EMBEDDED COMPUTING AND MECHATRONICS WITH THE PIC32 MICROCONTROLLER

Kevin M. Lynch, Nicholas Marchuk and Matthew L. Elwin

For more information visit www.nu32.org
Embedded Computing and Mechatronics with the PIC32 Microcontroller

For more information visit www.nu32.org
Embedded Computing and Mechatronics with the PIC32 Microcontroller

Kevin M. Lynch
Nicholas Marchuk
Matthew L. Elwin
Dedication

To Yuko, Erin, and Patrick.
—Kevin M. Lynch

To Mark and Liz.
—Nicholas Marchuk

To Hannah.
—Matthew L. Elwin

For more information
visit www.nu32.org
Figure Credits

The authors thank the following companies for permission to use their copyrighted images in this book.

* Contelec AG. www.contelec.ch/en.

For more information visit www.nu32.org
Contents

Preface ................................................................. xix
Acknowledgments .................................................... xxvii

I QUICKSTART ....................................................... 1

Chapter 1: Quickstart ................................................ 3
1.1 What You Need .................................................. 4
  1.1.1 Hardware .................................................. 4
  1.1.2 Software .................................................. 4
1.2 Compiling the Bootloader Utility ............................... 6
1.3 Compiling Your First Program ................................ 7
1.4 Loading Your First Program ................................... 9
1.5 Using make .................................................... 10
1.6 Chapter Summary ............................................... 13

Further Reading .................................................... 14

II FUNDAMENTALS ................................................... 15

Chapter 2: Hardware ................................................ 17
2.1 The PIC32 ....................................................... 17
  2.1.1 Pins, Peripherals, and Special Function Registers (SFRs) .................................................. 17
  2.1.2 PIC32 Architecture ...................................... 20
  2.1.3 The Physical Memory Map .............................. 27
  2.1.4 Configuration Bits ...................................... 28
2.2 The NU32 Development Board ................................. 28
2.3 Chapter Summary ............................................... 32
2.4 Exercises ....................................................... 32

Further Reading .................................................... 34

For more information
visit www.nu32.org
### Chapter 3: Software

- **3.1 The Virtual Memory Map** .......................................................... 35
- **3.2 An Example: simplePIC.c** ..................................................... 37
- **3.3 What Happens When You Build?** ........................................... 38
- **3.4 What Happens When You Reset the PIC32?** ............................... 39
- **3.5 Understanding simplePIC.c** .................................................. 40
  - **3.5.1 Down the Rabbit Hole** .................................................... 44
  - **3.5.2 The Header File p32mx795f512h.h** ..................................... 46
  - **3.5.3 Other Microchip Software: Harmony** .................................. 49
  - **3.5.4 The NU32bootloaded.ld Linker Script** ................................ 50
- **3.6 Bootloaded Programs vs. Standalone Programs** .......................... 51
- **3.7 Build Summary** ...................................................................... 53
- **3.8 Useful Command Line Utilities** .............................................. 55
- **3.9 Chapter Summary** .................................................................... 55
- **3.10 Exercises** ............................................................................... 56

### Chapter 4: Using Libraries

- **4.1 Talking PIC** .............................................................................. 60
- **4.2 The NU32 Library** .................................................................... 61
- **4.3 Bootloaded Programs** .............................................................. 62
- **4.4 An LCD Library** ....................................................................... 63
- **4.5 Microchip Libraries** ................................................................. 66
- **4.6 Your Libraries** .......................................................................... 67
- **4.7 Chapter Summary** .................................................................... 67
- **4.8 Exercises** ................................................................................ 67

### Chapter 5: Time and Space

- **5.1 Compiler Optimization** ............................................................. 70
- **5.2 Time and the Disassembly File** .............................................. 71
  - **5.2.1 Timing Using a Stopwatch (or an Oscilloscope)** ................... 71
  - **5.2.2 Timing Using the Core Timer** ............................................ 71
  - **5.2.3 Disassembling Your Code** ................................................ 72
  - **5.2.4 The Prefetch Cache Module** ............................................. 77
  - **5.2.5 Math** ................................................................................ 78
- **5.3 Space and the Map File** .......................................................... 78
- **5.4 Chapter Summary** .................................................................... 84
- **5.5 Exercises** ............................................................................... 84

### Chapter 6: Interrupts

- **6.1 Overview** ................................................................................. 91
- **6.2 Details** .................................................................................. 92
## Contents

6.3 Steps to Configure and Use an Interrupt ........................................... 99
6.4 Sample Code ................................................................. 99
6.4.1 Core Timer Interrupt .................................................. 99
6.4.2 External Interrupt ..................................................... 102
6.4.3 Speedup Due to the Shadow Register Set .................................. 103
6.4.4 Sharing Variables with ISRs ........................................... 106
6.5 Chapter Summary ........................................................... 107
6.6 Exercises ................................................................. 108
Further Reading ......................................................................... 111

### III PERIPHERAL REFERENCE

113

#### Chapter 7: Digital Input and Output ........................................... 115

7.1 Overview .............................................................................. 115
7.2 Details ................................................................................. 117
7.2.1 Change Notification ....................................................... 118
7.3 Sample Code ....................................................................... 119
7.4 Chapter Summary ............................................................... 120
7.5 Exercises ............................................................................ 121
Further Reading ......................................................................... 121

#### Chapter 8: Counter/Timers ......................................................... 123

8.1 Overview .............................................................................. 123
8.2 Details ................................................................................. 125
8.3 Sample Code ....................................................................... 128
8.3.1 A Fixed Frequency ISR .................................................. 128
8.3.2 Counting External Pulses ............................................... 128
8.3.3 Timing the Duration of an External Pulse ......................... 130
8.4 Chapter Summary ............................................................... 131
8.5 Exercises ............................................................................ 131
Further Reading ......................................................................... 131

#### Chapter 9: Output Compare ....................................................... 133

9.1 Overview .............................................................................. 133
9.2 Details ................................................................................. 134
9.3 Sample Code ....................................................................... 136
9.3.1 Generating a Pulse Train with PWM .................................... 136
9.3.2 Analog Output .............................................................. 137
9.4 Chapter Summary ............................................................... 141
9.5 Exercises ............................................................................ 141
Further Reading ......................................................................... 143

For more information
visit www.nu32.org

visit www.nu32.org
Chapter 10: Analog Input .................................................. 145
  10.1 Overview ................................................................. 145
  10.2 Details ................................................................. 149
  10.3 Sample Code .......................................................... 152
    10.3.1 Manual Sampling and Conversion .......................... 152
    10.3.2 Maximum Possible Sample Rate ......................... 153
  10.4 Chapter Summary .................................................... 157
  10.5 Exercises ............................................................. 158
  Further Reading .......................................................... 158

Chapter 11: UART ......................................................... 159
  11.1 Overview ................................................................. 159
  11.2 Details ................................................................. 161
  11.3 Sample Code .......................................................... 163
    11.3.1 Loopback .......................................................... 163
    11.3.2 Interrupt Based .................................................. 165
    11.3.3 NU32 Library ..................................................... 166
    11.3.4 Sending Data from an ISR ................................. 168
    11.3.5 Communication with MATLAB ............................ 172
    11.3.6 Communication with Python ............................... 173
  11.4 Wireless Communication with an XBee Radio ............... 174
  11.5 Chapter Summary .................................................... 175
  11.6 Exercises ............................................................. 175
  Further Reading .......................................................... 176

Chapter 12: SPI Communication ......................................... 177
  12.1 Overview ................................................................. 177
  12.2 Details ................................................................. 179
  12.3 Sample Code .......................................................... 181
    12.3.1 Loopback .......................................................... 181
    12.3.2 SRAM ............................................................. 183
    12.3.3 LSM303D Accelerometer/Magnetometer ............... 186
  12.4 Chapter Summary .................................................... 190
  12.5 Exercises ............................................................. 190
  Further Reading .......................................................... 190

Chapter 13: I²C Communication ........................................... 191
  13.1 Overview ................................................................. 191
  13.2 Details ................................................................. 193
  13.3 Sample Code .......................................................... 196
    13.3.1 Loopback .......................................................... 196
    13.3.2 Interrupt-Based Master ..................................... 200
    13.3.3 Accelerometer/Magnetometer ............................ 203
    13.3.4 OLED Screen .................................................... 205

For more information
visit www.nu32.org
### 13.3.5 Multiple Devices

208

### 13.4 Chapter Summary

211

### 13.5 Exercises

211

### Further Reading

211

### Chapter 14: Parallel Master Port

- **Overview**
  - 213
- **Details**
  - 214
- **Sample Code**
  - 215
- **Chapter Summary**
  - 219
- **Exercises**
  - 220
- **Further Reading**
  - 220

### Chapter 15: Input Capture

- **Overview**
  - 221
- **Details**
  - 221
- **Sample Code**
  - 223
- **Chapter Summary**
  - 225
- **Exercises**
  - 225
- **Further Reading**
  - 225

### Chapter 16: Comparator

- **Overview**
  - 227
- **Details**
  - 227
- **Sample Code**
  - 230
  - **16.3.1 Voltage Comparison**
  - 230
  - **16.3.2 Analog Output**
  - 231
- **Chapter Summary**
  - 231
- **Exercises**
  - 232
- **Further Reading**
  - 232

### Chapter 17: Sleep, Idle, and the Watchdog Timer

- **Overview**
  - 233
  - **17.1.1 Power Saving**
  - 233
  - **17.1.2 Watchdog Timer**
  - 234
- **Details**
  - 234
- **Sample Code**
  - 235
- **Chapter Summary**
  - 237
- **Exercises**
  - 237
- **Further Reading**
  - 238

### Chapter 18: Flash Memory

- **Overview**
  - 239
- **Details**
  - 240
23.1 The PID Controller ..................................................... 376
23.2 Variants of the PID Controller ................................... 379
23.3 Empirical Gain Tuning ................................................. 380
23.4 Model-Based Control .................................................. 381
## Chapter 24: Feedback Control of LED Brightness

- **24.1 Wiring and Testing the Circuit** ........................................... 389
- **24.2 Powering the LED with OC1** ........................................... 389
- **24.3 Playing an Open-Loop PWM Waveform** .................................. 390
- **24.4 Establishing Communication with MATLAB** ................................. 391
- **24.5 Plotting Data in MATLAB** ............................................... 391
- **24.6 Writing to the LCD Screen** ............................................. 395
- **24.7 Reading the ADC** .................................................... 395
- **24.8 PI Control** .......................................................... 395
- **24.9 Additional Features** .................................................... 396
- **24.10 Chapter Summary** .................................................... 396
- **24.11 Exercises** .......................................................... 397

## Chapter 25: Brushed Permanent Magnet DC Motors

- **25.1 Motor Physics** .......................................................... 399
- **25.2 Governing Equations** ................................................... 403
- **25.3 The Speed-Torque Curve** .............................................. 406
- **25.4 Friction and Motor Efficiency** .......................................... 410
- **25.5 Motor Windings and the Motor Constant** ......................... 412
- **25.6 Other Motor Characteristics** ........................................... 413
- **25.7 Motor Data Sheet** .................................................... 415
- **25.8 Chapter Summary** .................................................... 420
- **25.9 Exercises** .......................................................... 421
- Further Reading .......................................................... 425

## Chapter 26: Gearing and Motor Sizing

- **26.1 Gearing** .......................................................... 427
  - **26.1.1 Practical Issues** ................................................... 428
  - **26.1.2 Examples** .................................................... 429
- **26.2 Choosing a Motor and Gearhead** ....................................... 431
  - **26.2.1 Speed-Torque Curve** ........................................... 431
  - **26.2.2 Inertia and Reflected Inertia** ..................................... 432
  - **26.2.3 Choosing a Motor and Gearhead** ................................. 434
- **26.3 Chapter Summary** .................................................... 435
- **26.4 Exercises** .......................................................... 435
- Further Reading .......................................................... 437

## Chapter 27: DC Motor Control

- **27.1 The H-Bridge and Pulse Width Modulation** ............................ 439
  - **27.1.1 The H-Bridge** .................................................. 443
## Chapter 28: A Motor Control Project

28.1 Hardware ................................................................. 459
28.2 Software Overview .................................................. 460
28.3 Software Development Tips ......................................... 467
28.4 Step by Step ............................................................... 470
   28.4.1 Decisions, Decisions ........................................... 470
   28.4.2 Establishing Communication with a Terminal Emulator 471
   28.4.3 Establishing Communication with the Client ............... 473
   28.4.4 Testing the Encoder ............................................ 474
   28.4.5 Adding Encoder Menu Options ............................... 475
   28.4.6 PIC32 Operating Mode .......................................... 477
   28.4.7 Current Sensor Wiring and Calibration ...................... 478
   28.4.8 ADC for the Current Sensor .................................. 480
   28.4.9 PWM and the H-Bridge ......................................... 480
   28.4.10 PI Current Control and ITEST Mode ....................... 482
   28.4.11 Position Control ................................................ 484
   28.4.12 Trajectory Tracking ........................................... 484
28.5 Extensions .................................................................. 487
28.6 Chapter Summary ...................................................... 489
28.7 Exercises .................................................................. 490

## Chapter 29: Other Actuators

29.1 Solenoids .................................................................. 491
29.2 Speakers and Voice Coil Actuators ............................... 493
29.3 RC Servos .................................................................. 494
29.4 Stepper Motors .......................................................... 495
   29.4.1 Stepper Motor Coil Configurations ............................ 498
29.5 Brushless DC Motors .................................................. 499
   29.5.1 Interfacing the PIC32 to a BLDC ............................... 502
   29.5.2 A BLDC Library .................................................... 504
   29.5.3 Commutation Using Hall Sensor Feedback ................ 507
   29.5.4 Synchronous Driving Using Open-Loop Control .......... 509
29.6 Linear Brushless Motors ............................................. 512
29.7 Chapter Summary ...................................................... 513

For more information
visit www.nu32.org
Appendix A: A Crash Course in C ................................................................. 515
A.1 Quick Start in C .................................................................................. 515
A.2 Overview .............................................................................................. 517
A.3 Important Concepts in C ....................................................................... 518
A.3.1 Data Types ......................................................................................... 518
A.3.2 Memory, Addresses, and Pointers .................................................... 526
A.3.3 Compiling ......................................................................................... 528
A.4 C Syntax .................................................................................................. 529
A.4.1 Basic Syntax ....................................................................................... 539
A.4.2 Program Structure .............................................................................. 541
A.4.3 Preprocessor Commands ................................................................. 542
A.4.4 Typedefs, Structs, and Enums ......................................................... 545
A.4.5 Defining Variables ............................................................................. 547
A.4.6 Defining and Calling Functions ....................................................... 550
A.4.7 Math .................................................................................................. 551
A.4.8 Pointers ............................................................................................. 553
A.4.9 Arrays and Strings ............................................................................. 554
A.4.10 Relational Operators and TRUE/FALSE Expressions ................. 557
A.4.11 Logical and Bitwise Operators ....................................................... 558
A.4.12 Conditional Statements ................................................................. 559
A.4.13 Loops ............................................................................................... 560
A.4.14 The C Standard Library .................................................................. 562
A.4.15 Multiple File Programs and Libraries ............................................ 567
A.5 Exercises ............................................................................................... 573
Further Reading .......................................................................................... 585

Appendix B: Circuits Review ................................................................. 587
B.1 Basics .................................................................................................... 587
B.2 Linear Elements: Resistors, Capacitors, and Inductors .................... 589
B.2.1 Time Response of RC and RL Circuits ............................................. 592
B.2.2 Frequency Response of RC and RL Circuits ................................... 596
B.3 Nonlinear Elements: Diodes and Transistors ...................................... 599
B.3.1 Diodes .............................................................................................. 599
B.3.2 Bipolar Junction Transistors ............................................................ 601
B.4 Operational Amplifiers .......................................................................... 603
B.4.1 Practical Op Amp Considerations ..................................................... 605
B.4.2 Single Supply Design and Virtual Ground ...................................... 606
B.4.3 Instrumentation Amps ...................................................................... 607
B.5 Modular Circuit Design: Input and Output Impedance ..................... 608
Further Reading .......................................................................................... 609

For more information
visit www.nu32.org
Appendix C: Other PIC32 Models .................................................. 611
C.1 The PIC32MX5xx/6xx/7xx Family ............................................ 611
C.2 PIC32MX3xx/4xx Family .......................................................... 612
C.3 PIC32MX1xx/2xx Family .......................................................... 612
C.4 PIC32MX1xx/2xx/5xx 64-100 Pin Family .......................... 613
C.5 PIC32MX330/350/370/430/450/470 Family .............................. 614
C.6 PIC32MZ Family ................................................................. 614
C.7 Conclusion ................................................................. 615

Index ................................................................. 617
Preface

This book is about the Microchip 32-bit PIC32 microcontroller, its hardware, programming it in C, and interfacing it to sensors and actuators. This book also covers related mechatronics topics such as motor theory, choosing motor gearing, and practical introductions to digital signal processing and feedback control. This book is written for:

- **Anyone starting out with the Microchip PIC32 32-bit microcontroller.** Microchip documentation can be hard to navigate; this is the book we wish we had when we started!
- **The hobbyist ready to explore beyond Arduino.** Arduino software and its large user support community allow you to be up and running quickly with Atmel microcontrollers. But reliance on Arduino software prevents you from fully exploiting or understanding the capability of the microcontroller.
- **Teachers and students in mechatronics.** The exercises, online material, and associated kit are designed to support introductory, advanced, and flipped or online courses in mechatronics.
- **Anyone interested in mechatronics, actuators, sensors, and practical embedded control.**

Contents

This book was written based on the two-quarter mechatronics sequence at Northwestern University, ME 333 Introduction to Mechatronics and ME 433 Advanced Mechatronics. In ME 333, students learn about PIC32 hardware, fundamentals of programming the PIC32 in C, the use of some basic peripherals, and interfacing the PIC32 with sensors and actuators. In ME 433, material from the rest of the book is used as reference by groups working on projects. Students taking the sequence range from sophomores to graduate students. The only prerequisite is introductory circuit analysis and design; experience in C programming is not required. While experience in C would allow faster progression through the material, we decided not to require it, to make the course available to the broad set of students interested in the material. To partially compensate for the wide range of experience in C (from expert to none), we begin ME 333 with an intensive two-week introduction to fundamental C concepts and syntax using the “Crash Course in C” in Appendix A. We also take advantage of student expertise by facilitating peer mentoring.

For more information visit www.nu32.org
Preface

The goals of this book mirror those of the Northwestern mechatronics sequence:

- to provide the beginner a sound introduction to microcontrollers using the example of the PIC32, a modern 32-bit architecture;
- to do so by first providing an overview of microcontroller hardware, firm in the belief that microcontroller programming is much more grounded when tightly connected to the hardware that implements it;
- to provide a clear understanding of the fundamentals of professional PIC32 programming in C, which builds a foundation for further exploration of the PIC32’s capabilities using Microchip documentation and other advanced references;
- to provide reference material and sample code on the major peripherals and special features of the PIC32;
- to instill an understanding of the theory of motor operation and control; and
- to teach how microcontroller peripherals can be used to interface with sensors and motors.

To achieve these goals, the book is divided into five main parts:

1. **Quickstart.** This part (Chapter 1) allows the student to get up and running with the PIC32 quickly.
2. **Fundamentals.** After achieving some early success with the quickstart, the five chapters in Fundamentals (Chapters 2 to 6) examine the PIC32 hardware, the build process in C and the connection of the code to the hardware, the use of libraries, and two important topics for real-time embedded computing: interrupts and the time and space efficiency of code. The time investment in these chapters provides the foundation needed to move quickly through later chapters and to profit from other reference material, like Microchip’s PIC32 Reference Manual, Data Sheets, and XC32 C/C++ Compiler User’s Guide.
3. **Peripheral Reference.** This part (Chapters 7 to 20) gives details on the operation of the various peripherals on the PIC32, as well as sample code and applications. It is primarily reference material that can be read in any order, though we recommend the first few chapters (digital I/O, counter/timers, output compare, and analog input) be covered in order. The peripheral reference concludes with an introduction to Harmony, Microchip’s recent framework for high-level programming of PIC32s.
4. **Mechatronics.** This part (Chapters 21 to 29) focuses on interfacing sensors to a microcontroller, digital signal processing, feedback control, brushed DC motor theory, motor sizing and gearing, control by a microcontroller, and other actuators such as brushless motors, stepper motors, and servo motors.
5. **Appendixes.** The appendixes cover background topics such as analysis of simple circuits and an introduction to programming in C. We have our students first get used to writing C programs on their laptops, and compiling with gcc, before moving on to programming a microcontroller.

