Raspberry Pi OS. If you want to remain in a terminal, `nano` is a good choice. If you prefer the desktop, then the text editor Mousepad should meet your needs. There are also other editors included with Raspberry Pi OS. Geany is a programmer's text editor, and Thonny Python IDE is tailored for Python programming. Both are found under **Raspberry ▶ Programming**. In this book's projects, I'll leave it up to you to decide which text editor you use.

You may also wish to manage your files and folders in other ways—move files around, delete files, and so forth. You can do all of this from File Manager, or you can do it from a terminal window. Here are some commands you can use from a bash prompt to get you started:

`cd folder` Change the current directory (folder).

`mkdir folder` Make a directory.

`rm file` Delete a file.

`rm -rf folder` Delete a folder and its contents, including its subfolders.

`mv file file2` Rename a file.

`mv file folder/` Move a file from one location to another.

`cp file folder/` Copy a file from one location to another.

Index

Numbers

- 8086 processor, 124
- 555 integrated circuit, 110
- 4-bit adder, 80–81
- 9-volt battery, 40
- 7400 series, 63
 - part numbers, 334
- 7408 integrated circuit, 65, 68
- 74191 integrated circuit, 103
- 7473 integrated circuit, 111
- 7432 integrated circuit, 65, 68
- 7402 integrated circuit, 65
- 6 degrees of freedom (6DoF), 307
- 3-bit counter, 101–103
- 3 degrees of freedom (3DoF), 307
- 2N2222 transistor, 66
- 0b prefix, 7
- 0x prefix, 12

A

- ABI (application binary interface), 214
- AC (alternating current), 35
- active low, 112
- additive color model, 23
- address
 - broadcast, 242
 - bus, 134

IP, 239
 private, 249
 static, 247
memory, 119
space
 32-bit, 205
 64-bit, 206
 physical, 132
 virtual, 204
of variable, 162
web, 262

Advanced Micro Devices, Inc. (AMD), 124

Alibaba Cloud, 301

alligator clips, 40

allocate memory, 166

alphanumeric, 18

altcoins, 302

alternating current (AC), 35

ALU (arithmetic logic unit), 125

Amazon

- Appstore, 294
- Elastic Compute Cloud (EC2), 300
- Web Services, 301
 - Elastic Beanstalk, 300
 - Lambda, 300

AMD64, 124

AMD (Advanced Micro Devices, Inc.), 124

American Standard Code for Information Interchange (ASCII), 19

amperes, 32

amps, 32

analog, 3

 and digital, 2

 signal, 5

analogy, 3

AND

 7408 integrated circuit, 65

 bitwise operator, 169

 Boolean operator, 171

 implemented with switches, 56

 implemented with transistors, 60

 logical operator, 25

 symbol, 61

 truth table, 26, 61

Android, 197, 293

 Platform APIs, 208, 211

Angular, 294

anode, 51

API. *See* application programming interface (API)

app, 292

 native, 292–294

 store, 292

 web, 292, 294–295

Apple

 App Store, 292

 iPhone, 293

 macOS, 197

 Safari, 277

application, 138, 292

application binary interface (ABI), 214

application layer, 245

 foundational protocols, 247–252

application programming interface (API), 207–209
and system calls, 211
defined, 207
implementation, 207

App Store, 292

AR (augmented reality), 307

architecture, instruction set, 123

A record (DNS), 251

arithmetic logic unit (ALU), 125

ARM, 125, 181
instructions
decoding, 139
examples, 142, 145

`arp-scan`, 254

ASCII (American Standard Code for Information Interchange), 19
in memory, 121

`as` (GNU assembler), 148

assembler, 140, 147
directives, 146
language. *See* assembly language

assembly language, 140–144, 160
directives, 146
labels, 146

assign, a variable, 162

asymmetric encryption, 266

Atari 2600, 194

atoms, 32

augmented reality (AR), 307

Azure. *See* Microsoft: Azure

B

base 2, 6
base 10, 5
base 16, 11
base, of a transistor, 57
Bash shell, 196, 202
battery, 33
 9-volt, 40
Bell Labs, 197
Berners-Lee, Tim, 277
big-endian, 156
binaries. *See* executable file
binary, 6
 addition, 74–76
 digital circuits, 54
 logic, 25
bipolar junction transistor (BJT), 57
bit, 8
Bitcoin
 address, 304
 blockchain, 302
 fee, 306
 genesis block, 306
 mining, 305
 network, 303
 node, 303
 transaction, 304
 wallet, 303
bitcoin (BTC), 302
bitmap, 24

bitness, 124
bitwise operators, 169
BJT (bipolar junction transistor), 57
black box, 60
Blink, 278
blockchain, 302
Boolean
 algebra, 30
 logic, 30
 operators, 170
 value, 170
 variable, 171
Boole, George, 30
bounce, in a switch, 110
branching, 143
breadboard, 40
breakpoint, 150, 192
broadcast address, 242
BTC (bitcoin), 302
building software, 160
bus
 address bus, 134
 communication, 134–135
 control bus, 134
 data bus, 134
buying electronic components, 333–336
byte, 8
byte-addressable, 119
bytecode, 180