For more information visit www.nu32.org
In ME 333, we cover the crash course in C; the Quickstart; the Fundamentals; select topics from the Peripheral Reference (digital I/O, counter/timers, output compare/PWM, and analog input); and simple sensor interfacing, DC motor theory, motor sizing and gearing, and control of a DC motor from the Mechatronics part. Other chapters are used for reference in ME 433 and other projects that students undertake.

**Choices made in this book**

We made several choices about how to teach mechatronics in ME 333, and those choices are reflected in this book. Our choices are based on the desire to expose our students to the topics they will need to integrate sensors and actuators and microcontrollers professionally, subject to the constraint that most students will take only one or two courses in mechatronics. Our choices are based on what we believe to be the smallest building blocks that a mechatronics engineer needs to know about. For example, we do not attempt to teach microcontroller architecture at the level that a computer engineer might learn it, since a mechatronics engineer is not likely to design a microcontroller. On the other hand, we also do not rely on software and hardware abstractions that keep the budding mechatronics engineer at arm’s length from concepts needed to progress beyond the level of a hobbyist. With that philosophy in mind, here are some of the choices made for ME 333 and this book:

- **Microcontrollers vs. sensors and actuators.** Mechatronics engineering integrates sensors, actuators, and microcontrollers. Handing a student a microcontroller development board and sample code potentially allows the course to focus on the sensors and actuators part. In ME 333, however, we opted to make understanding the hardware and software of the microcontroller approximately 50% of the course. This choice recognizes the fundamental role microcontrollers play in mechatronics, and that mechatronics engineers must be comfortable with programming.

- **Choice of microcontroller manufacturer.** There are many microcontrollers on the market, with a wide variety of features. Manufacturers include Microchip, Atmel, Freescale, Texas Instruments, STMicroelectronics, and many others. In particular, Atmel microcontrollers are used in Arduino boards. Arduinos are heavily used by hobbyists and in K-12 and university courses in large part due to the large online user support community and the wide variety of add-on boards and user-developed software libraries. In this book, we opt for the commercially popular Microchip PIC microcontrollers, and we avoid the high-level software abstractions synonymous with Arduino. (Arduinos are used in other Northwestern courses, particularly those focusing on rapid product prototyping with little mechatronics design.)

- **Choice of a particular microcontroller model.** Microchip’s microcontroller line consists of hundreds of different models, including 8-bit, 16-bit, and 32-bit architectures. We have chosen a modern 32-bit architecture. And instead of trying to write a book that deals with all PIC32 models, which includes six different families of PIC32s as of this writing (see

For more information visit www.nu32.org
Appendix C), we focus on one particular model: the PIC32MX795F512H. The reasons for this choice are (a) it is a powerful chip with plenty of peripherals and memory (128 KB data RAM and 512 KB program flash), and, more importantly, (b) focusing on a single chip allows us to be concrete in the details of its operation. This is especially important when learning how the hardware relates to the software. (One of the reasons Microchip’s documentation is difficult to read, and is so full of exceptions and special cases, is that it is written to be general to all PIC32s in the case of the Reference Manual, or all PIC32s in a specific family in the case of the Data Sheets.) Once the reader has learned about the operation of a specific PIC32, it is not too difficult to learn about the differences for a different PIC32 model.

- **Programming language: C++ vs. C vs. assembly.** C++ is a relatively high-level language, C is lower level, and assembly is lower still. We choose to program in C because of the portability of the language, while staying relatively close to the assembly language level and minimizing abstractions introduced by C++.

- **Integrated Development Environment vs. command line.** MPLAB X is Microchip’s Integrated Development Environment (IDE) for developing software for PICs. So why do we avoid using it in this book? Because we feel that it hides key steps in understanding how the code you write turns into an executable for the PIC32. In this book, code is written in a text editor and the C compiler is invoked at the command line. There are no hidden steps. Once the reader has mastered the material in the first few chapters of this book, MPLAB will no longer be mysterious.

- **Use of the Harmony software vs. ignoring it.** Microchip provides an extensive library of middleware, device drivers, system services, and other software to support all of their PIC32 models. One goal of this software is to allow you to write programs that are portable across different PIC32 models. To achieve this, however, a significant amount of abstraction is introduced, separating the code you write from the actual hardware implementation. This is bad pedagogically as you learn about the PIC32. Instead, we use low-level software commands to control the PIC32’s peripherals, reinforcing the hardware documentation in this book and in the Data Sheet and Reference Manual. Only with the more complicated peripherals do we use the Harmony software, specifically for USB, in Chapter 20.

- **Sample code vs. writing it yourself.** The usual way to learn to program PIC32s is to take some working sample code and try to modify it to do something else. This is natural, except that if your modified code fails, you are often left with no idea what to do. In this book we provide plenty of sample code, but we also focus on the fundamentals of programming the PIC32 so that you learn to write code from scratch as well as strategies to debug if things go wrong (Figure 0.1).

The philosophy represented by the choices above can be summed up succinctly: There should be no magic steps! You should know how and why the code you write works, and how it
connects to the hardware. You should not be simply modifying opaque and abstract code, compiling with a mysterious IDE, and hoping for the best.

The NU32 development board

The NU32 development board was created to support this book. If you do not have the board, you can still learn a lot about how a PIC32 works from reading this book. We highly recommend that you get the NU32 board and the kit of mechatronics parts, however, to allow you to work through the examples in the book.

In keeping with the “no magic” philosophy, the primary function of the NU32 is to break out the pins of the PIC32MX795F512H to a solderless prototyping breadboard, to allow easy wiring to the pins. Otherwise we try to keep the board as bare bones and inexpensive as possible, leaving external circuits to the reader. To allow you to get up and running as quickly as possible, though, the board does provide a few devices external to the PIC32: two LEDs and two buttons for simple user interaction; a 3.3 V regulator (to provide power to the PIC32) and a 5 V regulator (to provide a commonly needed voltage); a resonator to provide a clock signal; and a USB-to-UART chip that simplifies communication between the user’s computer and the PIC32.

The PIC32 on the NU32 comes with a bootloader program pre-installed, allowing you to program the PIC32 with just a USB cable. The NU32 can also be programmed directly using a programmer device, like the PICkit 3. This is covered in Chapter 3.6.

For more information
visit www.nu32.org
How to use this book in a course

Mechatronics is fundamentally an integrative discipline, requiring knowledge of microcontrollers, programming, circuit design, sensors, signal processing, feedback control, and motors. This book contains a practical introduction to these topics.

Recognizing that most students take no more than one or two courses in mechatronics, however, this book does not delve deeply into the mathematical theory underlying topics such as linear systems, circuit analysis, signal processing, or control theory. Instead, a course based on this book is meant to motivate further theoretical study in these disciplines by exposing students to their practical applications.

As a result, students need only a basic background in circuits and programming to be able to take a course based on this book. At Northwestern, this means that students take ME 333 as early as their sophomore year. ME 333 is an intense 11-week quarter, covering, in order:

- Appendix A, a Crash Course in C. (Approximately 2 weeks.)
- Chapters 1–6, fundamentals of hardware and software of the PIC32 microcontroller. (Approximately 3 weeks.)
- Chapters 7–10, covering digital input and output, counter/timers, output compare/PWM, and analog input. These chapters are primarily used as reference in the context of the following assignment.
- Chapters 23 and 24, on feedback control and PI control of the brightness of an LED using a phototransistor for feedback. This project is the students’ first significant project using the PIC32 for embedded control. It also serves as a warmup for the final project. (Approximately 2 weeks.)
- Chapter 25 on theory and experimental characterization of a brushed DC motor. (Approximately 1 week.)
- Introduction to encoders and current sensing in Chapter 21 and all of Chapters 27 and 28 on DC motor control. Chapter 27 introduces all the hardware and software elements of a professional DC motor control system, including a nested-loop control system with an outer-loop motion controller and an inner-loop current controller. Chapter 28 is a chapter-long project that applies the ideas, leading the student through a significant software design project to develop a motor control system that interfaces with a menu system in MATLAB. This “capstone” project is motivated by professional motor amplifier design and integrates the student’s knowledge of the PIC32, C programming, brushed DC motors, feedback control, and the interfacing of sensors and actuators with a PIC32. (Approximately 3 weeks.)

---

1 Because other courses generally do not cover the operation of motors, this book goes into greater detail on motor theory.
This is a very full quarter, which would be less intense if students were required to know C before taking the course.

ME 333 at Northwestern is taught as a flipped class. Students watch videos that support the text on their own time, then work on assignments and projects during class time while the instructor and TAs circulate to help answer questions. Students bring their laptops and portable mechatronics kits to every class. This kit includes an inexpensive function generator and oscilloscope, the nScope, that uses their laptop as the display. Thus ME 333 does not use a lab facility; students use the classroom and their own dorm rooms. Students work and learn together during classes, but each student completes her own assignment individually. The follow-on course ME 433 focuses on more open-ended mechatronics projects in teams and makes extensive use of a mechatronics lab that is open to students 24/7.

For a 15-week semester, good additions to the course would be two weeks on different sensor technologies (Chapter 21) and digital signal processing of sensor data (Chapter 22). Another week should also be devoted to the final motor control project (Chapter 28), to allow students to experiment with various extensions. Time permitting, other common actuators (e.g., steppers, RC servos, and brushless motors) could be covered in Chapter 29.

For a two-quarter or two-semester sequence, the second course could focus on open-ended team design projects, similar to ME 433 at Northwestern. The book then serves as a reference. Other appropriate material includes chapters on communication protocols and supporting PIC32 peripherals (e.g., UART, SPI, I²C, USB, and CAN).

Website, videos, and flipped classrooms

The book’s website, www.nu32.org, has links to downloadable data sheets, sample code, PCB layouts and schematics, chapter extensions, errata, and other useful information and updates. This website also links to short videos that summarize many of the chapters. These videos can be used to flip a traditional classroom, as in ME 333, allowing students to watch the lectures at home and to use class time to ask questions and work on projects.

Other PIC32 references

One goal of this book is to organize Microchip reference material in a logical way, for the beginner. Another goal is to equip the reader to be able to parse Microchip documentation. This ability allows the reader to continue to develop her PIC32 programming abilities beyond the limits of this book. The reader should download and have at the ready the first two references below; the others are optional. The readings are summarized in Figure 0.2.

- **The PIC32 Reference Manual.** The Reference Manual sections describe software and hardware for all PIC32 families and models, so they can sometimes be confusing in their
generality. Nevertheless, they are a good source for understanding the functions of the PIC32 in more detail. Some of the sections, particularly the later ones, focus on the PIC32MZ family and are not relevant to the PIC32MX795F512H.

- **The PIC32MX5xx/6xx/7xx Family Data Sheet.** This Data Sheet provides details specific to the PIC32MX5xx/6xx/7xx family. In particular, the Memory Organization section of the Data Sheet clarifies which special function registers (SFRs) are included on the PIC32MX795F512H, and therefore which Reference Manual functions are available for that model.


- **(Optional) MPLAB Harmony Help.** This documentation, which comes with the Harmony installation, can be helpful once you start writing more complex code that uses the Harmony software.

- **(Optional) MIPS32 Architecture for Programmers manuals and other MIPS32 documentation.** If you are curious about the MIPS32 M4K CPU, which is used on the PIC32MX795F512H, and its assembly language instruction set, you can find references online.

For more information visit www.nu32.org
Acknowledgments

This book has benefited from the feedback of many students and teaching assistants in ME 333 at Northwestern over the years, particularly Alex Ansari, Philip Dames, Mike Hwang, Andy Long, David Meyer, Ben Richardson, Nelson Rosa, Jarvis Schultz, Jian Shi, Craig Shultz, Matt Turpin, and Zack Woodruff.
Quickstart
Quickstart

Edit, compile, run, repeat: familiar to generations of C programmers, this mantra applies to programming in C, regardless of platform. Architecture, program loading, input and output: these details differ between your computer and the PIC32. Architecture refers to processor type: your computer’s x86-64 CPU and the PIC32’s MIPS32 CPU understand different machine code and therefore require different compilers. Your computer’s operating system allows you to seamlessly run programs; the PIC32’s bootloader writes programs it receives from your computer to flash memory and executes them when the PIC32 resets. You interact directly with your computer via the screen and keyboard; you interact indirectly with the PIC32 using a terminal emulator to relay information between your computer and the microcontroller. As you can see, programming the PIC32 requires attention to details that you probably ignore when programming your computer.

Armed with an overview of the differences between computer programming and microcontroller programming, you are ready to get your hands dirty. The rest of this chapter will guide you through gathering the hardware and installing the software necessary to program the PIC32. You will then verify your setup by running two programs on the PIC32. By the end of the chapter, you will be able to compile and run programs for the PIC32 (almost) as easily as you compile and run programs for your computer!

Throughout this book, we will refer to “the PIC32.” Although there are many PIC32 models, for us “the PIC32” is shorthand for the PIC32MX795F512H. While most of the concepts in this book apply to many PIC32 models, you should be aware that some of the details differ between models. (See Appendix C for a discussion of the differences.) Further, we refer to the PIC32MX795F512H as it is configured on the NU32 development board; in particular, it is powered by 3.3 V and is clocked by a system clock and a peripheral bus clock at 80 MHz. You will learn more about these details in Chapter 2.

---

1 Your computer also has a bootloader. It runs when you turn the computer on and loads the operating system. Also, operating systems are available for the PIC32, but we will not use them in this book.
1.1 What You Need

This section explains the hardware and software that you need to program the PIC32. Links to purchase the hardware and download the free software are provided at the book’s website, www.nu32.org.

1.1.1 Hardware

Although PIC32 microcontrollers integrate many devices on a single chip, they also require external circuitry to function. The NU32 development board, shown in Figure 1.1, provides this circuitry and more: buttons, LEDs, breakout pins, a USB port, and a virtual USB serial port. The examples in this book assume that you use this board. You will also need the following hardware:

1. **Computer with a USB port.** The host computer is used to create PIC32 programs. The examples in this book work with the Linux, Windows, and Mac operating systems.
2. **USB A to mini-B cable.** This cable carries signals between the NU32 board and your computer.
3. **AC/DC adapter (6 V).** This cable provides power to the PIC32 and NU32 board.

1.1.2 Software

Programming the PIC32 requires various software. You should be familiar with some of the software from programming your computer in C; if not, refer to Appendix A.
For your convenience, we have aggregated the software you need at the book’s website. You should download and install all of the following software.

1. **The command prompt** allows you to control your computer using a text-based interface. This program, `cmd.exe` on Windows, `Terminal` on Mac, and `bash` on Linux, comes with your operating system so you should not need to install it. See Appendix A for more information about the command line.

2. **A text editor** allows you to create text files, such as those containing C source code. See Appendix A for more information.

3. **A native C compiler** converts human-readable C source code files into machine code that your computer can execute. We suggest the free GNU compiler, `gcc`, which is available for Windows, Mac, and Linux. See Appendix A for more information.

4. **Make** simplifies the build process by automatically executing the instructions required to convert source code into executables. After manually typing all of the commands necessary create your first program, you will appreciate `make`.

5. **The Microchip XC32 compiler** converts C source files into machine code that the PIC32 understands. This compiler is known as a *cross compiler* because it runs on one processor architecture (e.g., x86-64 CPU) and creates machine code for another (e.g., MIPS32). This compiler installation also includes C libraries to help you control PIC32-specific features. Note where you install the compiler; we will refer to this directory as `<xc32dir>`. If you are asked during installation whether you would like to add XC32 to your path variable, do so.

6. **MPLAB Harmony** is Microchip’s collection of libraries and drivers that simplify the task of writing code targeting multiple PIC32 models. We will use this library only in Chapter 20; however, you should install it now. Note the installation directory, which we will refer to as `<harmony>`.

7. **The FTDI Virtual COM Port Driver** allows you to use a USB port as a “virtual serial communication (COM) port” to talk to the NU32 board. This driver is already included with most Linux distributions, but Windows and Mac users may need to install it.

8. **A terminal emulator** provides a simple interface to a COM port on your computer, sending keyboard input to the PIC32 and displaying output from the PIC32. For Linux/Mac, you can use the built-in `screen` program. For Windows, we recommend you download PuTTY. Remember where you install PuTTY; we refer to this directory as `<puttyPath>`.

9. **The PIC32 quickstart code** contains source code and other support files to help you program the PIC32. Download `PIC32quickstart.zip` from the book’s website, extract it, and put it in a directory that you create. We will refer to this directory as `<PIC32>`. In `<PIC32>` you will keep the quickstart code, plus all of the PIC32 code you write, so make sure the directory name makes sense to you. For example, depending on your operating system, `<PIC32>` could be `/Users/kevin/PIC32` or `C:\Users\kevin\Documents\PIC32`. In `<PIC32>`, you should have the following three files and one directory:
Chapter 1

- `nu32utility.c`: a program for your computer, used to load PIC32 executable programs from your computer to the PIC32
- `simplePIC.c`, `talkingPIC.c`: PIC32 sample programs that we will test in this chapter
- `skeleton`: a directory containing
  - `Makefile`: a file that will help us compile future PIC32 programs
  - `NU32.c`, `NU32.h`: a library of useful functions for the NU32 board
  - `NU32bootloaded.ld`: a linker script used when compiling programs for the PIC32

We will learn more about each of these shortly.

You should now have code in the following directories (if you are a Windows user, you will also have PuTTY in the directory `<puttyPath>`):

- `<xc32dir>`. The Microchip XC32 compiler. **You will never modify code in this directory.** Microchip wrote this code, and there is no reason for you to change it. Depending on your operating system, your `<xc32dir>` could look something like the following:
  - `/Applications/microchip/xc32`
  - `C:\Program Files (x86)\Microchip\xc32`
- `<harmony>`. Microchip Harmony. **You will never modify code in this directory.** Depending on your operating system, your `<harmony>` could look something like the following:
  - `/Users/kevin/microchip/harmony`
  - `C:\microchip\harmony`
- `<PIC32>`. Where PIC32 quickstart code, and code you will write, is stored, as described above.

Now that you have installed all of the necessary software, it is time to program the PIC32. By following these instructions, not only will you run your first PIC32 program, you will also verify that all of the software and hardware is functioning properly. Do not worry too much about what all the commands mean, we will explain the details in subsequent chapters.

**Notation:** Wherever we write `<something>`, replace it with the value relevant to your computer. On Windows, use a backslash (`\`) and on Linux/Mac use a slash (`/`) to separate the directories in a path. At the command line, place paths that contain spaces between quotation marks (i.e., "C:\Program Files"). Enter the text following a `>` at the command line. Use a single line, even if the command spans multiple lines in the book.

### 1.2 Compiling the Bootloader Utility

The bootloader utility, located at `<PIC32>/nu32utility.c`, sends compiled code to the PIC32. To use the bootloader utility you must compile it. Navigate to the `<PIC32>` directory by typing:
Verify that `<PIC32>/nu32utility.c` exists by executing the following command, which lists all the files in a directory:

- **Windows**
  > dir
- **Linux/Mac**
  > ls

Next, compile the bootloader utility using the native C compiler `gcc`:

- **Windows**
  > gcc nu32utility.c -o nu32utility -lwinmm
- **Linux/Mac**
  > gcc nu32utility.c -o nu32utility

When you successfully complete this step the executable file `nu32utility` will be created. Verify that it exists by listing the files in `<PIC32>`.

### 1.3 Compiling Your First Program

The first program you will load onto your PIC32 is `<PIC32>/simplePIC.c`, which is listed below. We will scrutinize the source code in Chapter 3, but reading it now will help you understand how it works. Essentially, after some setup, the code enters an infinite loop that alternates between delaying and toggling two LEDs. The delay loops infinitely while the USER button is pressed, stopping the toggling.

**Code Sample 1.1 simplePIC.c. Blinking Lights on the NU32, Unless the USER Button Is Pressed.**

```c
#include <xc.h> // Load the proper header for the processor

void delay(void);

int main(void) {
    TRISF = 0x0FFFC; // Pins 0 and 1 of Port F are LED1 and LED2. Clear
                      // bits 0 and 1 to zero, for output. Others are inputs.
    LATFbits.LATF0 = 0; // Turn LED1 on and LED2 off. These pins sink current
    LATFbits.LATF1 = 1; // on the NU32, so "high" (1) = "off" and "low" (0) = "on"

    while(1) {
        delay();
        LATFINV = 0x0003; // toggle LED1 and LED2: same as LATFINV = 0x3;
    }
    return 0;
}
```

For more information
visit www.nu32.org
8 Chapter 1

```c
void delay(void) {
    int j;
    for (j = 0; j < 1000000; j++) { // number is 1 million
        while(!PORTDbits.RD7) {
            // Pin D7 is the USER switch, low (FALSE) if pressed.
        }
    }
}
```

To compile this program you will use the `xc32-gcc` cross compiler, which compiles code for the PIC32’s MIPS32 processor. This compiler and other Microchip tools are located at `<xc32dir>/xc32ver/bin`, where `<xc32ver>` refers to the XC32 version (e.g., v1.40). To find `<xc32ver>` list the contents of the Microchip XC32 directory, e.g.,

```bash
> ls <xc32dir>
```

The subdirectory displayed is your `<xc32ver>` value. If you happen to have installed two or more versions of XC32, you will always use the most recent version (the largest version number).

Next you will compile `simplePIC.c` and create the executable `hex file`. To do this, you first create the `simplePIC.elf` file and then you create the `simplePIC.hex` file. (This two-step process will be discussed in greater detail in Chapter 3.) Issue the following commands from your `<PIC32>` directory (where `simplePIC.c` is), being sure to replace the text between the <> with the values appropriate to your system. Remember, if the paths contain spaces, you must surround them with quotes (i.e., "C:\Program Files\xc32\v1.40\bin\xc32-gcc").

```bash
> <xc32dir>/xc32ver/bin/xc32-gcc -mprocessor=32MX795F512H -o simplePIC.elf -Wl,-script=skeleton/NU32bootloaded.ld simplePIC.c
> <xc32dir>/xc32ver/bin/xc32-bin2hex simplePIC.elf
```

The `-Wl` is “-W ell” not “-W one.” You can list the contents of `<PIC32>` to make sure both `simplePIC.elf` and `simplePIC.hex` were created. The hex file contains MIPS32 machine code in a format that the bootloader understands, allowing it to load your program onto the PIC32.

If, when you installed XC32, you selected to have XC32 added to your path, then in the two commands above you could have simply typed

```bash
> xc32-gcc -mprocessor=32MX795F512H -o simplePIC.elf -Wl,-script=skeleton/NU32bootloaded.ld simplePIC.c
> xc32-bin2hex simplePIC.elf
```

and your operating system would be able to find `xc32-gcc` and `xc32-bin2hex` without needing the full paths to them.

For more information
visit www.nu32.org
1.4 Loading Your First Program

Loading a program onto the PIC32 from your computer requires communication between the two devices. When the PIC32 is powered and connected to a USB port, your computer creates a new serial communication (COM) port. Depending on your specific system setup, this COM port will have different names. Therefore, we will determine the name of your COM port through experimentation. First, with the PIC32 unplugged, execute the following command to enumerate the current COM ports, and note the names that are listed:

- **Windows:**
  ```shell```
  > mode
  ```shell```
- **Mac:**
  ```shell```
  > ls /dev/tty.*
  ```shell```
- **Linux:**
  ```shell```
  > ls /dev/ttyUSB*
  ```shell```

Next, plug the NU32 board into the wall using the AC adapter, turn the power switch on, and verify that the red “power” LED illuminates. Connect the USB cable from the NU32’s mini-B USB jack (next to the power jack) to a USB port on the host computer. Repeat the steps above, and note that a new COM port appears. If it does not appear, make sure that you installed the FTDI driver from Section 1.1.2. The name of the port will differ depending on the operating system; therefore we have listed some typical names:

- **Windows:** COM4
- **Mac:** /dev/tty.usbserial-DJ00DV5V
- **Linux:** /dev/ttyUSB0

Your computer, upon detecting the NU32 board, has created this port. Your programs and the bootloader use this port to communicate with your computer.

After identifying the COM port, place the PIC32 into program receive mode. Locate the RESET button and the USER button on the NU32 board (Figure 1.1). The RESET button is immediately above the USER button on the bottom of the board (the power jack is the board’s top). Press and hold both buttons, release RESET, and then release USER. After completing this sequence, the PIC32 will flash LED1, indicating that it has entered program receive mode.

Assuming that you are still in the `<PIC32>` directory, start the loading process by typing

- **Windows**
  ```shell```
  nu32utility <COM> simplePIC.hex
  ```shell```
- **Linux/Mac**
  ```shell```
  > ./nu32utility <COM> simplePIC.hex
  ```shell```

For more information visit www.nu32.org
where `<COM>` is the name of your COM port. After the utility finishes, LED1 and LED2 will flash back and forth. Hold USER and notice that the LEDs stop flashing. Release USER and watch the flashing resume. Turn the PIC32 off and then on. The LEDs resume blinking because you have written the program to the PIC32’s nonvolatile flash memory. Congratulations, you have successfully programmed the PIC32!

### 1.5 Using `make`

As you just witnessed, building an executable for the PIC32 requires several steps. Fortunately, you can use `make` to simplify this otherwise tedious and error-prone procedure. Using `make` requires a Makefile, which contains instructions for building the executable. We have provided a Makefile in `<PIC32>/skeleton`. Prior to using `make`, you need to modify `<PIC32>/skeleton/Makefile` so that it contains the paths and COM port specific to your system.

Aside from the paths you have already used, you need your terminal emulator’s location, `<termEmu>`, and the Harmony version, `<harmVer>`. On Windows, `<termEmu>` is `<puttyPath>/putty.exe` and for Linux/Mac, `<termEmu>` is `screen`. To find Harmony’s version, `<harmVer>`, list the contents of the `<harmony>` directory. Edit `<PIC32>/skeleton/Makefile` and update the first five lines as indicated below.

```
XC32PATH=\xc32dir/\xc32ver/bin
HARMONYPATH=\harmony/\harmVer
NU32PATH=\PIC32
PORT=\COM
TERMEMU=\termEmu
```

In the Makefile, do not surround paths with quotation marks, even if they contain spaces.

If your computer has more than one USB port, you should always use the same USB port to connect your NU32. Otherwise, the name of the COM port may change, requiring you to edit the Makefile again.

After saving the Makefile, you can use the skeleton directory to easily create new PIC32 programs. The skeleton directory contains not only the Makefile, but also the NU32 library (NU32.h and NU32.c), and the linker script NU32bootloaded.ld, all of which will be used extensively throughout the book. The Makefile automatically compiles and links every .c file in the directory into a single executable; therefore, your project directory should contain all the C files you need and none that you do not want!

---

2 Windows: Write the ports as `\\.\COMx` rather than `COMx`. Linux: To avoid needing to execute commands as root, add yourself to the group that owns the COM port (e.g., uucp).
Each new project you create will have its own directory in <PIC32>, e.g.,
<PIC32>/<projectdir>. We now explain how to use the <PIC32>/skeleton directory to create a
new project, using <PIC32>/talkingPIC.c as an example. For this example, we will name the
project talkingPIC, so <projectdir> is talkingPIC. By following this procedure, you will have
access to the NU32 library and will be able to avoid repeating the previous setup steps. Make
sure you are in the <PIC32> directory, then copy the <PIC32>/skeleton directory to the new
project directory:

- **Windows**
  > mkdir <projectdir>
  > copy skeleton\*.* <projectdir>

- **Linux/Mac**
  > cp -R skeleton <projectdir>

Now copy the project source files, in this case just talkingPIC.c, to <PIC32>/<projectdir>,
and change to that directory:

- **Windows**
  > copy talkingPIC.c <projectdir>
  > cd <projectdir>

- **Linux/Mac**
  > cp talkingPIC.c <projectdir>
  > cd <projectdir>

Before explaining how to use make, we will examine talkingPIC.c, which accepts input
from and prints output to a terminal emulator running on the host computer. These
capabilities facilitate user interaction and debugging. The source code for talkingPIC.c is
listed below:

---

**Code Sample 1.2 talkingPIC.c. The PIC32 Echoes Any Messages Sent to It from the
Host Keyboard Back to the Host Screen.**

```c
#include "NU32.h" // constants, funcs for startup and UART
#define MAX_MESSAGE_LENGTH 200

int main(void) {
    char message[MAX_MESSAGE_LENGTH];

    NU32_Startup(); // cache on, interrupts on, LED/button init, UART init
    while (1) {
        NU32_ReadUART3(message, MAX_MESSAGE_LENGTH); // get message from computer
        NU32_WriteUART3(message); // send message back
        NU32_WriteUART3("\r\n"); // carriage return and newline
        NU32_LED1 = !NU32_LED1; // toggle the LEDs
        NU32_LED2 = !NU32_LED2;
    }
    return 0;
}
```

---

For more information
visit www.nu32.org
Chapter 1

The NU32 library function `NU32_ReadUART3` allows the PIC32 to read data sent from your computer’s terminal emulator. The function `NU32_WriteUART3` sends data from your PIC32 to be displayed by the terminal emulator.

Now that you know how `talkingPIC.c` works, it is time to see it in action. First, make sure you are in the `<projectdir>`. Next, build the project using `make`.

```
> make
```

This command compiles and assembles all `.c` files into `.o` object files, links them into a single `out.elf` file, and turns that `out.elf` file into an executable `out.hex` file. You can do a directory listing to see all of these files.

Next, put the PIC32 into program receive mode (use the RESET and USER buttons) and execute

```
> make write
```

to invoke the bootloader utility `nu32utility` and program the PIC32 with `out.hex`. When LED1 stops flashing, the PIC32 has been programmed.

In summary, to create a new project and program the PIC32, you (1) create the project directory `<PIC32>/<projectdir>`; (2) copy the contents of `<PIC32>/skeleton` to this new directory; (3) create the source code (`talkingPIC.c` in this case) in `<projectdir>`; (4) build the executable by executing `make` in `<projectdir>`; and (5) use the RESET and USER buttons to put the PIC32 in program receive mode and execute `make write` from `<projectdir>`. To modify the program, you simply edit the source code and repeat steps (4) and (5) above. In fact, you can skip step (4), since `make write` also builds the executable before loading it onto the PIC32.

Now, to communicate with `talkingPIC`, you must connect to the PIC32 using your terminal emulator. Recall that the terminal emulator communicates with the PIC32 using `<COM>`. Enter the following command:

- **Windows**
  ```
  <puttyPath>/putty -serial <COM> -sercfg 230400,R
  ```
- **Linux/Mac**
  ```
  screen <COM> 230400,crtscts
  ```

PuTTY will launch in a new window, whereas `screen` will use the command prompt window. The number 230400 in the above commands is the baud, the speed at which the PIC32 and computer communicate, and the other parameter enables hardware flow control (see Chapter 11 for details).

For more information
visit www.nu32.org
After connecting, press RESET to restart the program. Start typing, and notice that no characters appear until you hit ENTER. This behavior may seem strange, but it occurs because the terminal emulator only displays the text it receives from the PIC32. The PIC32 does not send any text to your computer until it receives a special control character, which you generate by pressing ENTER.³

For example, if you type Hello! ENTER, the PIC32 will receive Hello!, write Hello!
 to the terminal emulator, and wait for more input.

When you are done conversing with the PIC32, you can exit the terminal emulator. To exit screen type

CTRL-a-key

Note that CTRL and a should be pressed simultaneously. To exit PuTTY make sure the command prompt window is focused and type

CTRL-c

Rather than memorizing these rather long commands to connect to the serial port, you can use the Makefile. To connect PuTTY to the PIC32 type

> make putty

To use screen type

> make screen

Your system is now configured for PIC32 programming. Although the build process may seem opaque, do not worry. For now it is only important that you can successfully compile programs and load them onto the PIC32. Later chapters will explain the details of the build process.

### 1.6 Chapter Summary

- To start a new project, copy the `<PIC32>/skeleton` directory to a new location, `<projectdir>`, and add your source code.
- From the directory `<projectdir>`, use make to build the executable.

³ Depending on the terminal emulator, ENTER may generate a carriage return (\r), newline (\n) or both. The terminal emulator typically moves the cursor to the leftmost column when it receives a \r and to the next line when it receives a \n.

For more information visit [www.nu32.org](http://www.nu32.org)
• Put the PIC32 into program receive mode by pressing the USER and RESET buttons simultaneously, then releasing the RESET button, and finally releasing the USER button. Then use `make write` to load your program.

• Use a terminal emulator to communicate with programs running on the PIC32. Typing `make putty` or `make screen` from `<projectdir>` will launch the appropriate terminal emulator and connect it to the PIC32.

Further Reading

*Embedded computing and mechatronics with the PIC32 microcontroller website. http://www.nu32.org.*
Fundamentals
Microcontrollers power the modern world: from cars to microwaves to toys. These tiny microchips integrate every component of a modern computer—a central processing unit (CPU), random access memory (RAM), nonvolatile memory (flash), and peripherals—into a single chip. Although microcontrollers have significantly less processing power than their personal computer counterparts, they are also much smaller, cost less, and use less power. Additionally, their peripherals—devices that connect the CPU with the physical world—allow software to interact with circuits directly: flashing lights, moving motors, and reading sensors.

Companies including (but certainly not limited to) Microchip, Atmel, Freescale, Texas Instruments, and STMicroelectronics manufacture an overwhelming array of microcontrollers with vastly different specifications. Rather than attempt to discuss microcontrollers generally, we focus on the PIC32MX795F512H (which we usually abbreviate as PIC32). With a fast processor, ample memory, and numerous peripherals, the PIC32MX795F512H is excellent for learning about microcontrollers and completing embedded control projects. Much of what you learn about the PIC32MX795F512H also applies more generally to the PIC32MX family of microcontrollers, and the broader concepts translate to microcontrollers more generally. Appendix C describes the differences between the PIC32MX795F512H and other PIC32 models.

### 2.1 The PIC32

#### 2.1.1 Pins, Peripherals, and Special Function Registers (SFRs)

The PIC32 requires a supply voltage between 2.3 and 3.6 V and features a maximum CPU clock frequency of 80 MHz, 512 KB of program memory (flash), and 128 KB of data memory (RAM). Its peripherals include a 10-bit analog-to-digital converter (ADC), many digital I/O pins, USB 2.0, Ethernet, two CAN modules, four I²C and three SPI synchronous serial communication modules, six UARTs for asynchronous serial communication, five 16-bit counter/timers (configurable to give two 32-bit timers and one 16-bit timer), five pulse-width modulation outputs, and several pins that can generate interrupts based on external signals. Whew. Do not worry if you do not know what all of these peripherals do, much of this book is dedicated to explaining them.
Pins connect the peripherals to the outside world. To cram so much functionality into only 64 pins, many serve multiple functions. See the pinout diagram for the PIC32MX795F512H (Figure 2.1). For example, pin 12 can be an analog input, a comparator input, a change notification input (which can generate an interrupt when an input changes state), or a digital input or output.

Table 2.1 summarizes some of the major pin functions. Other pin functions can be found in the PIC32MX5xx/6xx/7xx Data Sheet.

---

**Figure 2.1**
The pinout of the PIC32MX795F512H, the microcontroller used on the NU32 development board.

For more information visit www.nu32.org
### Table 2.1: Some of the pin functions on the PIC32

<table>
<thead>
<tr>
<th>Pin Label</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANx (x = 0 to 15)</td>
<td>Analog-to-digital (ADC) inputs</td>
</tr>
<tr>
<td>AVDD, AVSS</td>
<td>Positive supply and ground reference for ADC</td>
</tr>
<tr>
<td>CxIN-, CxIN+, CxOUT (x = 1, 2)</td>
<td>Comparator negative and positive input and output</td>
</tr>
<tr>
<td>CxRX, CxTx (x = 1, 2)</td>
<td>CAN receive and transmit pins</td>
</tr>
<tr>
<td>CLKI, CLKO</td>
<td>Clock input and output (for particular clock modes)</td>
</tr>
<tr>
<td>CNx (x = 0 to 18)</td>
<td>Change notification; voltage changes on these pins can generate interrupts</td>
</tr>
<tr>
<td>CVREF-, CVREF+, CVREFOUT</td>
<td>Comparator reference voltage low and high inputs, output</td>
</tr>
<tr>
<td>D+, D-</td>
<td>USB communication lines</td>
</tr>
<tr>
<td>ENVREG</td>
<td>Enable for on-chip voltage regulator that provides 1.8 V to internal core (on the NU32 board it is set to VDD to enable the regulator)</td>
</tr>
<tr>
<td>ICx (x = 1 to 5)</td>
<td>Input capture pins for measuring frequencies and pulse widths</td>
</tr>
<tr>
<td>INTx (x = 0 to 4)</td>
<td>Voltage changes on these pins can generate interrupts</td>
</tr>
<tr>
<td>MCLR</td>
<td>Master clear reset pin, resets PIC when low</td>
</tr>
<tr>
<td>OCx (x = 1 to 5)</td>
<td>Output compare pins, usually used to generate pulse trains (pulse-width modulation) or individual pulses</td>
</tr>
<tr>
<td>OCFA, OCFB</td>
<td>Fault protection for output compare pins; if a fault occurs, they can be used to make OC outputs be high impedance (neither high nor low)</td>
</tr>
<tr>
<td>OSC1, OSC2</td>
<td>Crystal or resonator connections for different clock modes</td>
</tr>
<tr>
<td>PMAx (x = 0 to 15)</td>
<td>Parallel master port address</td>
</tr>
<tr>
<td>PMDx (x = 0 to 7)</td>
<td>Parallel master port data</td>
</tr>
<tr>
<td>PMENB, PMRD, PMWR</td>
<td>Enable and read/write strobes for parallel master port</td>
</tr>
<tr>
<td>Rxy (x = B to G, y = 0 to 15)</td>
<td>Digital I/O pins</td>
</tr>
<tr>
<td>RTCC</td>
<td>Real-time clock alarm output</td>
</tr>
<tr>
<td>SCLx, SDAx (x = 1, 3, 4, 5)</td>
<td>(\text{I}^2\text{C}) serial clock and data input/output for (\text{I}^2\text{C}) synchronous serial communication modules</td>
</tr>
<tr>
<td>SCKx, SDIx, SDOx (x = 2 to 4)</td>
<td>Serial clock, serial data in, out for SPI synchronous serial communication modules</td>
</tr>
<tr>
<td>SSx (x = 2 to 4)</td>
<td>Slave select (active low) for SPI communication</td>
</tr>
<tr>
<td>T1CK</td>
<td>Input pin for counter/timer 1 when counting external pulses</td>
</tr>
<tr>
<td>UxCTS, UxRTS, UxRX, UxTX (x = 1 to 6)</td>
<td>UART clear to send, request to send, receive input, and transmit output for UART modules</td>
</tr>
<tr>
<td>VDD</td>
<td>Positive voltage supply for peripheral digital logic and I/O pins (3.3 V on NU32)</td>
</tr>
<tr>
<td>VDDCAP</td>
<td>Capacitor filter for internal 1.8 V regulator when ENVREG enabled</td>
</tr>
<tr>
<td>VDDCORE</td>
<td>External 1.8 V supply when ENVREG disabled</td>
</tr>
<tr>
<td>VREF-, VREF+</td>
<td>Can be used as negative and positive limit for ADC</td>
</tr>
<tr>
<td>VSS</td>
<td>Ground for logic and I/O</td>
</tr>
<tr>
<td>VBUS</td>
<td>Monitors USB bus power</td>
</tr>
<tr>
<td>USB</td>
<td>Power for USB transceiver</td>
</tr>
<tr>
<td>USBID</td>
<td>USB on-the-go (OTG) detect</td>
</tr>
</tbody>
</table>

See Section 1 of the Data Sheet for more information.
Which function a particular pin actually serves is determined by *Special Function Registers* (SFRs). Each SFR is a 32-bit word that sits at a memory address. The values of the SFR bits, 0 (cleared) or 1 (set), control the functions of the pins as well as other PIC32 behavior.