C

C++ programming language, 161
canonical name (CNAME) record (DNS), 251
capacitor, 119
 electrolytic, 107
 polarized, 107
 symbols, 107
cartridge, 194
Cascading Style Sheets (CSS), 268, 271–273
cathode, 51
Celeron, 124
central processing unit (CPU). *See* processor
CERN httpd, 277
characters, 18
charge, 32
chip. *See* integrated circuit (IC)
Chrome. *See* Google: Chrome
Chromium, 279
 DevTools, 283
CIDR (Classless Inter-Domain Routing), 241
circuit, 36
 diagrams, 35–38
 digital, 54
 integrated. *See* integrated circuit (IC)
 open, 36
 short, 37
 troubleshooting, 340–341
class, 177
 variable, 178
CLI (command line interface), 196

client, 235
clock signal, 98, 127
 manual, 110
close function, 224
cloud. *See* internet
cloud computing, 298–301
 categories, 299–301
 consumer, 299
 provider, 299
CLR (Common Language Runtime), 180
CNAME (canonical name) record (DNS), 251
Cocoa, 208
 Touch, 208
code
 machine, 138–144
 portability, 208
 source, 138, 179
 trusted, 200
 untrusted, 200
collector, of a transistor, 57
colors, 21
 additive model, 23
 subtractive model, 23
combinational logic, 62, 92
command line interface (CLI), 196
comment, in programming languages, 161
common collector, 58
Common Intermediate Language, 180
Common Language Runtime (CLR), 180
communications protocol, 234

comparison operators, 171
compiled language, 179
compiler, 160
computer, 2
 hardware overview, 117
 network, 234
condition, in ARM instruction, 139
conductor, 33
container, 296
control bus, 134
control flow. *See* program flow
control unit, 125
convenience variable, 149
copper wire, 33
core
 logical, 203
 physical, 203
coulomb, 32
counter, 101–103
C programming language, 161
 in operating systems, 208
C# programming language, 180
CPU cache. *See* processor cache
CPU (central processing unit). *See* processor
CPU core, 128
 logical, 203
 physical, 203
CPython, 180
`createFileA` function, 208
`createFileW` function, 212

cross-coupled, 93
cross-platform framework, 293
cryptocurrency, 302
cryptographic key, 266
CSS (Cascading Style Sheets), 268, 271–273
curl, 285
current, 32
 measuring, 49
Cutler, Dave, 198
cycle, 98

D

daemons, 196, 216
dark web, 302
Darwin (operating system), 198
data, 2
 bus, 134
 representing digitally, 18
datagram, 243
DC (direct current), 35
debounce circuit, 110
debugger, 148, 184
DEC (Digital Equipment Corporation), 198
decimal, 5
declare, a variable, 162
decode, an instruction, 18, 126
decrement operator, 168
decryption, 266, 303
deep web, 302
default gateway, 247, 256

destination register, 139
device driver, 196, 214
`df` tool, 230
DHCP. *See* Dynamic Host Configuration Protocol (DHCP)
digit, 8
digital, 4
 and analog, 2
certificate, 267
circuits, 54
 powering, 336–339
colors and images, 21
 photographs, 22
Digital Equipment Corporation (DEC), 198
diode, 42
 light-emitting. *See* LED (light-emitting diode)
diode-transistor logic (DTL), 63
DIP (dual in-line package), 63
direct current (DC), 35
directory, of filesystem, 215
disassembler, 141
disassembly, 149
discrete component, 63
DLL (dynamic link library), 213
`dmesg`, 222
Docker, 296
Document Object Model (DOM), 273
Domain Name System (DNS)
 hierarchy, 251
 record, 251
 root domain, 251

server, 250
top-level domain (TLD), 252
don't repeat yourself (DRY), 175
`double`, 167
DRAM (dynamic random access memory), 119
driver. *See* device driver
Dropbox, 301
DTL (diode–transistor logic), 63
dual in-line package (DIP), 63
dynamic
 link library (DLL), 213
 ports, 244
 random access memory (DRAM), 119
 website, 280
Dynamic Host Configuration Protocol (DHCP)
 broadcast, 248
 lease, 248
 server, 248

E

EC2 (Amazon Elastic Compute Cloud), 300
ECMAScript, 275, 294
edge-triggered
 negative, 99
 positive, 99
electric
 charge, 32
 current, 32
 potential, 33
electrical
 circuit. *See* circuit

conductor, 33
resistance, 32
terms, 32–35
electrolytic capacitor, 107
electronic switch, 57
electrons, 32
electrostatic-sensitive device, 66
ELF (executable and linkable format), 156
`elif` statement, 172
`else` statement, 172
emitter, of a transistor, 57
emulation, 297
emulator, 297
encapsulation, 60, 175, 210, 235
encoding, 18
encryption, 266, 303
endianness, 156
endpoint, 245
energy, 33
ephemeral ports, 244
Ethernet, 238
exclusive OR. *See* XOR
executable and linkable format (ELF), 156
executable file, 147, 160, 179
execute, an instruction, 126
executive, 200
ext2/ext3/ext4 filesystems, 215
Extensible Markup Language (XML), 275