For example, pin 51 in Figure 2.1 can be OC4 (output compare 4) or RD3 (digital I/O number 3 on port D). If we wanted to use pin 51 as a digital output we would set the SFRs that control this pin to disable the OC4 functionality and enable RD3 as a digital output. The Data Sheet explains the memory addresses and meanings of the SFRs. Be careful, because it includes information for many different PIC32 models. Looking at the Data Sheet section on Output Compare reveals that the 32-bit SFR named “OC4CON” determines whether OC4 is enabled. Specifically, for bits numbered 0-31, we see that bit 15 is responsible for enabling or disabling OC4. We refer to this bit as OC4CON\(\langle 15 \rangle\). If it is cleared (0), OC4 is disabled, and if it is set (1), OC4 is enabled. So we clear this bit to 0. (Bits can be “cleared to 0” or simply “cleared,” or “set to 1” or simply “set.”) Now, referring to the I/O Ports section of the Data Sheet, we see that the input/output direction of Port D is controlled by the SFR TRISD, and bits 0-11 correspond to RD0-RD11. Bit 3 of the SFR TRISD, i.e., TRISD\(\langle 3 \rangle\), should be cleared to 0 to make RD3 (pin 51) a digital output.

According to the Memory Organization section of the Data Sheet, OC4CON\(\langle 15 \rangle\) is cleared by default on reset, so it is not necessary for our program to clear OC4CON\(\langle 15 \rangle\). On the other hand, TRISD\(\langle 3 \rangle\) is set to 1 on reset, making pin 51 a digital input by default, so the program must clear TRISD\(\langle 3 \rangle\). For safety, all pins are inputs on reset to prevent the PIC32 from imposing an unwanted voltage on external circuitry.

In addition to setting the behavior of the pins, SFRs are the primary means of communication between the PIC32’s CPU and its peripherals. You can think of a peripheral, such as a UART communication peripheral, as an independent circuit on the same chip as the CPU. Your program, running on the CPU, configures behavior of this circuit (such as the speed of UART communication) by writing bits to one or more SFRs which are read by the peripheral circuit. The CPU sends data to the peripheral (e.g., data to be sent by the UART) by writing to SFRs, and the CPU receives data from the peripheral (e.g., data received by the UART) by reading SFRs controlled by the peripheral.

We will see and use SFRs repeatedly as we learn about the PIC32.

### 2.1.2 PIC32 Architecture

**Peripherals**

Figure 2.2 depicts the PIC32’s architecture. Of course there is a CPU, program memory (flash), and data memory (RAM). Perhaps most interesting to us, though, are the *peripherals*, which make microcontrollers useful for embedded control. We briefly discuss the available peripherals here; subsequent chapters cover them in detail. The peripherals are listed roughly in top to bottom, left to right order, as they appear in Figure 2.2.
Figure 2.2
The PIC32MX5XX/6XX/7XX architecture. The PIC32MX795F512H is missing the digital I/O PORTA and has only 19 change notification inputs, 3 SPI modules, and 4 I²C modules.

Digital input and output

Digital I/O ports (PORTB to PORTG on the PIC32MX795F512H) allow you to read or output a digital voltage. A digital I/O pin configured as an input can detect whether the input voltage is low or high. On the NU32, the PIC32 is powered by 3.3 V, so voltages close to 0 V are considered low and those close to 3.3 V are considered high. Some input pins can tolerate up
to 5.5 V, while voltages over 3.3 V on other pins could damage the PIC32 (see Figure 2.1 for the pins that can tolerate 5.5 V).

A digital I/O pin configured as an output can produce a voltage of 0 or 3.3 V. An output pin can also be configured as open drain. In this configuration, the pin is connected by an external pull-up resistor to a voltage of up to 5.5 V. This allows the pin’s output transistor to either sink current (to pull the voltage down to 0 V) or turn off (allowing the voltage to be pulled up as high as 5.5 V), increasing the range of output voltages the pin can produce.

Universal Serial Bus

The Universal Serial Bus (USB) is an asynchronous communication protocol heavily used by computers and other devices. One master communicates with one or more slaves over a four-line bus: +5 V, ground, D+, and D− (differential data signals). The PIC32 has a single USB peripheral implementing USB 2.0 full-speed and low-speed options, and can communicate at theoretical data rates of up to several megabits per second.

Controller area network

Controller area network (CAN) is pervasive in industrial and automotive applications, where electrical noise can be problematic. CAN allows many devices to communicate over a single two-wire bus. Data rates of up to 1 megabit per second are possible. The CAN peripheral uses an external transceiver chip to convert between signals on the bus and signals that the PIC32 can process. The PIC32 contains two CAN modules.

Ethernet

The Ethernet module allows the PIC32 to connect to the Internet. It uses an external physical layer protocol transceiver (PHY) chip and direct memory access (DMA) to offload the heavy processing requirements of Ethernet communication from the CPU. The NU32 board does not include a PHY chip.

DMA controller

The direct memory access (DMA) controller (DMAC) transfers data without involving the CPU. For example, DMA can allow an external device to dump data through a UART directly into PIC32 RAM.

In-Circuit Debugger

The In-Circuit Debugger (ICD) is used by Microchip debugging tools to control the PIC32’s operation during debugging.

For more information visit www.nu32.org
Watchdog timer

If the watchdog timer (WDT) is used by your program, your program must periodically reset a counter. Otherwise, when the counter reaches a specified value, the PIC32 will reset. The WDT allows the PIC32 to recover from an unexpected state or infinite loop caused by software errors.

Change notification

A change notification (CN) pin can be used to generate an interrupt when the input voltage changes from low to high or vice-versa. The PIC32 has 19 change notification pins (CN0 to CN18).

Counter/timers

The PIC32 has five 16-bit counters/timers (Timer1 to Timer5). A counter counts the number of pulses of a signal. If the pulses occur at a regular frequency, the count can be used as a time; hence timers are just counters with inputs at a fixed frequency. Microchip uniformly refers to these devices as “timers”, so we adopt that terminology from now on. Each timer can count from 0 up to $2^{16} - 1$, or any preset value less than $2^{16} - 1$ that we choose, before rolling over. Timers can count external events, such as pulses on the T1CK pin, or internal pulses on the peripheral bus clock. Two 16-bit timers can be configured to make a single 32-bit timer. Two different pairs of timers can be combined, yielding one 16-bit and two 32-bit timers.

Output compare

The five output compare (OC) pins (OC1 to OC5) are used to generate a single pulse of specified duration, or a continuous pulse train of specified duty cycle and frequency. They work with timers to generate pulses with precise timing. Output compare is commonly used to generate PWM (pulse-width modulated) signals that can control motors or be low-pass filtered to create a specified analog voltage output. (You cannot output an arbitrary analog voltage from the PIC32.)

Input capture

The five input capture (IC) pins (IC1 to IC5) store the current time, as measured by a timer, when an input changes. Thus, this peripheral allows precise measurements of input pulse widths and signal frequencies.

Serial Peripheral Interface

The PIC32 has three Serial Peripheral Interface (SPI) peripherals (SPI2 to SPI4). The SPI bus provides a method for synchronous serial communication between a master device (typically a
microcontroller) and one or more slave devices. The interface typically requires four communication pins: a clock (SCK), data in (SDI), data out (SDO), and slave select (SS). Communication can occur at up to tens of megabits per second.

Inter-integrated circuit

The PIC32 has four inter-integrated circuit (I²C) modules (I2C1, I2C3, I2C4, I2C5). I²C (pronounced “I squared C”) is a synchronous serial communication standard (like SPI) that allows several devices to communicate over only two wires. Any device can be the master and control communication at any given time. The maximum data rate is less than for SPI, usually 100 or 400 kilobits per second.

Parallel master port

The parallel master port (PMP) module is used to read data from and write data to external parallel devices. Parallel communication allows multiple data bits to be transferred simultaneously, but each bit requires its own wire.

Analog input

The PIC32 has one analog-to-digital converter (ADC), but 16 different pins can be connected to it, allowing up to 16 analog voltage values (typically sensor inputs) to be monitored. The ADC can be programmed to continuously read data from a sequence of input pins, or to read a single value. Input voltages must be between 0 and 3.3 V. The ADC has 10 bits of resolution, allowing it to distinguish $2^{10} = 1024$ different voltage levels. Conversions are theoretically possible at a maximum rate of 1 million samples per second.

Universal asynchronous receiver/transmitter

The PIC32 has six universal asynchronous receiver transmitter (UART) modules (UART1 to UART6). These peripherals provide another method for serial communication between two devices. Unlike synchronous serial protocols such as SPI, the UART has no clock line; rather the devices communicating each have their own clocks that must operate at the same frequency. Each of the two devices participating in UART communication has, at minimum, a receive (RX) and transmit (TX) line. Often request to send (RTS) and clear to send (CTS) lines are used as well, allowing the devices to coordinate when to send data. Typical data rates are 9600 bits per second (9600 baud) up to hundreds of thousands of bits per second. The talkingPIC.c program uses a UART configured to operate at 230,400 baud to communicate with your computer.

Real-time clock and calendar

The real-time clock and calendar (RTCC) module maintains accurate time, in seconds, minutes, days, months, and years, over extended periods of time.

For more information visit www.nu32.org
Comparators

The PIC32 has two comparators, each of which compares two analog input voltages and determines which is larger. A configurable internal voltage reference may be used in the comparison, or even output to a pin, resulting in a limited-resolution digital-to-analog converter.

Other components

Note that the peripherals are on two different buses: one is clocked by the system clock SYSCLK, and the other is clocked by the peripheral bus clock PBCLK. A third clock, USBCLK, is used for USB communication. The timing generation block that creates these clock signals and other elements of the architecture in Figure 2.2 are briefly described below.

CPU

The central processing unit runs everything. It fetches program instructions over its “instruction side” (IS) bus, reads data over its “data side” (DS) bus, executes the instructions, and writes the results over the DS bus. The CPU can be clocked by SYSCLK at up to 80 MHz, meaning it can execute one instruction every 12.5 ns. The CPU is capable of multiplying a 32-bit integer by a 16-bit integer in one cycle, or a 32-bit integer by a 32-bit integer in two cycles. There is no floating point unit (FPU), so floating point math is carried out by software algorithms, making floating point operations much slower than integer math.

The CPU is the MIPS32® M4K® microprocessor core, licensed from Imagination Technologies. The CPU operates at 1.8 V (provided by a voltage regulator internal to the PIC32, as it’s used on the NU32 board). The interrupt controller, discussed below, can notify the CPU about external events.

Bus matrix

The CPU communicates with other units through the 32-bit bus matrix. Depending on the memory address specified by the CPU, the CPU can read data from, or write data to, program memory (flash), data memory (RAM), or SFRs. The memory map is discussed in Section 2.1.3.

Interrupt controller

The interrupt controller presents “interrupt requests” to the CPU. An interrupt request (IRQ) may be generated by a variety of sources, such as a changing input on a change notification pin or by the elapsing of a specified time on one of the timers. If the CPU accepts the request, it will suspend whatever it is doing and jump to an interrupt service routine (ISR), a function defined in the program. After completing the ISR, program control returns to where it was suspended. Interrupts are an extremely important concept in embedded control and are discussed thoroughly in Chapter 6.

For more information visit www.nu32.org
Memory: Program flash and data RAM

The PIC32 has two types of memory: flash and RAM. Flash is generally more plentiful on PIC32’s (e.g., 512 KB flash vs. 128 KB RAM on the PIC32MX795F512H), nonvolatile (meaning that its contents are preserved when powered off, unlike RAM), but slower to read and write than RAM. Your program is stored in flash memory and your temporary data is stored in RAM. When you power cycle the PIC32, your program is still there but your data in RAM is lost.1

Because flash is slow, with a max access speed of 30 MHz for the PIC32MX795F512H, reading a program instruction from flash may take three CPU cycles when operating at 80 MHz (see Electrical Characteristics in the Data Sheet). The prefetch cache module (described below) can minimize or eliminate the need for the CPU to wait for program instructions to load from flash.

Prefetch cache module

You might be familiar with the term cache from your web browser. Your browser’s cache stores recent documents or pages you have accessed, so the next time you request them, your browser can provide a local copy immediately, instead of waiting for the download.

The prefetch cache module operates similarly—it stores recently executed program instructions, which are likely to be executed again soon (as in a program loop), and, in linear code with no branches, it can even run ahead of the current instruction and predictively prefetch future instructions into the cache. In both cases, the goal is to have the next instruction requested by the CPU already in the cache. When the CPU requests an instruction, the cache is first checked. If the instruction at that memory address is in the cache (a cache hit), the prefetch module provides the instruction to the CPU immediately. If there is a miss, the slower load from flash memory begins.

In some cases, the prefetch module can provide the CPU with one instruction per cycle, hiding the delays due to slow flash access. The module can cache all instructions in small program loops, so that flash memory does not have to be accessed while executing the loop. For linear code, the 128-bit wide data path between the prefetch module and flash memory allows the prefetch module to run ahead of execution despite the slow flash load times.

The prefetch cache module can also store constant data.

Clocks and timing generation

There are three clocks on the PIC32: SYSCLK, PBCLK, and USBCLK. USBCLK is a 48 MHz clock used for USB communication. SYSCLK clocks the CPU at a maximum

---

1 It is also possible to store program instructions in RAM, and to store data in flash, but we ignore that for now.
frequency of 80 MHz, adjustable down to 0 Hz. Higher frequency means more calculations per second but higher power usage (approximately proportional to frequency). PBCLK is used by many peripherals, and its frequency is set to SYSCLK’s frequency divided by 1, 2, 4, or 8. You might want to set PBCLK’s frequency lower than SYSCLK’s if you want to save power. If PBCLK’s frequency is less than SYSCLK’s, then programs with back-to-back peripheral operations will cause the CPU to wait a few cycles before issuing the second peripheral command to ensure that the first one has completed.

All clocks are derived either from an oscillator internal to the PIC32 or an external resonator or oscillator provided by the user. High-speed operation requires an external circuit, so the NU32 provides an external 8 MHz resonator as a clock source. The NU32 software sets the PIC32’s configuration bits (see Section 2.1.4) to use a phase-locked loop (PLL) on the PIC32 to multiply this frequency by a factor of 10, generating a SYSCLK of 80 MHz. The PBCLK is set to the same frequency. The USBCLK is also derived from the 8 MHz resonator by multiplying the frequency by 6.

2.1.3 The Physical Memory Map

The CPU accesses peripherals, data, and program instructions in the same way: by writing a memory address to the bus. The PIC32’s memory addresses are 32-bits long, and each address refers to a byte in the memory map. Thus, the PIC32’s memory map consists of 4 GB (four gigabytes, or $2^{32}$ bytes). Of course most of these addresses are meaningless; there are far more addresses than needed.

The PIC32’s memory map consists of four main components: RAM, flash, peripheral SFRs that we write to (to control the peripherals or send outputs) or read from (to get sensor input, for example), and boot flash. Of these, we have not yet seen “boot flash.” This extra flash memory, 12 KB on the PIC32MX795F512H, contains program instructions that are executed immediately upon reset. The boot flash instructions typically perform PIC32 initialization and then call the program installed in program flash. For the PIC32 on the NU32 board, the boot flash contains a “bootloader” program that communicates with your computer when you load a new program on the PIC32 (see Chapter 3).

The following table illustrates the PIC32’s physical memory map. It consists of a block of “RAMsize” bytes of RAM (128 KB for the PIC32MX795F512H), “flashsize” bytes of flash (512 KB for the PIC32MX795F512H), 1 MB for the peripheral SFRs, and “bootsize” for the boot flash (12 KB for the PIC32MX795F512H):

---

2 The last four 32-bit words of the boot flash memory region are Device Configuration Registers (see Section 2.1.4).
### 2.1.4 Configuration Bits

The last four 32-bit words of the boot flash are the Device Configuration Registers, DEVCFG0 to DEVCFG3, containing the *configuration bits*. The values in these configuration bits determine important properties of how the PIC32 will function. You can learn more about configuration bits in the Special Features section of the Data Sheet. For example, DEVCFG1 and DEVCFG2 contain configuration bits that determine the frequency multiplier converting the external resonator frequency to the SYSCLK frequency, as well as bits that determine the ratio between the SYSCLK and PBCLK frequencies. On the NU32 board (below), the PIC32’s configuration bits were programmed along with the bootloader.

### 2.2 The NU32 Development Board

The NU32 development board is shown in *Figure 1.1*, and the pinout is given in *Table 2.2*. The NU32 board provides easy breadboard access to most of the PIC32MX795F512H’s 64 pins. The NU32 acts like a big 60-pin DIP (dual in-line package) chip and plugs into a standard prototyping breadboard as shown in *Figure 1.1*. More details and the latest information on the NU32 can be found on the book’s website.

Beyond simply breaking out the pins, the NU32 provides many features that make it easy to get started with the PIC32. For example, to power the PIC32, the NU32 provides a barrel jack that accepts a 1.35 mm inner diameter, 3.5 mm outer diameter center-positive power plug. The plug should provide 1 A at DC 6 V or more. The PIC32 requires a supply voltage VDD between 2.3 and 3.6 V, and the NU32 provides a 3.3 V voltage regulator providing a stable voltage source for the PIC32 and other electronics on board. Since it is often convenient to have a 5 V supply available, the NU32 also has a 5 V regulator. The power plug’s raw input

---

<table>
<thead>
<tr>
<th>Physical Memory Start Address</th>
<th>Size (bytes)</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>RAMsize (128 KB)</td>
<td>Data RAM</td>
</tr>
<tr>
<td>0x1D000000</td>
<td>flashsize (512 KB)</td>
<td>Program Flash</td>
</tr>
<tr>
<td>0x1F800000</td>
<td>1 MB</td>
<td>Peripheral SFRs</td>
</tr>
<tr>
<td>0x1FC000000</td>
<td>bootsize (12 KB)</td>
<td>Boot Flash</td>
</tr>
</tbody>
</table>

The memory regions are not contiguous. For example, the first address of program flash is 480 MB after the first address of data RAM. An attempt to access an address between the data RAM segment and the program flash segment would generate an error.

It is also possible to allocate a portion of RAM to hold program instructions.

In *Chapter 3*, when we discuss programming the PIC32, we will introduce the *virtual* memory map and its relationship to the physical memory map.
### Table 2.2: The NU32 pinout (in gray, with power jack at top) with PIC32MX795F512H pin numbers

<table>
<thead>
<tr>
<th>Function</th>
<th>PIC32</th>
<th>Function</th>
<th>PIC32</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>GND</td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>3.3 V</td>
<td>3.3 V</td>
<td>3.3 V</td>
<td>3.3 V</td>
</tr>
<tr>
<td>5 V</td>
<td>5 V</td>
<td>5 V</td>
<td>5 V</td>
</tr>
<tr>
<td>VIN</td>
<td>VIN</td>
<td>VIN</td>
<td>VIN</td>
</tr>
<tr>
<td>C1RX/RF0</td>
<td>F0</td>
<td>C1TX/RF1</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td></td>
<td>59</td>
</tr>
<tr>
<td>PMD0/RE0</td>
<td>E0</td>
<td>PMD1/RE1</td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>G8</td>
<td></td>
<td>G7</td>
</tr>
<tr>
<td>PMD2/RE2</td>
<td>E2</td>
<td></td>
<td>G6</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>PMD3/RE3</td>
<td>E3</td>
<td>PMD4/RE4</td>
<td>E4</td>
</tr>
<tr>
<td></td>
<td>MCLR</td>
<td></td>
<td>D7</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>AN0/PMA6/CN2/RB0</td>
<td>B0</td>
<td>AN1/CN3/RB1</td>
<td>B1</td>
</tr>
<tr>
<td></td>
<td>D3</td>
<td></td>
<td>D2</td>
</tr>
<tr>
<td>AN2/C2IN-/CN4/ RB2</td>
<td>B2</td>
<td>AN3/C2IN+/CN5/ RB3</td>
<td>B3</td>
</tr>
<tr>
<td></td>
<td>D1</td>
<td></td>
<td>D0</td>
</tr>
<tr>
<td>AN4/C2IN-/CN6/ RB4</td>
<td>B4</td>
<td>AN5/C2IN+/CN7/ RB5</td>
<td>B5</td>
</tr>
<tr>
<td></td>
<td>C14</td>
<td></td>
<td>C13</td>
</tr>
<tr>
<td>AN6/OCFA/RB6</td>
<td>B6</td>
<td>AN7/RB7</td>
<td>B7</td>
</tr>
<tr>
<td></td>
<td>D11</td>
<td></td>
<td>D10</td>
</tr>
<tr>
<td>AN8/C2TX/U5RX/ U2CTS/RB8</td>
<td>B8</td>
<td>AN9/PMA7/RB9</td>
<td>B9</td>
</tr>
<tr>
<td></td>
<td>D9</td>
<td></td>
<td>D8</td>
</tr>
<tr>
<td>AN10/PMA13/RB10</td>
<td>B10</td>
<td>AN11/PMA12/RB11</td>
<td>B11</td>
</tr>
<tr>
<td></td>
<td>G2</td>
<td></td>
<td>G3</td>
</tr>
<tr>
<td>AN12/PMA11/RB12</td>
<td>B12</td>
<td>AN13/PMA10/RB13</td>
<td>B13</td>
</tr>
<tr>
<td></td>
<td>VBUS</td>
<td></td>
<td>F3</td>
</tr>
<tr>
<td>AN14/C2RX/SCK4/U5TX/U2RTS/PMA1/RB14</td>
<td>B14</td>
<td>AN15/OCFB/PMA0/ CN12/RB15</td>
<td>B15</td>
</tr>
<tr>
<td></td>
<td>F4</td>
<td></td>
<td>F5</td>
</tr>
<tr>
<td></td>
<td>SCL5/SDO4/U2TX/PMA8/CN18/RF5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pins marked with a √ are 5.5 V tolerant. Not all pin functions are listed; see Figure 2.1 or the PIC32 Data Sheet. Board pins in **bold** should only be used with care, as they are shared with other functions on the NU32. In particular, the NU32 pins G6, G7, G8, G9, F0, F1, D7, and MCLR should be considered outputs during normal usage. The value of MCLR is determined by the MCLR button on the NU32; the value of D7 is determined by the USER button; F0 and F1 are used by the PIC32 as digital outputs to control LED1 and LED2 on the NU32, respectively; and G6 through G9 are used by the PIC32’s UART3 for communication with the host computer through the mini-B USB jack.

For more information
visit www.nu32.org
voltage Vin and ground, as well as the regulated 3.3 V and 5 V supplies, are made available to the user as illustrated in Figure 1.1. The power jack is directly connected to the Vin and GND pins so you could power the NU32 by putting Vin and GND on these pins directly and not connecting the power jack.

The 3.3 V regulator provides up to 800 mA and the 5 V regulator provides up to 1 A of current, provided the power supply can source that much current. In practice you should stay well under each of these limits. For example, you should not plan to draw more than 200-300 mA or so from the NU32. Even if you use a higher-current power supply, such as a battery, you should respect these limits, as the current has to flow through the relatively thin traces of the PCB. It is also not recommended to use high voltage supplies greater than 9 V or so, as the regulators will heat up.

Since motors tend to draw lots of current (even small motors may draw hundreds of milliamps up to several amps), do not try to power them from the NU32. Use a separate battery or power supply instead.

In addition to the voltage regulators, the NU32 provides an 8 MHz resonator as the source of the PIC32’s 80 MHz clock signal. It also has a mini-B USB jack to connect your computer’s USB port to a USB-to-UART FTDI chip that allows your PIC32 to use its UART to communicate with your computer.

A USB micro-B jack is provided to allow the PIC32 to speak USB to another external device, like a smartphone.

The NU32 board also has a power switch which connects or disconnect the input power supply to the voltage regulators, and two LEDs and two buttons (labeled USER and RESET) allowing very simple input and output. The two LEDs, LED1 and LED2, are connected at one end by a resistor to 3.3 V and the other end to digital outputs RF0 and RF1, respectively, so that they are off when those outputs are high and on when they are low. The USER and RESET buttons are attached to the digital input RD7 and MCLR pins, respectively, and both buttons are configured to give 0 V to these inputs when pressed and 3.3 V otherwise. See Figure 2.3.

Because pins RG6 through RG9, RF0, RF1, and RD7 on the PIC32 are used for UART communication with the host computer, LEDs, and the USER button, other potential functions of these pins are not available if you would like to use the communication, LEDs, and USER button. In particular:

- UART6 is unavailable (conflicts with pins RG6 and RG9). Since UART3 is used for communication with the host computer, this leaves UART1, UART2, UART4, and UART5 for your programs.
- SPI2 is unavailable (conflicts with pins RG6 and RG7). This leaves SPI3 and SPI4.
- I2C4 is unavailable (conflicts with pins RG7 and RG8). This leaves I2C1, I2C3, and I2C5.
• The default CAN1 pins C1RX and C1TX cannot be used (they conflict with pins RF0 and RF1), but the configuration bit FCANIO in DEVCFG3 has been cleared to zero on the NU32, thereby setting CAN1 to use the alternate pins AC1RX (RF4) and AC1TX (RF5). Therefore no CAN module is lost.

• Media-independent interface (MII) Ethernet is unavailable (conflicts with pins RD7, RF0, and RF1). The PIC32 can implement Ethernet communication using either the MII or the reduced media-independent interface (RMII), and RMII Ethernet communication, which uses many fewer pins than MII, is still available on the NU32.

• Several change notification and digital I/O pins are unavailable, but many more remain.

In all, very little functionality is unavailable due to connections on the NU32, and advanced users can find ways to bypass even these limitations.

Although the NU32 comes with a bootloader installed in its flash memory, you have the option to use a programmer to install a standalone program. The five plated through-holes on the USB board align with the pins of devices such as the PICkit 3 programmer (Figure 2.4).

![Figure 2.3](image1.png)

**Figure 2.3**
The NU32 connection of the PIC32 pins RF0, RF1, and RD7 to LED1, LED2, and the USER button, respectively.

![Figure 2.4](image2.png)

**Figure 2.4**
Attaching the PICkit 3 programmer to the NU32 board.

For more information visit [www.nu32.org](http://www.nu32.org)
2.3 Chapter Summary

- The PIC32 features a 32-bit data bus and a CPU capable of performing some 32-bit operations in a single clock cycle.
- In addition to nonvolatile flash program memory and RAM data memory, the PIC32 provides peripherals particularly useful for embedded control, including analog inputs, digital I/O, PWM outputs, counter/timers, inputs that generate interrupts or measure pulse widths or frequencies, and pins for a variety of communication protocols, including USB, Ethernet, CAN, I^2C, and SPI.
- The functions performed by the pins and peripherals are determined by Special Function Registers. SFRs are also used for communication back and forth between the CPU and peripherals.
- The PIC32 has three main clocks: the SYSCLK that clocks the CPU, the PBCLK that clocks peripherals, and the USBCLK that clocks USB communication.
- Physical memory addresses are specified by 32 bits. The physical memory map contains four regions: data RAM, program flash, SFRs, and boot flash. RAM can be accessed in one clock cycle, while flash access may be slower. The prefetch cache module can be used to minimize delays in accessing program instructions.
- Four 32-bit configuration words, DEVCFG0 to DEVCFG3, set important behavior of the PIC32. For example, these configuration bits determine how an external clock frequency is multiplied or divided to create the PIC32 clocks.
- The NU32 development board provides voltage regulators for power, includes a resonator for clocking, breaks out the PIC32 pins to a solderless breadboard, provides a couple of LEDs and buttons for simple input and output, and simplifies communication with the PIC32 via your computer’s USB port.

2.4 Exercises

You will need to refer to the PIC32MX5XX/6XX/7XX Data Sheet and PIC32 Reference Manual to answer some questions.

1. Search for a listing of PIC32 products on Microchip’s webpage, showing the specifications of all the PIC32 models.
   a. Find PIC32s that meet the following specs: at least 128 KB of flash, at least 32 KB of RAM, and at least 80 MHz max CPU speed. What is the cheapest PIC32 that meets these specs, and what is its volume price? How many ADC, UART, SPI, and I^2C channels does it have? How many timers?
   b. What is the cheapest PIC32 overall? How much flash and RAM does it have, and what is its maximum clock speed?
   c. Among all PIC32s with 512 KB flash and 128 KB RAM, which is the cheapest? How does it differ from the PIC32MX795F512H?

For more information visit www.nu32.org
2. Based on C syntax for bitwise operators and bit-shifting, calculate the following and give your results in hexadecimal.
   a. \(0x37 \mid 0xA8\)
   b. \(0x37 \& 0xA8\)
   c. \(\sim 0x37\)
   d. \(0x37 >> 3\)

3. Describe the four functions that pin 12 of the PIC32MX795F512H can have. Is it 5 V tolerant?

4. Referring to the Data Sheet section on I/O Ports, what is the name of the SFR you have to modify if you want to change pins on PORTC from output to input?

5. The SFR CM1CON controls comparator behavior. Referring to the Memory Organization section of the Data Sheet, what is the reset value of CM1CON in hexadecimal?

6. In one sentence each, without going into detail, explain the basic function of the following items shown in the PIC32 architecture block diagram Figure 2.2: SYSCLK, PBCLK, PORTA to PORTG (and indicate which of these can be used for analog input on the NU32’s PIC32), Timer1 to Timer5, 10-bit ADC, PWM OC1-5, Data RAM, Program Flash Memory, and Prefetch Cache Module.

7. List the peripherals that are not clocked by PBCLK.

8. If the ADC is measuring values between 0 and 3.3 V, what is the largest voltage difference that it may not be able to detect? (It’s a 10-bit ADC.)

9. Refer to the Reference Manual chapter on the Prefetch Cache. What is the maximum size of a program loop, in bytes, that can be completely stored in the cache?

10. Explain why the path between flash memory and the prefetch cache module is 128 bits wide instead of 32, 64, or 256 bits.

11. Explain how a digital output could be configured to swing between 0 and 4 V, even though the PIC32 is powered by 3.3 V.

12. PIC32’s have increased their flash and RAM over the years. What is the maximum amount of flash memory a PIC32 can have before the current choice of base addresses in the physical memory map (for RAM, flash, peripherals, and boot flash) would have to be changed? What is the maximum amount of RAM? Give your answers in bytes in hexadecimal.

13. Examine the Special Features section of the Data Sheet.
   a. If you want your PBCLK frequency to be half the frequency of SYSCLK, which bits of which Device Configuration Register do you have to modify? What values do you give those bits?
   b. Which bit(s) of which SFR set the watchdog timer to be enabled? Which bit(s) set the postscale that determines the time interval during which the watchdog must be reset to prevent it from restarting the PIC32? What values would you give these bits to enable the watchdog and to set the time interval to be the maximum?
c. The SYSCLK for a PIC32 can be generated several ways, as discussed in the Oscillator chapter in the Reference Manual and the Oscillator Configuration section in the Data Sheet. The PIC32 on the NU32 uses the (external) primary oscillator in HS mode with the phase-locked loop (PLL) module. Which bits of which device configuration register enable the primary oscillator and turn on the PLL module?

14. Your NU32 board provides four power rails: GND, regulated 3.3 V, regulated 5 V, and the unregulated input voltage (e.g., 6 V). You plan to put a load from the 5 V output to ground. If the load is modeled as a resistor, what is the smallest resistance that would be safe? An approximate answer is fine. In a sentence, explain how you arrived at the answer.

15. The NU32 could be powered by different voltages. Give a reasonable range of voltages that could be used, minimum to maximum, and explain the reason for the limits.

16. Two buttons and two LEDs are interfaced to the PIC32 on the NU32. Which pins are they connected to? Give the actual pin numbers, 1-64, as well as the name of the pin function as it is used on the NU32. For example, pin 37 on the PIC32MX795F512H could have the function D+ (USB data line) or RG2 (Port G digital input/output), but only one of these functions could be active at a given time.

Further Reading

PIC32MX5XX/6XX/7XX family data sheet. (2013). Microchip Technology Inc.
Most electric motors operate on the principle that current flowing through a magnetic field creates a force. Because of this relationship between current and force, electric motors can be used to convert electrical power to mechanical power. They can also be used to convert mechanical power to electrical power; as with, for example, generators in hydroelectric dams or regenerative braking in electric and hybrid cars.

In this chapter we study perhaps the simplest, cheapest, most common, and arguably most useful electrical motor: the brushed permanent magnet direct current (DC) motor. For brevity, we refer to these simply as DC motors. A DC motor has two input terminals, and a voltage applied across these terminals causes the motor shaft to spin. For a constant load or resistance at the motor shaft, the motor shaft achieves a speed proportional to the input voltage. Positive voltage causes spinning in one direction, and negative voltage causes spinning in the other.

Depending on the specifications, DC motors cost anywhere from tens of cents up to thousands of dollars. For most small-scale or hobby applications, appropriate DC motors typically cost a few dollars. DC motors are often outfitted with a sensing device, most commonly an encoder, to track the position and speed of the motor, and a gearhead to reduce the output speed and increase the output torque.

### 25.1 Motor Physics

DC motors exploit the Lorentz force law,

\[ \mathbf{F} = \ell \mathbf{I} \times \mathbf{B}, \]

where \( \mathbf{F}, \mathbf{I}, \) and \( \mathbf{B} \) are three-vectors, \( \mathbf{B} \) describes the magnetic field created by permanent magnets, \( \mathbf{I} \) is the current vector (including the magnitude and direction of the current flow through the conductor), \( \ell \) is the length of the conductor in the magnetic field, and \( \mathbf{F} \) is the force on the conductor. For the case of a current perpendicular to the magnetic field, the force is easily understood using the right-hand rule for cross-products: with your right hand, point your index finger along the current direction and your middle finger along the magnetic field flux lines. Your thumb will then point in the direction of the force (see Figure 25.1).
Two magnets create a magnetic field \( B \), and a current \( I \) along the conductor causes a force \( F \) on the conductor.

Now let us replace the conductor by a loop of wire, and constrain that loop to rotate about its center. See Figures 25.2 and 25.3. In one half of the loop, the current flows into the page, and in the other half of the loop the current flows out of the page. This creates forces of opposite directions on the loop. Referring to Figure 25.3, let the magnitude of the force acting on each half of the loop be \( f \), and let \( d \) be the distance from the halves of the loop to the center of the loop. Then the total torque acting on the loop about its center can be written

\[
\tau = 2df \cos \theta,
\]

where \( \theta \) is the angle of the loop. The torque changes as a function of \( \theta \). For \(-90^\circ < \theta < 90^\circ\), the torque is positive, and it is maximum at \( \theta = 0 \). A plot of the torque on the loop as a function of \( \theta \) is shown in Figure 25.4(a). The torque is zero at \( \theta = -90^\circ \) and \( 90^\circ \), and of these two, \( \theta = 90^\circ \) is a stable equilibrium while \( \theta = -90^\circ \) is an unstable equilibrium. Therefore, if we send a constant current through the loop, it will likely come to rest at \( \theta = 90^\circ \).
A loop of wire in a magnetic field, viewed end-on. Current flows into the page on one side of the loop and out of the page on the other, creating forces of opposite directions on the two halves of the loop. These opposite forces create torque on the loop about its center at most angles $\theta$ of the loop.

(a) The torque on the loop of Figure 25.3 as a function of its angle for a constant current. (b) If we reverse the current direction at the angles $\theta = -90^\circ$ and $\theta = 90^\circ$, we can make the torque nonnegative at all $\theta$. (c) If we use several loops offset from each other, the sum of their torques (the thick curve) becomes more constant as a function of angle. The remaining variation contributes to torque ripple.

To make a more useful motor, we can reverse the direction of current at $\theta = -90^\circ$ and $\theta = 90^\circ$, which makes the torque nonnegative at all angles (Figure 25.4(b)). The torque is still zero at $\theta = -90^\circ$ and $\theta = 90^\circ$, however, and it undergoes a large variation as a function of $\theta$. To make the torque more constant as a function of $\theta$, we can introduce more loops of wire, each offset from the others in angle, and each reversing their current direction at appropriate angles. Figure 25.4(c) shows an example with three loops of wire offset from each other by $120^\circ$. Their component torques sum to give a more constant torque as a function of angle. The remaining variation in torque contributes to angle-dependent torque ripple.
Finally, to increase the torque generated, each loop of wire is replaced by a coil of wire (also called a winding) that loops back and forth through the magnetic field many times. If the coil consists of 100 loops, it generates 100 times the torque of the single loop for the same current. Wire used to create coils in motors, like magnet wire, is very thin, so there is resistance from one end of a coil to the other, typically from fractions of an ohm up to hundreds of ohms.

As stated previously, the current in the coils must switch direction at the appropriate angle to maintain non-negative torque. Figure 25.5 shows how brushed DC motors accomplish this current reversal. The two input terminals are connected to brushes, typically made of a soft conducting material like graphite, which are spring-loaded to press against the commutator, which is connected to the motor coils. As the motor rotates, the brushes slide over the commutator and switch between commutator segments, each of which is electrically connected to the end of one or more coils. This switching changes the direction of current through the coils. This process of switching the current through the coils as a function of the angle of the motor is called commutation. Figure 25.5 shows a schematic of a minimal motor design with three commutator segments and a coil between each pair of segments. Most high quality motors have more commutator segments and coils.

Unlike the simplified example in Figure 25.4, the brush-commutator geometry means that each coil in a real brushed motor is only energized at a subset of angles of the motor. Apart

**Figure 25.5**

(Left) A schematic end-on view of a simple DC motor. The two brushes are held against the commutator by leaf springs which are electrically connected to the external motor terminals. This commutator has three segments and there are coils between each segment pair. The stator magnets are epoxied to the inside of the motor housing. (Right) This disassembled Pittman motor has seven commutator segments. The two brushes are attached to the motor housing, which has otherwise been removed. One of the two permanent magnets is visible inside the housing. The coils are wrapped around a ferromagnetic core to increase magnetic permeability. This motor has a gearhead on the output.
Figure 25.4(c) illustrates the sum of the torque of three coils offset by 120° if they are all energized at the same time. The geometry of the brushes and commutator ensure that not all coils are energized simultaneously, however. This figure shows the angle-dependent torque of a three-coil brushed motor that has only one coil energized at a time, which is approximately what happens if the brushes in Figure 25.5 are small. The energized coil is the one at the best angle to create a torque. The result is a motor torque as indicated by the thick curve; the thinner curves are the torques that would be provided by the other coils if they were energized. Comparing this figure to Figure 25.4(c) shows that this more realistic motor produces half the torque, but uses only one-third of the electrical power, since only one of the three coils is energized. Power is not wasted by putting current through coils that would generate little torque.

from being a consequence of the geometry, this has the added benefit of avoiding wasting power when current through a coil would provide little torque. Figure 25.6 is a more realistic version of Figure 25.4(c).

The stationary portion of the motor attached to the housing is called the stator, and the rotating portion of the motor is called the rotor.

Figure 25.7 shows a cutaway of a Maxon brushed motor, exposing the brushes, commutator, magnets, and windings. The figure also shows other elements of a typical motor application: an encoder attached to one end of the motor shaft to provide feedback on the angle and a gearhead attached to the other end of the motor shaft. The output shaft of the gearhead provides lower speed but higher torque than the output shaft of the motor.

*Brushless* motors are a variant that use electronic commutation as opposed to brushed commutation. For more on brushless DC motors, see Chapter 29.5.