F

FaaS (Function as a Service), 300
factorial, 180
 in C, 180–183
 in machine code, 141–144
farads, 97
FAT (File Allocation Table), 215
fetch, an instruction, 126
FET (field-effect transistor), 57
field, 178
field-effect transistor (FET), 57
file
 creating, 207
 defined, 215
 descriptor, 226
File Allocation Table (FAT), 215
filesystem, 215
 directory, 215
 ext2/ext3/ext4, 215
 FAT, 215
 folder, 215
 format, 215
 NTFS, 215
File Transfer Protocol (FTP), 245
fixed-purpose device, 118
flip-flop, 119
 JK
 compared with SR latch, 99
 symbol, 100
 table, 100
 T, 100
 symbol, 101

table, 101

`float`, 167

floating-point arithmetic, 167

floppy disk, 216

Flutter, 293

folder, of filesystem, 215

`fopen` function, 208

`for` loop, 173

format, filesystem, 215

forward voltage, 43

FQDN (fully qualified domain name), 250

frame, 238

FreeBSD, 198

free memory, 166

frequency, 98

FTP (File Transfer Protocol), 245

full adder, 80

- defined, 78
- symbol, 78
- truth table, 79

fully qualified domain name (FQDN), 250

function, 177

- body, 175
- call, 176
- defined, 174
- definition, 175
- parameters, 174
- return value, 174

Function as a Service (FaaS), 300

G

- garbage collection, 166
- `gcc` (GNU C Compiler), 179, 184, 221
- `gdb` (GNU debugger), 148
 - `break`, 192
 - `disassemble`, 192
 - `info proc mappings`, 187
 - `info sharedlibrary`, 229
 - `quit`, 229
 - `run`, 185
 - `start`, 229
 - `x`, 185
- gdi32.dll*, 213
- Gecko, 279
- general-purpose input/output (GPIO), 311
- `get-oui`, 255
- `glibc`, 212, 227
- GND. *See* ground
- GNU C Compiler (`gcc`), 179, 184
- GNU (GNU’s Not Unix), 197
 - assembler (`as`), 148
 - C Compiler (`gcc`), 221
 - C Library (`glibc`), 212, 227
 - debugger (`gdb`), 148
 - linker (`ld`), 148
- Google
 - App Engine, 300
 - Cardboard, 307
 - Chrome, 277
 - address bar, 263

- DevTools, 283
- Cloud Functions, 300
- Cloud Platform, 301
- Compute Engine, 300
- Daydream, 308
- G Suite, 301
- Play Store, 292, 294
 - search, 268
- GPIO (general-purpose input/output), 311
- GPIO Zero, 314
- graphical user interface (GUI), 196
- grayscale, 21
 - images, 22
- `grep`, 222
- ground, 37
 - in a digital circuit, 54
 - symbol, 37
- GUI (graphical user interface), 196

H

- half adder
 - defined, 76
 - symbol, 76
 - truth table, 77
- hard disk drive (HDD), 130, 215
- hardware, 118
- Hardware Abstraction Layer (HAL), 200
- heap memory, 165
 - allocate, 166
 - free, 166
- hertz (Hz), 98

hexadecimal, 11
`hexdump`, 155, 192
high definition display, 22
high-level programming
 languages, 160
 overview, 160
host, 235
 identifier, 240
 tool, 260
hostname, 250
HTC Vive, 308
HTML5, 271, 294
hub, 239
hyperlink, 264
hypertext, 264
HyperText Markup Language (HTML), 268
 element, 269
 Living Standard, 271
 tag, 269
HyperText Transfer Protocol (HTTP), 245, 266
 method, 265
 request, 264
 response, 264
 status code, 265
 verb, 265
HyperText Transfer Protocol Secure (HTTPS)
 client hello, 267
 server hello, 267
hyper-threading, 203
hypervisor, 296

Hz (hertz), 98

I

IA-32, 124

IA-64, 124

IaaS (Infrastructure as a Service), 299

IBM, 198

Cloud, 301

IC. *See* integrated circuit (IC)

IEEE

802.3, 238

802.11, 238

1541, 11

`ifconfig`, 254

`if` statement, 172

images, 21

bitmap, 24

compression, 24

grayscale, 22

JPEG, 24

PNG, 24

RGB, 23

immediate

bit, 139

value, 139

increment operator, 168

Infrastructure as a Service (IaaS), 299

`init` process, 201

input/output (I/O), 118, 131

device, 131

memory-mapped, 132

- port, 132
- port-mapped, 132
- instance
 - of a class, 177
 - variable, 178
- instruction pointer, 126, 150
- instruction register, 127
- instruction set architecture (ISA), 123
- `int`, 162
- integer
 - overflow, 87
 - math, 167
- integrated circuit (IC), 64
 - 555, 110
 - 7400, 65
 - 7400 series, 63
 - part numbers, 334
 - 7402, 65
 - 7408, 65
 - 7432, 65
 - 7473, 111
 - 74191, 103
 - full adder, 78
 - half adder, 76
 - pins, 62
- Intel 64, 124
- Intel Corporation, 124
- intermediate language, 180
- International System of Units (SI), 9
- internet
 - a trip through, 246–247

defined, 234
layer, 239–243
Protocol (IP), 235, 239
 Version 4 (IPv4), 240
 Version 6 (IPv6), 240
Internet of Things (IoT), 308
internet protocol suite, 237, 246
 application layer, 245
 foundational protocols, 247–252
 defined, 235
 internet layer, 239–243
 layers, 236
 link layer, 238
 model, 236
 transport layer, 243–245
interpreted language, 179
interpreter, 179
I/O (input/output). *See* input/output (I/O)
iOS, 197, 293
IoT (Internet of Things), 308
IP address, 239
 private, 249
 static, 247
`ipconfig`, 255
iPhone, 293
IP (Internet Protocol), 235, 239
`ip route`, 256
IPv4 (Internet Protocol Version 4), 240
IPv6 (Internet Protocol Version 6), 240
ISA (instruction set architecture), 123