### 25.2 Governing Equations

To derive an equation to model the motor’s behavior, we ignore the details of the commutation and focus instead on electrical and mechanical power. The electrical power into the motor is $IV$, where $I$ is the current through the motor and $V$ is the voltage across the motor. We know that the motor converts some of this input power to mechanical power $\tau \omega$, where $\tau$
and $\omega$ are the torque and velocity of the output shaft, respectively. Electrically, the motor is described by a resistance $R$ between the two terminals as well as an inductance $L$ due to the coils. The resistance of the motor coils dissipates power $I^2R$ as heat. The motor also stores energy $\frac{1}{2}LI^2$ in the inductor’s magnetic field, and the time rate of change of this is $LI\frac{dI}{dt}$, the power into (charging) or out of (discharging) the inductor. Finally, power is dissipated as sound, heat due to friction at the brush-commutator interface and at the bearings between the motor shaft and the housing, etc. In SI units, all these power components are expressed in watts. Combining all of these factors provides a full accounting for the electrical power put into the motor:

$$IV = \tau\omega + I^2R + LI\frac{dI}{dt} + \text{power dissipated due to friction, sound, etc.}$$

Ignoring the last term, we have our simple motor model, written in terms of power:

$$IV = \tau\omega + I^2R + LI\frac{dI}{dt}.$$ (25.2)
From (25.2) we can derive all other relationships of interest. For example, dividing both sides of (25.2) by \( I \) yields

\[
V = \frac{\tau}{I} \omega + IR + L \frac{dI}{dt}.
\]  
(25.3)

The ratio \( \tau/I \) is a constant, an expression of the Lorentz force law for the particular motor design. This constant, relating current to torque, is called the **torque constant** \( k_t \). The torque constant is one of the most important properties of the motor:

\[
k_t = \frac{\tau}{I} \quad \text{or} \quad \tau = k_t I.
\]  
(25.4)

The SI units of \( k_t \) are Nm/A. (In this chapter, we only use SI units, but you should be aware that many different units are used by different manufacturers, as on the speed-torque curve and data sheet in Figure 25.16 in the Exercises.) Equation (25.3) also shows that the SI units for \( k_t \) can be written equivalently as Vs/rad, or simply Vs.

When using these units, we sometimes call the motor constant the **electrical constant** \( k_e \). The inverse is sometimes called the **speed constant**. You should recognize that these terms all refer to the same property of the motor. For consistency, we usually refer to the torque constant \( k_t \).

We now express the motor model in terms of voltage as

\[
V = k_t \omega + IR + L \frac{dI}{dt}.
\]  
(25.5)

You should remember, or be able to quickly derive, the power equation (25.2), the torque constant (25.4), and the voltage equation (25.5).

The term \( k_t \omega \), with units of voltage, is called the **back-emf**, where emf is short for **electromotive force**. We could also call this “back-voltage.” Back-emf is the voltage generated by a spinning motor to “oppose” the input voltage generating the motion. For example, assume that the motor’s terminals are not connected to anything (open circuit). Then \( I = 0 \) and \( \frac{dI}{dt} = 0 \), so (25.5) reduces to

\[
V = k_t \omega.
\]

This equation indicates that back-driving the motor (e.g., spinning it by hand) will generate a voltage at the terminals. If we were to connect a capacitor across the motor terminals, then spinning the motor by hand would charge the capacitor, storing some of the mechanical energy we put in as electrical energy in the capacitor. In this situation, the motor acts as a generator, converting mechanical energy to electrical energy.

The existence of this back-emf term also means that if we put a constant voltage \( V \) across a free-spinning frictionless motor (i.e., the motor shaft is not connected to anything), after some
time it will reach a constant speed $V/k_t$. At this speed, by (25.5), the current $I$ drops to zero, meaning there is no more torque $\tau$ to accelerate the motor. This happens because as the motor accelerates, the back-emf increases, countering the applied voltage until no current flows (and hence there is no torque or acceleration).

### 25.3 The Speed-Torque Curve

Consider a motor spinning a boat’s propeller at constant velocity. The torque $\tau$ provided by the motor can be written

$$\tau = \tau_{\text{fric}} + \tau_{\text{pushing water}},$$

where $\tau_{\text{fric}}$ is the torque the motor has to generate to overcome friction and begin to spin, while $\tau_{\text{pushing water}}$ is the torque needed for the propeller to displace water when the motor is spinning at velocity $\omega$. In this section we assume $\tau_{\text{fric}} = 0$, so $\tau = \tau_{\text{pushing water}}$ in this example. In Section 25.4 we consider nonzero friction.

For a motor spinning at constant speed $\omega$ and providing constant torque $\tau$ (as in the propeller example above), the current $I$ is constant and therefore $dI/dt = 0$. Under these assumptions, (25.5) reduces to

$$V = k_t \omega + IR. \quad (25.6)$$

Using the definition of the torque constant, we get the equivalent form

$$\omega = \frac{1}{k_t} V - \frac{R}{k_t^2} \tau. \quad (25.7)$$

Equation (25.7) gives $\omega$ as a linear function of $\tau$ for a given constant $V$. This line, of slope $-R/k_t^2$, is called the speed-torque curve for the voltage $V$.

The speed-torque curve plots all the possible constant-current operating conditions with voltage $V$ across the motor. Assuming friction torque is zero, the line intercepts the $\tau = 0$ axis at

$$\omega_0 = V/k_t = \text{no load speed}.$$

The line intercepts the $\omega = 0$ axis at

$$\tau_{\text{stall}} = \frac{k_t V}{R} = \text{Stall torque}.$$

At the no-load condition, $\tau = I = 0$; the motor rotates at maximum speed with no current or torque. At the stall condition, the shaft is blocked from rotating, and the current ($I_{\text{stall}} = \tau_{\text{stall}}/k_t = V/R$) and output torque are maximized due to the lack of back-emf. Which
point along the speed-torque curve the motor actually operates at is determined by the load attached to the motor shaft.

An example speed-torque curve is shown in Figure 25.8. This motor has $\omega_0 = 500$ rad/s and $\tau_{\text{stall}} = 0.1067$ Nm for a nominal voltage of $V_{\text{nom}} = 12$ V. The operating region is any point below the speed-torque curve, corresponding to voltages less than or equal to 12 V. If the motor is operated at a different voltage $cV_{\text{nom}}$, the intercepts of the speed-torque curve are linearly scaled to $c\omega_0$ and $c\tau_{\text{stall}}$.

Imagine squeezing the shaft of a motor powered by a voltage $V$ and spinning at a constant velocity. Your hand is applying a small torque to the shaft. Since the motor is not accelerating and we are neglecting friction in the motor, the torque created by the motor’s coils must be equal and opposite the torque applied by your hand. Thus the motor operates at a specific point on the speed-torque curve. If you slowly squeeze the shaft harder, increasing the torque you apply to the rotor, the motor will slow down and increase the torque it applies, to balance your hand’s torque. Assuming the motor’s current changes slowly (i.e., $LdI/dt$ is negligible), then the operating point of the motor moves down and to the right on the speed-torque curve as you increase your squeeze force. When you squeeze hard enough that the motor can no longer move, the operating point is at the stall condition, the bottom-right point on the speed-torque curve.

The speed-torque curve corresponds to constant $V$, but not to constant input power $P_{\text{in}} = IV$. The current $I$ is linear with $\tau$, so the input electrical power increases linearly with $\tau$. The output mechanical power is $P_{\text{out}} = \tau\omega$, and the efficiency in converting electrical to mechanical power is $\eta = P_{\text{out}}/P_{\text{in}} = \tau\omega/IV$. We return to efficiency in Section 25.4.
To find the point on the speed-torque curve that maximizes the mechanical output power, we can write points on the curve as \((\tau, \omega) = (c\tau_{\text{stall}}, (1 - c)\omega_0)\) for \(0 \leq c \leq 1\), so the output power is expressed as

\[
P_{\text{out}} = \tau \omega = (c - c^2)\tau_{\text{stall}}\omega_0,
\]

and the value of \(c\) that maximizes the power output is found by solving

\[
\frac{d}{dc} \left((c - c^2)\tau_{\text{stall}}\omega_0\right) = (1 - 2c)\tau_{\text{stall}}\omega_0 = 0 \quad \rightarrow \quad c = \frac{1}{2}.
\]

Thus the mechanical output power is maximized at \(\tau = \tau_{\text{stall}}/2\) and \(\omega = \omega_0/2\). This maximum output power is

\[
P_{\text{max}} = \left(\frac{1}{2}\tau_{\text{stall}}\right)\left(\frac{1}{2}\omega_0\right) = \frac{1}{4}\tau_{\text{stall}}\omega_0.
\]

See Figure 25.9.

Motor current is proportional to motor torque, so operating at high torques means large coil heating power loss \(I^2R\), sometimes called ohmic heating. For that reason, motor manufacturers specify a maximum continuous current \(I_{\text{cont}}\), the largest continuous current such that the coils’ steady-state temperature remains below a critical point.\(^1\) The maximum continuous current has a corresponding maximum continuous torque \(\tau_{\text{cont}}\). Points to the left of this torque and under the speed-torque curve are called the continuous operating region. The motor can be

---

\(\text{Figure 25.9}\)

The quadratic mechanical power plot \(P = \tau \omega\) plotted alongside the speed-torque curve. The area of the speed-torque rectangle below and to the left of the operating point is the mechanical power.

\(^1\) The maximum continuous current depends on thermal properties governing how fast coil heat can be transferred to the environment. This depends on the environment temperature, typically considered to be room temperature. The maximum continuous current can be increased by cooling the motor.
operated intermittently outside of the continuous operating region, in the *intermittent operating region*, provided the motor is allowed to cool sufficiently between uses in this region. Motors are commonly rated with a nominal voltage that places the maximum mechanical power operating point (at $\tau_{\text{stall}}/2$) outside the continuous operating region.

Given thermal characteristics of the motor of Figure 25.8, the speed-torque curve can be refined to Figure 25.10, showing the continuous and intermittent operating regions of the motor. The point on the speed-torque curve at $\tau_{\text{cont}}$ is the *rated* or *nominal operating point*, and the mechanical power output at this point is called the motor’s *power rating*. For the motor of Figure 25.10, $\tau_{\text{cont}} = 26.67$ mNm, which occurs at $\omega = 375$ rad/s, for a power rating of

$$0.02667 \text{ Nm} \times 375 \text{ rad/s} = 10.0 \text{ W}.$$  

Figure 25.10 also shows the constant output power hyperbola $\tau \omega = 10$ W passing through the nominal operating point.

The speed-torque curve for a motor is drawn based on a nominal voltage. This is a “safe” voltage that the manufacturer recommends. It is possible to overvolt the motor, however, provided it is not continuously operated beyond the maximum continuous current. A motor also may have a specified *maximum permissible speed* $\omega_{\text{max}}$, which creates a horizontal line constraint on the permissible operating range. This speed is determined by allowable brush wear, or possibly properties of the shaft bearings, and it is typically larger than the no-load speed $\omega_0$. The shaft and bearings may also have a maximum torque rating $\tau_{\text{max}} > \tau_{\text{stall}}$. These

![Figure 25.10](image)

The continuous operating region (under the speed-torque curve and left of $\tau_{\text{cont}}$) and the intermittent operating region (the rest of the area under the speed-torque curve). The 10 W mechanical power hyperbola is indicated, including the nominal operating point at $\tau_{\text{cont}}$. 

For more information

visit www.nu32.org
limits allow the definition of overvolted continuous and intermittent operating regions, as shown in Figure 25.11.

25.4 Friction and Motor Efficiency

Until now we have been assuming that the full torque $\tau = k_t I$ generated by the windings is available at the output shaft. In practice, some torque is lost due to friction at the brushes and the shaft bearings. Let us use a simple model of friction: assume a torque $\tau \geq \tau_{\text{fric}} > 0$ must be generated to overcome friction and initiate motion, and any torque beyond $\tau_{\text{fric}}$ is available at the output shaft regardless of the motor speed (e.g., no friction that depends on speed magnitude). When the motor is spinning, the torque available at the output shaft is

$$\tau_{\text{out}} = \tau - \tau_{\text{fric}}.$$ 

Nonzero friction results in a nonzero no-load current $I_0 = \tau_{\text{fric}}/k_t$ and a no-load speed $\omega_0$ less than $V/k_t$. The speed-torque curve of Figure 25.11 is modified to show a small friction torque in Figure 25.12. The torque actually delivered to the load is reduced by $\tau_{\text{fric}}$.

Taking friction into account, the motor’s efficiency in converting electrical to mechanical power is

$$\eta = \frac{\tau_{\text{out}} \omega}{IV}.$$ \hspace{1cm} (25.8)
Figure 25.12
The speed-torque curve of Figure 25.11 modified to show a nonzero friction torque $\tau_{\text{fric}}$ and the resulting reduced no-load speed $\omega_0$.

Figure 25.13
The speed-torque curve for a motor and two efficiency plots, one for high friction torque (case 1) and one for low friction torque (case 2). For each case, efficiency is zero for all $\tau$ below the level needed to overcome friction. The low friction version of the motor (case 2) achieves a higher maximum efficiency, at a higher speed and lower torque, than the high friction version (case 1).

The efficiency depends on the operating point on the speed-torque curve, and it is zero when either $\tau_{\text{out}}$ or $\omega$ is zero, as there is no mechanical power output. Maximum efficiency generally occurs at high speed and low torque, approaching the limit of 100% efficiency at $\tau = \tau_{\text{out}} = 0$ and $\omega = \omega_0$ as $\tau_{\text{fric}}$ approaches zero. As an example, Figure 25.13 plots efficiency vs. torque for the same motor with two different values of $\tau_{\text{fric}}$. Lower friction results in a higher maximum efficiency $\eta_{\text{max}}$, occurring at a higher speed and lower torque.
To derive the maximally efficient operating point and the maximum efficiency $\eta_{\text{max}}$ for a given motor, we can express the motor current as

$$I = I_0 + I_a,$$

where $I_0$ is the no-load current necessary to overcome friction and $I_a$ is the added current to create torque to drive the load. Recognizing that $\tau_{\text{out}} = k_I I_a$, $V = I_{\text{stall}} R$, and $\omega = R(I_{\text{stall}} - I_a - I_0)/k_I$ by the linearity of the speed-torque curve, we can rewrite the efficiency (25.8) as

$$\eta = \frac{I_a(I_{\text{stall}} - I_0 - I_a)}{(I_0 + I_a)I_{\text{stall}}}.$$  \hspace{1cm} (25.9)

To find the operating point $I_a^*$ maximizing $\eta$, we solve $d\eta/dI_a = 0$ for $I_a^*$, and recognizing that $I_0$ and $I_{\text{stall}}$ are nonnegative, the solution is

$$I_a^* = \sqrt{I_{\text{stall}}I_0} - I_0.$$

In other words, as the no-load current $I_0$ goes to zero, the maximally efficient current (and therefore $\tau$) goes to zero.

Plugging $I_a^*$ into (25.9), we find

$$\eta_{\text{max}} = \left(1 - \sqrt{\frac{I_0}{I_{\text{stall}}}}\right)^2.$$

This answer has the form we would expect: maximum efficiency approaches 100% as the friction torque approaches zero, and maximum efficiency approaches 0% as the friction torque approaches the stall torque.

Choosing an operating point that maximizes motor efficiency can be important when trying to maximize battery life in mobile applications. For the majority of analysis and motor selection problems, however, ignoring friction is a good first approximation.

### 25.5 Motor Windings and the Motor Constant

It is possible to build two different versions of the same motor by simply changing the windings while keeping everything else the same. For example, imagine a coil of resistance $R$ with $N$ loops of wire of cross-sectional area $A$. The coil carries a current $I$ and therefore has a voltage drop $IR$. Now we replace that coil with a new coil with $N/c$ loops of wire with cross-sectional area $cA$. This preserves the volume occupied by the coil, fitting in the same

For more information
visit www.nu32.org
form factor with similar thermal properties. Without loss of generality, let us assume that the new coil has fewer loops and uses thicker wire \((c > 1)\).

The resistance of the new coil is reduced to \(R/c^2\) (a factor of \(c\) due to the shorter coil and another factor of \(c\) due to the thicker wire). To keep the torque of the motor the same, the new coil would have to carry a larger current \(cI\) to make up for the fewer loops, so that the current times the pathlength through the magnetic field is unchanged. The voltage drop across the new coil is \((cI)(R/c^2) = IR/c\).

Replacing the coils allows us to create two versions of the motor: a many-loop, thin wire version that operates at low current and high voltage, and a fewer-loop, thick wire version that operates at high current and low voltage. Since the two motors create the same torque with different currents, they have different torque constants. Each motor has the same motor constant \(k_m\), however, where

\[
k_m = \frac{\tau}{\sqrt{I^2R}} = \frac{k_t}{\sqrt{R}}
\]

with units of Nm/√W. The motor constant defines the torque generated per square root of the power dissipated by coil resistance. In the example above, the new coil dissipates \((cI)^2(R/c^2) = I^2R\) power as heat, just as the original coil does, while generating the same torque.

Figure 25.16 shows the data sheet for a motor that comes in several different versions, each identical in every way except for the winding. Each version of the motor has a similar stall torque and motor constant but different nominal voltage, resistance, and torque constant.

### 25.6 Other Motor Characteristics

Electrical time constant

When the motor is subject to a step in the voltage across it, the electrical time constant \(T_e\) measures the time it takes for the unloaded current to reach 63% of its final value. The motor’s voltage equation is

\[
V = k_t \omega + IR + L \frac{dI}{dt}.
\]

Ignoring back-emf (because the motor speed does not change significantly over one electrical time constant), assuming an initial current through the motor of \(I_0\), and an instantaneous drop in the motor voltage to 0, we get the differential equation
\[ 0 = I_0 R + L \frac{dI}{dt} \]

or

\[ \frac{dI}{dt} = -\frac{R}{L} I_0 \]

with solution

\[ I(t) = I_0 e^{-tR/L} = I_0 e^{-t/T_e}. \]

The time constant of this first-order decay of current is the motor’s electrical time constant, \( T_e = L/R \).

Mechanical time constant

When the motor is subject to a step voltage across it, the mechanical time constant \( T_m \) measures the time it takes for the unloaded motor speed to reach 63\% of its final value.

Beginning from the voltage equation

\[ V = k_i \omega + IR + L \frac{dI}{dt}, \]

ignoring the inductive term, and assuming an initial speed \( \omega_0 \) at the moment the voltage drops to zero, we get the differential equation

\[ 0 = IR + k_i \omega_0 = \frac{R}{k_i} \tau + k_i \omega_0 = \frac{JR}{k_i} \frac{d\omega}{dt} + k_i \omega_0, \]

where we used \( \tau = Jd\omega/dt \), where \( J \) is the inertia of the motor. We can rewrite this equation as

\[ \frac{d\omega}{dt} = -\frac{k_i^2}{JR} \omega_0 \]

with solution

\[ \omega(t) = \omega_0 e^{-t/T_m}, \]

with a time constant of \( T_m = JR/k_i^2 \). If the motor is attached to a load that increases the inertia \( J \), the mechanical time constant increases.
Short-circuit damping

When the terminals of the motor are shorted together, the voltage equation (ignoring inductance) becomes

\[ 0 = k_t \omega + IR = k_t \omega + \frac{\tau}{k_t} R \]

or

\[ \tau = -B \omega = -\frac{k_t^2}{R} \omega, \]

where \( B = \frac{k_t^2}{R} \) is the short-circuit damping. A spinning motor is slowed more quickly by shorting its terminals together, compared to leaving the terminals open circuit, due to this damping.

\section*{25.7 Motor Data Sheet}

Motor manufacturers summarize motor properties described above in a speed-torque curve and in a data sheet similar to the one in Figure 25.14. When you buy a motor second-hand or surplus, you may need to measure these properties yourself. We will use all SI units, which is not the case on most motor data sheets.

Many of these properties have been introduced already. Below we describe some methods for estimating them.

\textit{Experimentally Characterizing a Brushed DC Motor}

Given a mystery motor with an encoder, you can use a function generator, oscilloscope, multimeter and perhaps some resistors and capacitors to estimate most of the important properties of the motor. Below are some suggested methods; you may be able to devise others.

**Terminal resistance \( R \)**

You can measure \( R \) with a multimeter. The resistance may change as you rotate the shaft by hand, as the brushes move to new positions on the commutator. You should record the minimum resistance you can reliably find. A better choice, however, may be to measure the current when the motor is stalled.

**Torque constant \( k_t \)**

You can measure this by spinning the shaft of the motor, measuring the back-emf at the motor terminals, and measuring the rotation rate \( \omega \) using the encoder. Or, if friction losses are
<table>
<thead>
<tr>
<th>Motor Characteristic</th>
<th>Symbol</th>
<th>Value</th>
<th>Units</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal resistance</td>
<td>$R$</td>
<td></td>
<td>$\Omega$</td>
<td>Resistance of the motor windings. May change as brushes slide over commutator segments. Increases with heat.</td>
</tr>
<tr>
<td>Torque constant</td>
<td>$k_t$</td>
<td>Nm/A</td>
<td></td>
<td>The constant ratio of torque produced to current through the motor.</td>
</tr>
<tr>
<td>Electrical constant</td>
<td>$k_e$</td>
<td>Vs/rad</td>
<td></td>
<td>Same numerical value as the torque constant (in SI units). Also called voltage or back-emf constant.</td>
</tr>
<tr>
<td>Speed constant</td>
<td>$k_s$</td>
<td>rad/(Vs)</td>
<td></td>
<td>Inverse of electrical constant.</td>
</tr>
<tr>
<td>Motor constant</td>
<td>$k_m$</td>
<td>Nm/$\sqrt{W}$</td>
<td></td>
<td>Torque produced per square root of power dissipated by the coils.</td>
</tr>
<tr>
<td>Max continuous current</td>
<td>$I_{cont}$</td>
<td>A</td>
<td></td>
<td>Max continuous current without overheating.</td>
</tr>
<tr>
<td>Max continuous torque</td>
<td>$\tau_{cont}$</td>
<td>Nm</td>
<td></td>
<td>Max continuous torque without overheating.</td>
</tr>
<tr>
<td>Short-circuit damping</td>
<td>$B$</td>
<td>Nms/rad</td>
<td></td>
<td>Not included in most data sheets, but useful for motor braking (and haptics).</td>
</tr>
<tr>
<td>Terminal inductance</td>
<td>$L$</td>
<td>H</td>
<td></td>
<td>Inductance due to the coils.</td>
</tr>
<tr>
<td>Electrical time constant</td>
<td>$T_e$</td>
<td>s</td>
<td></td>
<td>The time for the motor current to reach 63% of its final value. Equal to $L/R$.</td>
</tr>
<tr>
<td>Rotor inertia</td>
<td>$J$</td>
<td>kgm$^2$</td>
<td></td>
<td>Often given in units gcm$^2$.</td>
</tr>
<tr>
<td>Mechanical time constant</td>
<td>$T_m$</td>
<td>s</td>
<td></td>
<td>The time for the motor to go from rest to 63% of its final speed under constant voltage and no load. Equal to $JR/k_t^2$.</td>
</tr>
<tr>
<td>Friction</td>
<td></td>
<td></td>
<td></td>
<td>Not included in most data sheets. See explanation.</td>
</tr>
</tbody>
</table>

| Values at Nominal Voltage                        |        |       |       |                                                                          |
| Nominal voltage                                  | $V_{nom}$ | V |       | Should be chosen so the no-load speed is safe for brushes, commutator, and bearings. |
| Power rating                                     | $P$    | W    |       | Output power at the nominal operating point (max continuous torque).       |
| No-load speed                                     | $\omega_0$ | rad/s |       | Speed when no load and powered by $V_{nom}$. Usually given in rpm (revs/min, sometimes m$^{-1}$). |
| No-load current                                   | $I_0$  | A    |       | The current required to spin the motor at the no-load condition. Nonzero because of friction torque. |
| Stall current                                     | $I$    | A    |       | Same as starting current, $V_{nom}/R$.                                   |
| Stall torque                                      | $\tau_{stall}$ | Nm |       | The torque achieved at the nominal voltage when the motor is stalled.    |
| Max mechanical power                              | $P_{max}$ | W |       | The max mechanical power output at the nominal voltage (including short-term operation). |
| Max efficiency                                    | $\eta$ | %    |       | The maximum efficiency achievable in converting electrical to mechanical power. |

**Figure 25.14**
A sample motor data sheet, with values to be filled in.

Negligible, a good approximation is $V_{nom}/\omega_0$. This eliminates the need to spin the motor externally.

**Electrical constant $k_e$**

Identical to the torque constant in SI units. The torque constant $k_t$ is often expressed in units of Nm/A or mNm/A or English units like oz-in/A, and often $k_e$ is given in V/rpm, but $k_t$ and $k_e$ have identical numerical values when expressed in Nm/A and Vs/rad, respectively.
Speed constant $k_s$
Just the inverse of the electrical constant.

Motor constant $k_m$
The motor constant is calculated as $k_m = k_t / \sqrt{R}$.

Max continuous current $I_{\text{cont}}$
This is determined by thermal considerations, which are not easy to measure. It is typically less than half the stall current.

Max continuous torque $\tau_{\text{cont}}$
This is determined by thermal considerations, which are not easy to measure. It is typically less than half the stall torque.

Short-circuit damping $B$
This is most easily calculated from estimates of $R$ and $k_t$: $B = k_t^2 / R$.

Terminal inductance $L$
There are several ways to measure inductance. One approach is to add a capacitor in parallel with the motor and measure the oscillation frequency of the resulting RLC circuit. For example, you could build the circuit shown in Figure 25.15, where a good choice for $C$ may be 0.01 or 0.1 $\mu$F. The motor acts as a resistor and inductor in series; back-emf will not be an issue, because the motor will be powered by tiny currents at high frequency and therefore will not move.

![Figure 25.15](https://www.nu32.org)

Using a capacitor to create an RLC circuit to measure motor inductance.

For more information visit www.nu32.org
Use a function generator to put a 1 kHz square wave between 0 and 5 V at the point indicated. The 1 kΩ resistor limits the current from the function generator. Measure the voltage with an oscilloscope where indicated. You should be able to see a decaying oscillatory response to the square wave input when you choose the right scales on your scope. Measure the frequency of the oscillatory response. Knowing \( C \) and that the natural frequency of an RLC circuit is \( \omega_n = 1/\sqrt{LC} \) in rad/s, estimate \( L \).

Let us think about why we see this response. Say the input to the circuit has been at 0 V for a long time. Then your scope will also read 0 V. Now the input steps up to 5 V. After some time, in steady state, the capacitor will be an open circuit and the inductor will be a closed circuit (wire), so the voltage on the scope will settle to \( 5 \text{ V} \times (R/(1000 + R)) \)—the two resistors in the circuit set the final voltage. Right after the voltage step, however, all current goes to charge the capacitor (as the zero current through the inductor cannot change discontinuously). If the inductor continued to enforce zero current, the capacitor would charge to 5 V. As the voltage across the capacitor grows, however, so does voltage across the inductor, and therefore so does the rate of change of current that must flow through the inductor (by the relation \( V_L + V_R = V_C \) and the constitutive law \( V_L = L \frac{dI}{dt} \)). Eventually the integral of this rate of change dictates that all current is redirected to the inductor, and in fact the capacitor will have to provide current to the inductor, discharging itself. As the voltage across the capacitor drops, though, the voltage across the inductor will eventually become negative, and therefore the rate of change of current through the inductor will become negative. And so on, to create the oscillation. If \( R \) were large, i.e., if the circuit were heavily damped, the oscillation would die quickly, but you should be able to see it.

Note that you are seeing a damped oscillation, so you are actually measuring a damped natural frequency. But the damping is low if you are seeing at least a couple of cycles of oscillation, so the damped natural frequency is nearly indistinguishable from the undamped natural frequency.

Electrical time constant \( T_e \)

The electrical time constant can be calculated from \( L \) and \( R \) as \( T_e = L/R \).

Rotor inertia \( J \)

The rotor inertia can be estimated from measurements of the mechanical time constant \( T_m \), the torque constant \( k_t \), and the resistance \( R \). Alternatively, a ballpark estimate can be made based on the mass of the motor, a guess at the portion of the mass that belongs to the spinning rotor, a guess at the radius of the rotor, and a formula for the inertia of a uniform density cylinder. Or, more simply, consult a data sheet for a motor of similar size and mass.
Mechanical time constant $T_m$

The time constant can be measured by applying a constant voltage to the motor, measuring the velocity, and determining the time it takes to reach 63% of final speed. Alternatively, you could make a reasonable estimate of the rotor inertia $J$ and calculate $T_m = JR/k_t^2$.

Friction

Friction torque arises from the brushes sliding on the commutator and the motor shaft spinning in its bearings, and it may depend on external loads. A typical model of friction includes both Coulomb friction and viscous friction, written

$$\tau_{\text{fric}} = b_0 \text{sgn}(\omega) + b_1 \omega,$$

where $b_0$ is the Coulomb friction torque ($\text{sgn}(\omega)$ just returns the sign of $\omega$) and $b_1$ is a viscous friction coefficient. At no load, $\tau_{\text{fric}} = k_t I_0$. An estimate of each of $b_0$ and $b_1$ can be made by running the motor at two different voltages with no load.

Nominal voltage $V_{\text{nom}}$

This is the specification you are most likely to know for an otherwise unknown motor. It is sometimes printed right on the motor itself. This voltage is just a recommendation; the real issue is to avoid overheating the motor or spinning it at speeds beyond the recommended value for the brushes or bearings. Nominal voltage cannot be measured, but a typical no-load speed for a brushed DC motor is between 3000 and 10,000 rpm, so the nominal voltage will often give a no-load speed in this range.

Power rating $P$

The power rating is the mechanical power output at the max continuous torque.

No-load speed $\omega_0$

You can determine $\omega_0$ by measuring the unloaded motor speed when powered with the nominal voltage. The amount that this is less than $V_{\text{nom}}/k_t$ can be attributed to friction torque.

No-load current $I_0$

You can determine $I_0$ by using a multimeter in current measurement mode.
Stall current $I_{\text{stall}}$

Stall current is sometimes called starting current. You can estimate this using your estimate of $R$. Since $R$ may be difficult to measure with a multimeter, you can instead stall the motor shaft and use your multimeter in current sensing mode, provided the multimeter can handle the current.

Stall torque $\tau_{\text{stall}}$

This can be obtained from $k_t$ and $I_{\text{stall}}$.

Max mechanical power $P_{\text{max}}$

The max mechanical power occurs at $\frac{1}{2} \tau_{\text{stall}}$ and $\frac{1}{2} \omega_0$. For most motor data sheets, the max mechanical power occurs outside the continuous operation region.

Max efficiency $\eta_{\text{max}}$

Efficiency is defined as the power out divided by the power in, $\tau_{\text{out}} \omega / (IV)$. The wasted power is due to coil heating and friction losses. Maximum efficiency can be estimated using the no-load current $I_0$ and the stall current $I_{\text{stall}}$, as discussed in Section 25.4.

25.8 Chapter Summary

- The Lorentz force law says that a current-carrying conductor in a constant magnetic field feels a net force according to
  \[
  \mathbf{F} = \ell \mathbf{I} \times \mathbf{B},
  \]
  where $\ell$ is the length of the conductor in the field, $\mathbf{I}$ is the current vector, and $\mathbf{B}$ is the (constant) magnetic field vector.

- A brushed DC motor consists of multiple current-carrying coils attached to a rotor, and magnets on the stator to create a magnetic field. Current is transmitted to the coils by two brushes connected to the stator sliding over a commutator ring attached to the rotor. Each coil attaches to two different commutator segments.

- The voltage across a motor’s terminals can be expressed as
  \[
  V = k_t \omega + IR + L \frac{dI}{dt},
  \]
  where $k_t$ is the torque constant and $k_t \omega$ is the back-emf.

- The speed-torque curve is obtained by plotting the steady-state speed as a function of torque for a given motor voltage $V$, for more information visit www.nu32.org
\[
\omega = \frac{1}{k_I} V - \frac{R}{k_I^2} \tau.
\]

The maximum speed (at \(\tau = 0\)) is called the no-load speed \(\omega_0\) and the maximum torque (at \(\omega = 0\)) is called the stall torque \(\tau_{stall}\).

- The continuous operating region of a motor is defined by the maximum current \(I\) the motor coils can conduct continuously without overheating due to \(I^2R\) power dissipation.
- The mechanical power \(\tau\omega\) delivered by a motor is maximized at half the stall torque and half the no-load speed, \(P_{\text{max}} = \frac{1}{4} \tau_{stall}\omega_0\).
- A motor’s electrical time constant \(T_e = L/R\) is the time needed for current to reach 63% of its final value in response to a step input in voltage.
- A motor’s mechanical time constant \(T_m = J/Rk_I^2\) is the time needed for the motor speed to reach 63% of its final value in response to a step change in voltage.

### 25.9 Exercises

1. Assume a DC motor with a five-segment commutator. Each segment covers 70° of the circumference of the commutator circle. The two brushes are positioned at opposite ends of the commutator circle, and each makes contact with 10° of the commutator circle.
   a. How many separate coils does this motor likely have? Explain.
   b. Choose one of the motor coils. As the rotor rotates 360°, what is the total angle over which that coil is energized? (For example, an answer of 360° means that the coil is energized at all angles; an answer of 180° means that the coil is energized at half of the motor positions.)

2. Figure 25.16 gives the data sheet for the 10 W Maxon RE 25 motor. The columns correspond to different windings.
   a. Draw the speed-torque curve for the 12 V version of the motor, indicating the no-load speed (in rad/s), the stall torque, the nominal operating point, and the rated power of the motor.
   b. Explain why the torque constant is different for the different versions of the motor.
   c. Using other entries in the table, calculate the maximum efficiency \(\eta_{\text{max}}\) of the 12 V motor and compare to the value listed.
   d. Calculate the electrical time constant \(T_e\) of the 12 V motor. What is the ratio to the mechanical time constant \(T_m\)?
   e. Calculate the short-circuit damping \(B\) for the 12 V motor.
   f. Calculate the motor constant \(k_m\) for the 12 V motor.
   g. How many commutator segments do these motors have?
   h. Which versions of these motors are likely to be in stock?
The data sheet for the Maxon RE 25 motor. The columns correspond to different windings for different nominal voltages. (Image courtesy of Maxon Precision Motors. Motor data is subject to change at any time; consult maxonmotorusa.com for the latest data sheets.)

1. (Optional) Motor manufacturers may specify slightly different continuous and intermittent operating regions than the ones described in this chapter. For example, the limit of the continuous operating region is not quite vertical in the speed-torque plot of Figure 25.16. Come up with a possible explanation, perhaps using online resources.

3. There are 21 entries on the motor data sheet from Section 25.7. Let us assume zero friction, so we ignore the last entry. To avoid thermal tests, you may also assume a maximum continuous power that the motor coils can dissipate as heat before overheating.
Of the 20 remaining entries, under the assumption of zero friction, how many independent entries are there? That is, what is the minimum number $N$ of entries you need to be able to fill in the rest of the entries? Give a set of $N$ independent entries from which you can derive the other $20 - N$ dependent entries. For each of the $20 - N$ dependent entries, give the equation in terms of the $N$ independent entries. For example, $V_{\text{nom}}$ and $R$ will be two of the $N$ independent entries, from which we can calculate the dependent entry $I_{\text{stall}} = V_{\text{nom}}/R$.

4. This exercise is an experimental characterization of a motor. For this exercise, you need a low-power motor (preferably without a gearhead to avoid high friction) with an encoder. You also need a multimeter, oscilloscope, and a power source for the encoder and motor. Make sure the power source for the motor can provide enough current when the motor is stalled. A low-voltage battery pack is a good choice.

a. Spin the motor shaft by hand. Get a feel for the rotor inertia and friction. Try to spin the shaft fast enough that it continues spinning briefly after you let go of it.

b. Now short the motor terminals by electrically connecting them. Spin again by hand, and try to spin the shaft fast enough that it continues spinning briefly after you let go of it. Do you notice the short-circuit damping?

c. Try measuring your motor’s resistance using your multimeter. It may vary with the angle of the shaft, and it may not be easy to get a steady reading. What is the minimum value you can get reliably? To double-check your answer, you can power your motor and use your multimeter to measure the current as you stall the motor’s shaft by hand.

d. Attach one of your motor’s terminals to scope ground and the other to a scope input. Spin the motor shaft by hand and observe the motor’s back-emf.

e. Power the motor’s encoder, attach the A and B encoder channels to your oscilloscope, and make sure the encoder ground and scope ground are connected together. Do not power the motor. (The motor inputs should be disconnected from anything.) Spin the motor shaft by hand and observe the encoder pulses, including their relative phase.

f. Now power your motor with a low-voltage battery pack. Given the number of lines per revolution of the encoder, and the rate of the encoder pulses you observe on your scope, calculate the motor’s no-load speed for the voltage you are using.

g. Work with a partner. Couple your two motor shafts together by tape or flexible tubing. (This may only work if your motor has no gearhead.) Now plug one terminal of one of the motors (we shall call it the passive motor) into one channel of a scope, and plug the other terminal of the passive motor into GND of the same scope. Now power the other motor (the driving motor) with a battery pack so that both motors spin. Measure the speed of the passive motor by looking at its encoder count rate on your scope. Also measure its back-emf. With this information, calculate the passive motor’s torque constant $k_t$.

5. Using techniques discussed in this chapter, or techniques you come up with on your own, create a data sheet with all 21 entries for your nominal voltage. Indicate how you
calculated the entry. (Did you do an experiment for it? Did you calculate it from other entries? Or did you estimate by more than one method to cross-check your answer?) For the friction entry, you can assume Coulomb friction only—the friction torque opposes the rotation direction \((b_0 \neq 0)\), but is independent of the speed of rotation \((b_1 = 0)\). For your measurement of inductance, turn in an image of the scope trace you used to estimate \(\omega_n\) and \(L\), and indicate the value of \(C\) that you used.

If there are any entries you are unable to estimate experimentally, approximate, or calculate from other values, simply say so and leave that entry blank.

6. Based on your data sheet from above, draw the speed-torque curves described below, and answer the associated questions. Do not do any experiments for this exercise; just extrapolate your previous results.

   a. Draw the speed-torque curve for your motor. Indicate the stall torque and no-load speed. Assume a maximum power the motor coils can dissipate continuously before overheating and indicate the continuous operating regime. Given this, what is the power rating \(P\) for this motor? What is the max mechanical power \(P_{\text{max}}\)?

   b. Draw the speed-torque curve for your motor assuming a nominal voltage four times larger than in Exercise 6a. Indicate the stall torque and no-load speed. What is the max mechanical power \(P_{\text{max}}\)?

7. You are choosing a motor for the last joint of a new direct-drive robot arm design. (A direct-drive robot does not use gearheads on the motors, creating high speeds with low friction.) Since it is the last joint of the robot, and it has to be carried by all the other joints, you want it to be as light as possible. From the line of motors you are considering from your favorite motor manufacturer, you know that the mass increases with the motor’s power rating. Therefore you are looking for the lowest power motor that works for your specifications. Your specifications are that the motor should have a stall torque of at least 0.1 Nm, should be able to rotate at least 5 revolutions per second when providing 0.01 Nm, and the motor should be able to operate continuously while providing 0.02 Nm. Which motor do you choose from Table 25.1? Give a justification for your answer.