Itanium, 124

J

Java programming language, 180

JavaScript, 268, 273–275

Object Notation (JSON), 276

JIT (just-in-time) compiler, 273

JK flip-flop

compared with SR latch, 99

symbol, 100

table, 100

Jobs, Steve, 293

joules, 33

JPEG, 24

JSON (JavaScript Object Notation), 276

just-in-time (JIT) compiler, 273

K

kernel, 195

address space, 205

mode, 199

module, 215

ring buffer, 222

kernel32.dll, 213

keyboard, 131

key pair, 266

kill tool, 222

Kirchhoff's voltage law, 38–39

`kthreadd`, 219

L

last-in first-out (LIFO), 164
latch, 92, 99
 SR, 94
 compared with JK flip-flop, 99
 implementation, 93
 symbol, 93
 table, 92
 using in a circuit, 95–98
`ld` (GNU Linker), 148
least significant bit, 74
LED (light-emitting diode). *See* light-emitting diode (LED)
Lenovo Mirage Solo, 308
libc.so.6, 212
library
 export, 212, 229
 import, 212, 229
 link, 212
 operating system, 196, 207, 213–214
 defined, 212
 standard, 177, 208
LIFO (last-in first-out), 164
light-emitting diode (LED), 42–44
 anode, 51
 cathode, 51
 forward voltage, 43
 symbol, 42
link
 layer, 238
 library, 212
 on the web, 264
linker, 147, 160

Linux, 197
 API, 208, 211
 distribution, 197
 system calls, 211
little-endian, 156
localhost, 285
logarithm, 120
logic
 binary, 25
 combinational, 62, 92
 gates, 59–60, 62
 7400 series, 63
 designing with, 61
 symbols, 61
 truth tables, 61
 universal, 95
programming, 172
 bitwise operators, 169
 Boolean operators, 170
 comparison operators, 171
sequential, 62, 91
with mechanical switches, 54–57
logical
 core, 203
 operator, 25
 AND, 25
 NAND, 29
 NOR, 29
 NOT, 29
 OR, 27
 XOR, 29

- statement, 25
- logic gates, 95
- loopback, 255
- looping, 173
- lowmem, 223
- `lsblk`, 230
- `lsmod`, 230

M

- machine
 - code, 138–144, 160
 - instruction, example, 139–140
 - language, 138
- MAC (media access control) address, 238
- macOS, 197
- mail exchanger (MX) record (DNS), 251
- main memory, 118–122
- malicious software, 217
- `malloc`, 166
- manual clock, 110
- math
 - 4-bit adder, 80–81
 - binary addition, 74–76
 - floating-point arithmetic, 167
 - full adder, 80
 - defined, 78
 - symbol, 78
 - truth table, 79
 - half adder
 - defined, 76
 - symbol, 76

- truth table, 77
- integer, 167
 - overflow, 87
- operators, 167
 - decrement, 168
 - increment, 168
- programming, 166–169
- Maxwell’s demon, 216
- mechanical switch, 54, 70
- media access control (MAC) address, 238
- memory, 92, 118
 - address, 119
 - physical, 204
 - virtual, 204
 - cell, 119
 - controller, 134
 - heap, 165
 - allocate, 166
 - free, 166
 - leak, 166
 - main, 118–122
 - paging, 206
 - stack, 164
 - overflow, 165
 - pointer, 164
 - virtual. *See* virtual memory
- memory-mapped I/O (MMIO), 132
- method
 - HTTP, 265
 - in object-oriented programming, 177
- metric system, 9

microprocessors, 118
microSD card, 230
Microsoft, 365
 Azure, 301
 App Service, 300
 Functions, 300
 Virtual Machines, 300
 Edge, 279
 Hyper-V, 296
 Internet Explorer, 277
 Store, 292, 294
 User-Mode Driver Framework, 215
 Windows. *See* Windows
micro-USB breakout board, 337
minify, 273
mmcblk0, 230
MMIO (memory-mapped I/O), 132
mnemonic, 140
`modinfo`, 230
momentary switch, 55
monitor, 131
Mosaic, 277
most significant bit, 75
Motorola 68000, 297
mount, 215
mouse, 131
Mozilla Firefox, 277
MS-DOS (Microsoft Disk Operating System), 198
multicore CPU, 128
multimeter, 46

continuity check, 340
measure
 current, 49
 resistance, 48
 voltage, 47
multitasking, 195
multithreaded, 204
MX (mail exchanger) record (DNS), 251

N

Nakamoto, Satoshi, 306
name, of variable, 162
NAND
 7400 integrated circuit, 65
 logical operator, 29
 symbol, 61
 truth table, 29, 61
nano tool, 348
native app, 292–294
NAT (Network Address Translation), 249
negative edge-triggered, 99
NES (Nintendo Entertainment System), 194
.NET, 180
Netscape Navigator, 277
netstat, 256
network, 234
 hub, 239
 ID, 242
 link, 238
 prefix, 240
 private, 249

stack, 235
switch, 239

Network Address Translation (NAT), 249

networking terms defined, 233–235

neutrons, 32

New Out of Box Software (NOOBS), 343

nibble (also nybble or nyble), 8

Nintendo Entertainment System (NES), 194

`nmap`, 257

node, 235

nonvolatile storage, 130

NOOBS (New Out of Box Software), 343

NOR

- 7402 integrated circuit, 65
- logical operator, 29
- symbol, 61
- truth table, 29, 61

NOT

- bitwise operator, 169
- Boolean operator, 171
- logical operator, 29
- symbol, 61
- truth table, 29, 61

NPN transistor, 57

NT. *See* Windows: NT

`NtCreateFile` function, 212

NT File System (NTFS), 215

ntoskrnl.exe, 200

number systems, 5

0

`objdump`, 156, 192, 228

object

 file, 147, 160

 in programming, 177

object-oriented programming

 class, 177

 field, 178

 instance, of a class, 177

 method, 177

octet, 240

Oculus, 308

ohms, 32

Ohm's law, 35

onion

 router, 302

 services, 302

opcode, 139

open circuit, 36

`open` function, 208, 224

open source, 197

Open Systems Interconnection (OSI), 236

OpenVZ, 296

operands, 126

operating system (OS), 195

 Android, 197, 293

 API, 207

 daemons, 196, 216

 Darwin, 198

 defined, 195

device driver, 196, 214
families, 197–199
FreeBSD, 198
iOS, 197, 293
kernel, 195
library, 196, 207, 213–214
 defined, 212
Linux, 197
macOS, 197
MS-DOS, 198
OS/2, 198
overview, 195–196
Raspberry Pi OS, 342
 File Manager, 346
 installing, 343
 text editor, 349
 using, 344
scheduler, 204
security, 217
services, 196, 216
shell, 195
system call, 210
 and APIs, 211
 CPU instructions, 211
Ubuntu, 198
Unix, 197
utilities, 196
VMS, 198
Windows, 197
Windows NT, 198
Windows XP, 198
Opteron, 124

OR

- 7432 integrated circuit, 65
 - bitwise operator, 169
 - Boolean operator, 171
 - implemented with switches, 57
 - logical operator, 27
 - symbol, 61
 - truth table, 27, 61
- Oracle Cloud, 301
- organizationally unique identifier (OUI), 254
- orphan process, 201
- OS (operating system). *See* operating system (OS)
- OS/2, 198
- OSI (Open Systems Interconnection), 236
- OUI (organizationally unique identifier), 254

P

PaaS (Platform as a Service), 300

package manager, 177

packet, 239

Pac-Man, 87

paging, 206

paradigm, 177

parallel execution, 128, 203

parameters, of a function, 174

partition, 215

`PathFindFileNameW` function, 212

PC (personal computer), 2

PC (program counter), 126, 150

PDP-7 minicomputer, 197

Pentium, 124
physical
 address space, 132
 core, 203
 memory address, 204
PID (process identifier), 201
pinout diagram, 64
 7400 series, 65
 7473, 112
pins, 62
`pip`, 177
pipe, 32
pipelining, 127
pixels, 21
place-value notation, 5
Platform as a Service (PaaS), 300
PlayStation 4, 198
`pmap`, 223
PMIO (port-mapped I/O), 132
PN2222 transistor, 66
PNG, 24
PNP transistor, 57
pointer, 166
polarized, 107
Portable Operating System Interface (POSIX), 208
port-mapped I/O (PMIO), 132
port number, 243
positional notation, 5
positive edge-triggered, 99
POSIX (Portable Operating System Interface), 208

potential energy, 33
powering digital circuits, 336–339
power-on reset, 114
power source, 33
prefixes, SI, 9
printer, 131
`printf` function, 184, 228
private IP addresses, 249
private key, 267, 303
private networks, 249
privilege level, 199
procedure. *See* function
process, 202
 defined, 200
 identifier (PID), 200
 `init`, 201
 virtual machine, 298
Process Explorer, 202
processor, 118, 122–123, 126, 129
 16-bit/32-bit/64-bit, 124
 cache, 129
 control unit, 125
 core, 128
 internals, 125
 multicore, 128
 pipelining, 127
 registers, 125
program, 118, 123, 138
 counter (PC), 126, 150
 header, 156

program flow, 174
 `elif` statement, 172
 `else` statement, 172
 `for` loop, 173
 `if` statement, 172
 looping, 173
 `while` loop, 173
programmability, 118
programming, 138
 high-level. *See* high-level programming languages
 C, 161
 C++, 161
 C#, 180
 compiled, 179
 intermediate, 180
 interpreted, 179
 Java, 180
 JavaScript, 268, 273–275
 Python, 161
 object-oriented
 class, 177
 field, 178
 instance, of a class, 177
 method, 177
paradigm, 177
 without an operating system, 193–194
Progressive Web App (PWA), 295
proof of work, 305
protocol, 234
protons, 32

prototypes, in programming, 273
proxy server, 249
pseudocode, 123
`ps` tool, 218
`pstree`, 201, 219
`pthread_create` function, 221
public key, 266, 303
pull-down resistor, 70
pulse, 98–102
pump, 33
pushbutton, 55, 68
PWA (Progressive Web App), 295
Python programming language, 161
 `http.server`, 285, 316
 `try/except/finally`, 318

Q

quantum, 204

R

race condition, 105
`raise` function, 184
`rand` function, 227
random access memory (RAM), 118
Raspberry Pi
 Imager, 343
 New Out of Box Software (NOOBS), 343
OS, 341–350
 File Manager, 346
 installing, 343

- text editor, 349
- using, 344
- React, 294
 - Native, 293
- `readelf`, 157, 229
- red, green, blue (RGB), 23
- registered ports, 244
- register file, 126
- registers, 125
 - destination, 139
 - instruction pointer, 150
 - program counter, 150
 - status, 143
- remote computing, history of, 298
- request, 235
 - HTTP, 264
- resistance, 32
 - measuring, 48
- resistor, 34
 - color-coded, 46
 - pull-down, 70
 - symbol, 35
- resistor-transistor logic (RTL), 63
- resolving, a hostname, 250
- resource, on the web, 261
- response, 235
 - HTTP, 264
- responsive web design, 294
- return value, 174
- RGB (red, green, blue), 23

ripple carry adder, 81
RJ45, 238
root domain, 251
router, 240
routing, 239
RTL (resistor–transistor logic), 63
runtime environment, 299

S

SaaS (Software as a Service), 301
Samsung Gear VR, 307
scheduler, 204
SCM (Service Control Manager), 216
scope, of variable, 162
SDK (Software Development Kit), 293
secondary storage, 130, 215
Secure Sockets Layer (SSL), 267
security, 217
Sega Genesis, 194, 297
segment, TCP, 243
sequential logic, 62, 91
series, 38
server, 2, 235
 DHCP, 248
 DNS, 250
 proxy, 249
 web, 262, 280–282
serverless computing, 300
Service Control Manager (SCM), 216
services, 196, 216

services.exe, 216
shared key, 266
shared object, 212
shell, 195
short circuit, 37
signed magnitude representation, 81
signed numbers, 81–85
SI (International System of Units), 9
Simple Mail Transfer Protocol (SMTP), 245
smartphone, 2
socket, 245
SoC (system-on-chip), 125
software, 138
Software as a Service (SaaS), 301
Software Development Kit (SDK), 293
solid-state drive (SSD), 130, 215
source code, 138, 179
SpiderMonkey, 279
square wave, 98
SRAM (static random access memory), 119
`srand` function, 227
SR latch, 94
 compared with JK flip-flop, 99
 implementation, 93
 symbol, 93
table, 92
 using in a circuit, 95–98
SSD (solid-state drive), 130, 215
`sshd`, 202
SSL (Secure Sockets Layer), 267

stack

 memory, 164

 network, 235

 overflow, 165

 pointer, 164

stack and heap memory, 164–166

standard library, 177, 208

static

 electricity, 66

 IP address, 247

 random access memory (SRAM), 119

 website, 280

status register, 143

storage

 partition, 215

 secondary, 215

`strace`, 226

string, 18

subnet, 240

 mask, 241

subroutine. *See* function

subtractive color model, 23

supervisor call, 211

supervisor mode, 199

surface web, 301

SVC instruction, 211

SWI instruction, 211

switch, 54

 bounce, 110

 electronic, 57

mechanical, 54, 70
momentary, 55
network, 239
pushbutton, 55, 68
slide, 68
symbolic link, 219
symmetric encryption, 266
`SYSCALL` instruction, 211
`SYSENTER` instruction, 211
system call, 210
 and APIs, 211
 CPU instructions, 211
`systemctl`, 231
`systemd`, 202, 216, 218
system-on-chip (SoC), 125

T

TCP/IP, 235
 model, 236
TCP (Transmission Control Protocol), 235, 243
terminal, 33
text, 18
T flip-flop, 100
 symbol, 101
 table, 101
TGID (thread group identifier), 203
thread, 203–204
 defined, 202
 ID, 202
thread group identifier (TGID), 203

TID (thread identifier), 202
`time` function, 227
TLS (Transport Layer Security), 267
TO-92 packaging, 66
top-level domain (TLD), 252
Tor, 302
Torvalds, Linus, 197
touchscreen, 131
`traceroute`, 258
transistor, 59, 119
 2N2222, 66
 as a switch, 58
 base, 57
 collector, 57
 emitter, 57
 NPN, 57
 PN2222, 66
 PNP, 57
 size, 128
transistor-transistor logic (TTL), 63
Transmission Control Protocol (TCP), 235, 243
transport layer, 243–245
Transport Layer Security (TLS), 267
`tree` tool, 231
troubleshooting circuits, 340–341
truth table, 26
 AND, 26, 61
 full adder, 79
 half adder, 77
 NAND, 29, 61

NOR, 29, 61
NOT, 29, 61
OR, 27, 61
XOR, 30, 61
TTL (transistor–transistor logic), 63
two’s complement, 82, 84–85
 notation, 83
 operation, 83
 terminology, 83
type, of variable, 162

U

Ubuntu, 198
UDP (User Datagram Protocol), 243
Unicode, 19
Uniform Resource Locator (URL)
 authority, 262
 path, 262
 query, 262
 relative, 264
 scheme, 262
Unity, 293
universal logic gates, 95
Universal Windows Platform (UWP), 209
Unix, 197
 Unix-like, 197
 API, 208
 unsigned numbers, 85–87
URL. *See* Uniform Resource Locator (URL)
user32.dll, 213

- user agent, 279
 - string, 279
- User Datagram Protocol (UDP), 243
- user mode, 199
 - address space, 205
 - bubble, 209
 - limitations, 209
- utilities, 196
- UWP (Universal Windows Platform), 209

V

- V8, 279
- value, of variable, 162
- variable, 164
 - assignment, 162
 - convenience, 149
 - declaration, 162
 - defined, 162
 - in C, 162
 - in Python, 163
- V_{cc} , 58
- vending machine, 92, 95
- video game consoles, 194
 - Atari 2600, 194
 - Nintendo Entertainment System (NES), 194
 - PlayStation 4, 198
 - Sega Genesis, 194, 297
 - Xbox One, 199
- virtual address, 204
 - space, 204
- VirtualBox, 296

virtualization, 295
virtual machine (VM), 180, 295
virtual memory, 205–207
 address, 204
 defined, 204
virtual reality (VR), 306
VMS, 198
VMware
 ESX, 296
 Player, 296
`void`, 166
volatile memory, 118
voltage, 32
 drop, 38
 Kirchhoff's voltage law, 38–39
 measuring, 47
 source, 33
 symbol, 36
volts, 32
VR (virtual reality), 306

W

W3C (World Wide Web Consortium), 271
water
 analogy, 32, 34
 pressure, 33
web
 address, 262
 app, 292, 294–295
 browser, 280
 address bar, 263

defined, 262
page rendering, 277–279

dark, 302
deep, 302
defined, 261
development frameworks, 294
distributed, 262
languages, 268–277
overview, 261–268
page, 262
protocols, 264–267
resource, 261
search, 267
server, 280–282
 defined, 262
 service, 275
 surface, 301

Web Hypertext Application Technology Working Group (WHATWG),
 271

WebKit, 278

website, 262
 dynamic, 280
 static, 280

WebVR, 308

WebXR, 308

well-known ports, 244

`wget`, 288

WHATWG (Web Hypertext Application Technology Working Group),
 271

`whereis`, 158

`while` loop, 173

Wi-Fi, 238
Windows, 197
 API, 208, 211
 executive, 200
 Mixed Reality, 308
 Native API, 211
 NT, 198
 Subsystem for Linux (WSL), 214
 system calls, 211
 UWP, 209
 Win16, 208
 Win32, 209
 win32k, 200
 Win64, 209
 XP, 198
word size, 124
work, 33
World Wide Web. *See* web
WorldWideWeb browser, 277
World Wide Web Consortium (W3C), 271
`write` function, 224
`write` system call, 211
WSL (Windows Subsystem for Linux), 214

X

x64, 124
x86, 124, 182
x86-64, 125
Xamarin, 293
Xbox One, 199

XML (Extensible Markup Language), 275

XOR

bitwise operator, 169

logical operator, 29

symbol, 61

truth table, 30, 61

XR, 308

Y

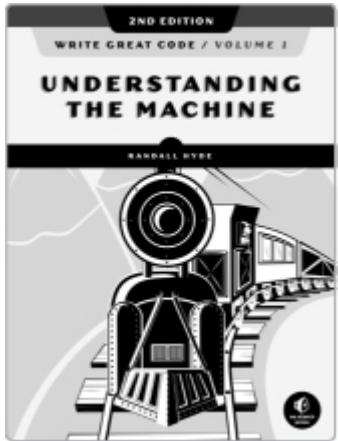
Yamaha YM2612, 297

How Computers Really Work is set in New Baskerville, Futura, Dogma, and TheSansMonoCondensed.

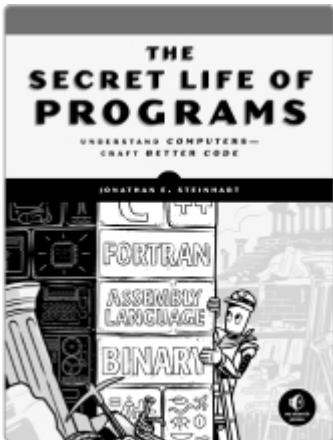
RESOURCES

Visit <https://nostarch.com/how-computers-really-work/> for errata and more information.

More no-nonsense books from  **NO STARCH PRESS**



WRITE GREAT CODE, VOLUME 1, 2ND EDITION
Understanding the Machine
by RANDALL HYDE
AUGUST 2020, 472 PP., \$49.95
ISBN 978-1-71850-036-5



THE SECRET LIFE OF PROGRAMS

Understand Computers—Craft Better Code

by JONATHAN E. STEINHART

504 PP., \$44.95

ISBN 978-1-59327-970-7



PYTHON CRASH COURSE, 2ND EDITION

A Hands-On, Project-Based Introduction to Programming

by ERIC MATTHES

544 PP., \$39.95

ISBN 978-1-59327-928-8



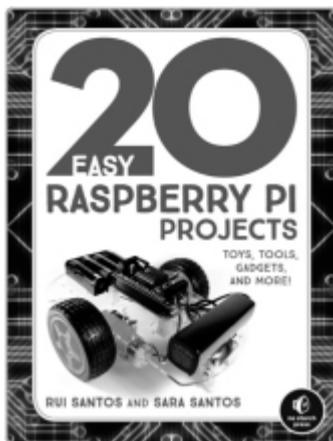
AUTOMATE THE BORING STUFF WITH PYTHON, 2ND EDITION

Practical Programming for Total Beginners

by AL SWEIGART

592 PP., \$39.95

ISBN 978-1-59327-992-9



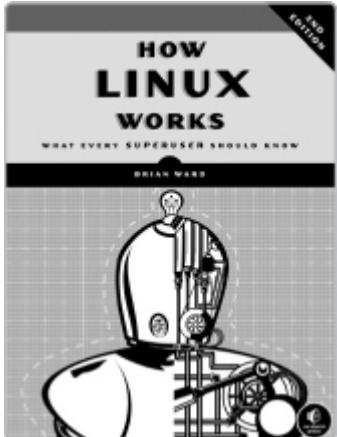
20 EASY RASPBERRY PI PROJECTS

Toys, Tools, Gadgets, and More!

by RUI SANTOS and SARA SANTOS

288 PP., \$24.95

ISBN 978-1-59327-843-4



HOW LINUX WORKS, 2ND EDITION

What Every Superuser Should Know

by BRIAN WARD

392 PP., \$39.95

ISBN 978-1-59327-567-9

PHONE:

800.420.7240 OR

415.863.9900

EMAIL:

SALES@NOSTARCH.COM

WEB:

WWW.NOSTARCH.COM



The Electronic Frontier Foundation (EFF) is the leading organization defending civil liberties in the digital world. We defend free speech on the Internet, fight illegal surveillance, promote the rights of innovators to develop new digital technologies, and work to ensure that the rights and freedoms we enjoy are enhanced – rather than eroded – as our use of technology grows.

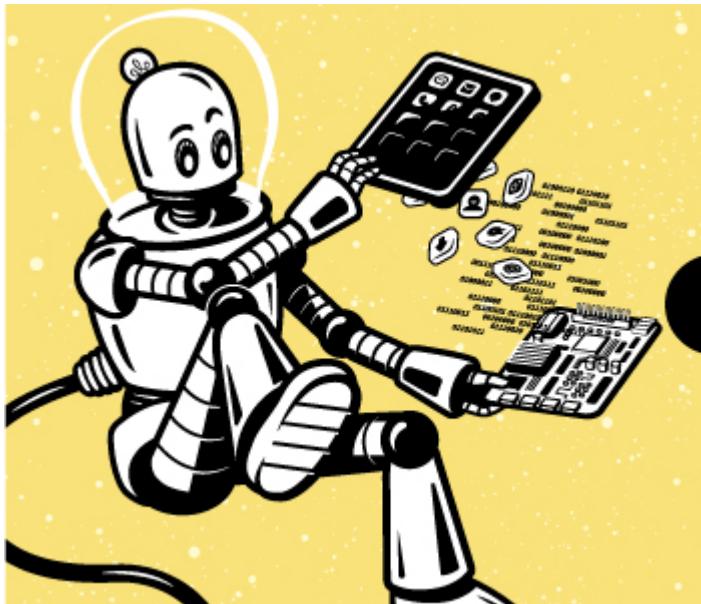
A stylized illustration of a city skyline composed of several buildings of varying heights. Each building features a small antenna tower on top. The background is filled with concentric, wavy lines radiating from behind the buildings, suggesting signal transmission or network coverage.

EFF.ORG

ELECTRONIC FRONTIER FOUNDATION

Protecting Rights and Promoting Freedom on the Electronic Frontier

FROM LOW-LEVEL CIRCUITS TO HIGH-LEVEL CODE



How Computers Really Work is a hands-on guide to the computing ecosystem: everything from circuits to memory and clock signals, machine code, programming languages, operating systems, and the internet.

But you won't just read about these concepts, you'll test your knowledge with exercises, and practice what you learn with 41 optional hands-on projects. Build digital circuits, craft a guessing game, convert decimal numbers to binary, examine virtual memory usage, run your own web server, and more.

Explore concepts like how to:

- Think like a software engineer as you use data to describe a real world concept
- Use Ohm's and Kirchhoff's laws to analyze an electrical circuit

- Think like a computer as you practice binary addition and execute a program in your mind, step-by-step

The book's projects will have you translate your learning into action, as you:

- Learn how to use a multimeter to measure resistance, current, and voltage
- Build a half adder to see how logical operations in hardware can be combined to perform useful functions
- Write a program in assembly language, then examine the resulting machine code
- Learn to use a debugger, disassemble code, and hack a program to change its behavior without changing the source code
- Use a port scanner to see which internet ports your computer has open
- Run your own server and get a solid crash course on how the web works

And since a picture is worth a thousand bytes, chapters are filled with detailed diagrams and illustrations to help clarify technical complexities.

Requirements for Optional Projects: See Appendix B for a list of recommended hardware.

ABOUT THE AUTHOR

Matthew Justice, a software engineer, spent 17 years at Microsoft where his work included debugging the Windows kernel, developing automated fixes, and leading a team of engineers building diagnostic tools and services. He has worked on everything from low-level software to high-level web applications.



THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com