8. The speed-torque curve of Figure 25.8 is drawn for the positive speed and positive torque quadrant of the speed-torque plane. In this exercise, we will draw the motor’s operating region for all four quadrants. The power supply used to drive the motor is 24 V, and assume the H-bridge motor controller (discussed in Chapter 27) can use that power supply to create any average voltage across the motor between \(-24\) and 24 V. The motor’s resistance is 1 \(\Omega\) and the torque constant is 0.1 Nm/A. Assume the motor has zero friction.

   a. Draw the four-quadrant speed-torque operating region for the motor assuming the 24 V power supply (and the H-bridge driver) has no limit on current. Indicate the torque and speed values where the boundaries of the operating region intersect the \(\omega = 0\) and \(\tau = 0\) axes. Assume there are no other speed or torque constraints on the motor except for the one due to the 24 V limit of the power supply. (Hint: the operating region is unbounded in both speed and torque!)
Table 25.1: Motors to choose from

<table>
<thead>
<tr>
<th>Assigned power rating (W)</th>
<th>Nominal voltage (V)</th>
<th>No load speed (rpm)</th>
<th>No load current (mA)</th>
<th>Nominal speed (rpm)</th>
<th>Max continuous torque (mNm)</th>
<th>Max continuous current (mA)</th>
<th>Stall torque (mNm)</th>
<th>Stall current (mA)</th>
<th>Max efficiency (%)</th>
<th>Terminal resistance (Ohm)</th>
<th>Terminal inductance (mH)</th>
<th>Torque constant (mNm/A)</th>
<th>Speed constant (rpm/V)</th>
<th>Mechanical time constant (ms)</th>
<th>Rotor inertia (g cm²)</th>
<th>Max permissible speed (rpm)</th>
<th>Cost (USD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>15</td>
<td>13,400</td>
<td>36.8</td>
<td>10,400</td>
<td>2.31</td>
<td>259</td>
<td>10.5</td>
<td>1030</td>
<td>65</td>
<td>14.6</td>
<td>0.486</td>
<td>10.2</td>
<td>932</td>
<td>7.51</td>
<td>0.541</td>
<td>16,000</td>
<td>88</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>4980</td>
<td>21.8</td>
<td>3920</td>
<td>28.2</td>
<td>1010</td>
<td>131</td>
<td>4570</td>
<td>87</td>
<td>3.28</td>
<td>0.353</td>
<td>28.6</td>
<td>334</td>
<td>4.23</td>
<td>10.6</td>
<td>5500</td>
<td>228</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>9660</td>
<td>60.8</td>
<td>8430</td>
<td>20.5</td>
<td>1500</td>
<td>225</td>
<td>15,800</td>
<td>82</td>
<td>0.952</td>
<td>0.088</td>
<td>14.3</td>
<td>670</td>
<td>4.87</td>
<td>10.4</td>
<td>14,000</td>
<td>236</td>
</tr>
<tr>
<td>90</td>
<td>15</td>
<td>7180</td>
<td>247</td>
<td>6500</td>
<td>73.1</td>
<td>4000</td>
<td>929</td>
<td></td>
<td></td>
<td>0.314</td>
<td>0.085</td>
<td>19.4</td>
<td>491</td>
<td>5.65</td>
<td>68.1</td>
<td>12,000</td>
<td>239</td>
</tr>
</tbody>
</table>

Note that sometimes the “Assigned power rating” is different from the mechanical power output at the nominal operating point, for manufacturer-specific reasons. The meanings of the other terms in the table are unambiguous.

b. Update the operating region with the constraint that the power supply can provide a maximum current of 30 A. What is the maximum torque that can be generated using this power supply, and what are the maximum and minimum motor speeds possible at this maximum torque? What is the largest back-emf voltage that can be achieved?

c. Update the operating region with the constraint that the maximum recommended speed for the motor brushes and shaft bearings is 250 rad/s.

d. Update the operating region with the constraint that the maximum recommended torque at the motor shaft is 5 Nm.

e. Update the operating region to show the continuous operating region, assuming the maximum continuous current is 10 A.

f. We typically think of a motor as consuming electrical power ($IV > 0$, or “motoring”) and converting it to mechanical power, but it can also convert mechanical power to electrical power ($IV < 0$, or “regenerating”). This occurs in electric car braking systems, for example. Update the operating region to show the portion where the motor is consuming electrical power and the portion where the motor is generating electrical power.

Further Reading


For more information visit www.nu32.org
A Crash Course in C

This appendix provides an introduction to C for readers with no C experience but some experience in another programming language. It is not intended as a complete reference; plenty of great C resources already exist and provide a more complete picture. This appendix applies to C in general, not just C on the Microchip PIC32. We recommend that you start by programming your computer so you can experiment with C without needing extra equipment or complication.

A.1 Quick Start in C

To start with C, you need a computer, a text editor, and a C compiler. You use the text editor to write your C program, a plain text file with a name ending with the extension .c (e.g., myprog.c). The C compiler converts this program into machine code that your computer can execute. There are many free C compilers available; we recommend the gcc C compiler, which is part of the GNU Compiler Collection (GCC, found at http://gcc.gnu.org). GCC is available for Windows, Mac OS, and Linux. For Windows, you can download the GCC collection in MinGW. If the installation asks you about what tools to install, make sure to include the make tools. For Mac OS, you can download the full Xcode environment from the Apple Developers website. This installation is multiple gigabytes; however, you can opt to install only the “Command Line Tools for Xcode,” which is smaller and more than sufficient for getting started with C (and for this appendix).

Many C installations come with an Integrated Development Environment (IDE) complete with text editor, menus, and graphical tools to assist you with your programming projects. Every IDE is different, and what we cover in this appendix does not require a sophisticated IDE. We therefore use only command line tools, meaning that we initiate compilation and run the program by typing at the command line. In Mac OS, the command line can be accessed from the Terminal program. In Windows, you can access the command line by searching for cmd or command prompt. Linux users should run a shell such as bash.

1 You are also welcome to use Visual C from Microsoft. The command line compile command will look a bit different than what you see in this appendix.
To use the command line, you must learn some command line instructions. The Mac operating system is built on top of Unix, which is similar to Linux, so Mac/Unix/Linux use the same syntax. Windows uses slightly different commands. See the table of a few useful commands below. You can find more information on how to use these commands as well as others by searching for command line commands in Unix, Linux, or Windows.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mac/Unix/Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show current directory</td>
<td>pwd</td>
<td>cd</td>
</tr>
<tr>
<td>List directory contents</td>
<td>ls</td>
<td>dir</td>
</tr>
<tr>
<td>Make subdirectory newdir</td>
<td>mkdir newdir</td>
<td>mkdir newdir</td>
</tr>
<tr>
<td>Change to subdirectory newdir</td>
<td>cd newdir</td>
<td>cd newdir</td>
</tr>
<tr>
<td>Move “up” to parent directory</td>
<td>cd ..</td>
<td>cd ..</td>
</tr>
<tr>
<td>Copy file to filenew</td>
<td>cp file filenew</td>
<td>copy file filenew</td>
</tr>
<tr>
<td>Delete file file</td>
<td>rm file</td>
<td>del file</td>
</tr>
<tr>
<td>Delete directory dir</td>
<td>rmdir dir</td>
<td>rmdir dir</td>
</tr>
<tr>
<td>Help on using command cmd</td>
<td>man cmd</td>
<td>cmd /?</td>
</tr>
</tbody>
</table>

Following the long-established programming tradition, your first C program will simply print “Hello world!” to the screen. Use a text editor to create the file `HelloWorld.c`:

```c
#include <stdio.h>
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

Possible text editors include Notepad++ for Windows, TextWrangler for Mac OS, and Gedit for Linux. You can also try vim or emacs, though they are not easy to get started with! Whichever editor you use, you should save your file as plain text, not rich text or any other formatted text.

To compile your program, navigate from the command line to the directory where the program sits. Then, assuming your command prompt appears as `>`, type the following at the prompt:

```bash
> gcc HelloWorld.c -o HelloWorld
```

This command should create the executable file `HelloWorld` in the same directory. (The argument after the `-o` output flag is the name of the executable file to be created from `HelloWorld.c`.) Now, to execute the program, type

**Windows:** `> HelloWorld`

**Linux/MacOS:** `> ./HelloWorld`

For Linux/MacOS users, the “.” is shorthand for “current directory,” and the `./` tells your computer to look in the current directory for `HelloWorld`. Windows implicitly searches the current directory for executables, so you need not explicitly specify it.
If you have succeeded in getting this far, your C installation works and you can proceed. If not, you may need to get help from friends or the internet.

### A.2 Overview

If you are familiar with a high-level language like MATLAB or Python, you may know about loops, functions, and other programming constructs. You will see that although C’s syntax is different, the same concepts translate to C. Rather than starting with basic loops, if statements, and functions, we begin by focusing on important concepts that you must master in C but which you probably have not dealt with in a language such as MATLAB or Python.

- **Memory, addresses, and pointers.** Variables are stored at particular *addresses* in memory as bits (0’s and 1’s). In C, unlike in MATLAB or Python, it is often useful to access a variable’s memory address. Special variables called *pointers* contain the address of another variable and can be used to access the contents of that address. Although powerful, pointers can also be dangerous; misusing them can cause all sorts of bugs, which is why many higher-level languages forgo them completely.

- **Data types.** In MATLAB, for example, you can simply type
  
  ```
  a = 1; 
  b = [1.2 3.1416]; 
  c = [1 2; 3 4]; 
  s = 'a string'; 
  ```

  MATLAB determines that a is a scalar, b is a vector with two elements, c is a $2 \times 2$ matrix, and s is a string; automatically tracks the variable’s type (e.g., a list of numbers for a vector or a list of characters for a string); and sets aside, or *allocates*, memory to store them. In C, on the other hand, you must first define the variable before you ever use it. To use a vector, for example, you must specify the number and *data type* of its elements—integers or decimal numbers (floating point). The variable definition tells the C compiler how much memory it needs to store the vector, the address of each element, and how to interpret the bits of each element (as integers or floating point numbers, for example).

- **Compiling.** MATLAB programs are typically *interpreted*: the commands are converted to machine code and executed while the program is running. C programs, on the other hand, are *compiled*, i.e., converted to machine-code in advance. This process consists of several steps whose purpose is to turn your C program into machine-code before it ever runs. The compiler can identify some errors and warn you about other questionable code. Compiled code typically runs faster than interpreted code, since the translation to machine code is done in advance.\(^2\)

Each of these concepts is described in Section A.3 without going into detail on C syntax. In Section A.4 we look at sample programs to introduce syntax, then offer more detailed explanations.

---

\(^2\) The distinction between compiled and interpreted programs is narrowing: many interpreted languages are actually just-in-time (JIT) compiled, that is program chunks are compiled in advance right before they are needed.
A.3 Important Concepts in C

We begin our discussion of C with this caveat:

C consists of an evolving set of standards for a programming language, and any specific C installation is an “implementation” of C. While C standards require certain behavior from all implementations, some details are implementation-dependent. For example, the number of bytes used for some data types is non-standard. We sometimes ignore these details in favor of clarity and succinctness. Platform- and compiler-specific results are from gcc 4.9.2 running on an x86_64 compatible processor.

A.3.1 Data Types

Binary and hexadecimal

On a computer, programs and data are represented by sequences of 0’s and 1’s. A 0 or 1 may be represented by two different voltages (low and high) held and controlled by a logic circuit, for example. Each of these units of memory is called a bit.

A sequence of bits may be interpreted as a base-2 or binary number, just as a sequence of digits in the range 0 to 9 is commonly treated as a base-10 or decimal number. In the decimal numbering system, a multi-digit number like 793 is interpreted as $7 \times 10^2 + 9 \times 10^1 + 3 \times 10^0$; the rightmost column is the $10^0$ (or 1's) column, the next column to the left is the $10^1$ (or 10's) column, the next column to the left is the $10^2$ (or 100's) column, and so on. Similarly, the rightmost column of a binary number is the $2^0$ column, the next column to the left is the $2^1$ column, etc. Converting the binary number 0011011 to its decimal representation, we get

$$0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 32 + 16 + 8 + 2 + 1 = 59.$$  

The leftmost digit in a multi-digit number is called the most significant digit, and the rightmost digit, corresponding to the 1’s column, is called the least significant digit. For binary representations, these are often called the most significant bit (msb) and least significant bit (lsb), respectively.

We specify that a sequence of numbers is base-2 by writing it as 0011011 or 0b0011011, where the b stands for “binary.”

To convert a base-10 number $x$ to binary:

1. Initialize the binary result to all zeros and $k$ to a large integer, such that $2^k$ is known to be larger than $x$.

---

3 Bit is a portmanteau of binary and digit.
2. If \(2^k \leq x\), place a 1 in the \(2^k\) column of the binary number and set \(x\) to \(x - 2^k\).
3. If \(x = 0\) or \(k = 0\), we are finished. Else set \(k\) to \(k - 1\) and go to line 2.

An alternative base-10 to binary conversion algorithm builds the binary number from the rightmost to leftmost bit.

1. Divide \(x\) by 2.
2. The next digit (from right to left) is the remainder (so 1 if \(x\) is odd, 0 if \(x\) is even).
3. \(x = \) the quotient. (So if \(x\) were 5, the new \(x\) is 2, and if \(x\) were 190 the new \(x\) is 95).
4. Repeat process until \(x = 0\).

Compared to base-10, base-2 has a closer connection to actual hardware. Binary can be inconvenient for human reading and writing, however, due to the large number of digits. Therefore we often group four binary digits together (taking values \(0b0000\) to \(0b1111\), or 0 to 15 in base-10) and represent them with a single character using the numbers 0 to 9 or the letters A to F. This base-16 representation is called hexadecimal or hex for short: Thus we can

<table>
<thead>
<tr>
<th>base-2 (binary)</th>
<th>base-16 (hex)</th>
<th>base-10 (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
<td>15</td>
</tr>
</tbody>
</table>

write the eight-digit binary number \(0b00111011\), or \(0011\ 1011\), more succinctly in hex as \(3B\), or \(3\text{B}_{16}\) or \(0x3B\) to clarify that it is a hex number. The corresponding decimal number is \(3 \times 16^1 + 11 \times 16^0 = 59\).

Bits, bytes, and data types

Bits of memory are grouped together in groups of eight called bytes. A byte can be written in binary or hexadecimal (e.g., \(0b00111011\) or \(0x3B\)), and can represent values between 0 and \(2^8 - 1 = 255\). Sometimes the four bits represented by a single hex digit are referred to as a nibble. (Get it?)

A word is a grouping of multiple bytes. The number of bytes in a word depends on the processor, but four-byte words are common, as with the PIC32. A word \(01001101\ 11110101\ 10000011\ 1100111\) in binary can be written in hexadecimal as \(0x4DFA83C7\). The most significant byte (MSB) is the leftmost byte, \(0x4D\) in this case, and the least significant byte
(LSB) is the rightmost byte \(0xC7\). The msb is the leftmost bit of the MSB, a 0 in this case, and the lsb is the rightmost bit of the LSB, a 1 in this case.

A byte is the smallest unit of memory that has its own address. The address of the byte is a number that represents the byte’s location in memory. Suppose your computer has 4 gigabytes (GB), or \(4 \times 2^{30} = 2^{32}\) bytes, of RAM. Then to find the value stored in a particular byte, you need at least 32 binary digits (8 hex digits or 4 bytes) to specify the address.

An example showing the first eight addresses in memory is given below. Here we show the lowest address on the right, but we could have made the opposite choice.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001101</td>
<td>00100111</td>
</tr>
<tr>
<td>0110001</td>
<td>01010111</td>
</tr>
<tr>
<td>01010011</td>
<td>00011110</td>
</tr>
<tr>
<td>10111011</td>
<td>01100010</td>
</tr>
</tbody>
</table>

Assume that the byte at address 4 is part of the representation of a variable. Do these 0’s and 1’s represent an integer, or part of an integer? A number with a fractional component? Something else?

The answer lies in the type of the variable at that address. In C, before you use a variable, you must define it and its type, telling the compiler how many bytes to allocate for the variable (its size) and how to interpret the bits.

The most common data types come in two flavors: integers and floating point numbers (numbers with a decimal point). Of the integers, the two most common types are char, often used to represent keyboard characters, and int. Of the floating point numbers, the two most common types are float and double. As we will see shortly, a char uses 1 byte and an int usually uses 4, so two possible interpretations of the data held in the eight memory addresses could be

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001101</td>
<td>00100111</td>
</tr>
<tr>
<td>0110001</td>
<td>01010111</td>
</tr>
<tr>
<td>01010011</td>
<td>00011110</td>
</tr>
<tr>
<td>10111011</td>
<td>01100010</td>
</tr>
</tbody>
</table>

\(4\) In common usage, a kilobyte (KB) is \(2^{10} = 1024\) bytes, a megabyte (MB) is \(2^{20} = 1,048,576\) bytes, a gigabyte is \(2^{30} = 1,073,741,824\) bytes, and a terabyte (TB) is \(2^{40} = 1,099,511,627,776\) bytes. To remove confusion with the common SI prefixes that use powers of 10 instead of powers of 2, these are sometimes referred to instead as kibibyte, mebibyte, gibibyte, and tebibyte, where the “bi” refers to “binary.”

\(5\) In C you can declare or define a variable. They use similar syntax, but a declaration simply gives the name and the type of the variable, while a definition also allocates the memory to hold it. We avoid using the distinction for now and just call everything a definition.

\(6\) char is derived from the word “character.” People pronounce char variously as “car” (as in “driving the car”), “care” (a shortening of “character”), and “char” (as in charcoal), and some just punt and say “character.”

For more information visit www.nu32.org
where byte 0 is used to represent a **char** and bytes 4-7 are used to represent an **int**, or

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>11001101</td>
<td>00100111</td>
<td>01110001</td>
<td>01010111</td>
<td>01010011</td>
<td>00011110</td>
<td>10111011</td>
<td>01100010</td>
</tr>
<tr>
<td>Value</td>
<td>char</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where bytes 0-3 are used to represent an **int** and byte 4 represents a **char**. Fortunately we do not usually have to worry about how variables are packed into memory.

Below we describe the common data types. Although the number of bytes used for each type is not the same for every processor, the numbers given are common on modern computers. (Differences for the PIC32 are noted in Table A.1.) Example syntax for defining variables is also given. Note that most C statements end with a semicolon.

**char**

**Example definition:**

```c
char ch;
```

This syntax defines a variable named `ch` to be of type `char`. **Chars** are the smallest data type, using only one byte. They are often interpreted according to the “ASCII table” (pronounced “ask-key”), the American Standard Code for Information Interchange, which maps the values 0 to 127 to letters, numbers, and other characters (Figure A.1). (The values 128 to 255 map to an “extended” ASCII table.) For example, the values 48 to 57 map to the characters '0' to '9', 65 to 90 map to the uppercase letters 'A' to 'Z', and 97 to 122 map to the lowercase letters 'a' to 'z'. The assignments

```c
ch = 'a';
```

and

```c
ch = 97;
```

are equivalent, as C equates characters inside single quotation marks with their ASCII table numerical value.

Depending on the C implementation, **char** may be treated by default as **unsigned**, i.e., taking values from 0 to 255, or **signed**, taking values from −128 to 127. If you use the **char** to represent a standard ASCII character, the distinction does not matter. If, however, you use the **char** data type for integer math on small integers, you should use the specifier **signed** or **unsigned**, as appropriate. For example, we could use the following definitions, where everything after `//` is a comment:
## ASCII Table

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>9</td>
<td>LF</td>
</tr>
<tr>
<td>10</td>
<td>VT</td>
</tr>
<tr>
<td>11</td>
<td>FF</td>
</tr>
<tr>
<td>12</td>
<td>CR</td>
</tr>
<tr>
<td>13</td>
<td>SUB</td>
</tr>
<tr>
<td>14</td>
<td>DB</td>
</tr>
<tr>
<td>15</td>
<td>VT</td>
</tr>
</tbody>
</table>

The 128 standard ASCII characters. The first 32 characters are non-printing characters and the names of most of them are obscure. Values 128 to 255 (or −128 to −1) correspond to the extended ASCII table.

```c
unsigned char ch1; // ch1 can take values 0 to 255
signed char ch2;  // ch2 can take values -128 to 127
```

Example definition:
```
int i,j;
signed int k;
signed n;
```

Integers typically use four bytes (32 bits) and take values from $-(2^{31})$ to $2^{31} - 1$ (approximately $\pm 2$ billion). In the example syntax, each of $i$, $j$, $k$, and $n$ are defined to be the same data type.

We can use specifiers to get the following integer data types: `unsigned int` or simply `unsigned`, a four-byte integer taking nonnegative values from 0 to $2^{32} - 1$; `short int`, `short`, `signed short`, or `signed short int`, all meaning the same thing: a two-byte integer taking values from $-(2^{15})$ to $2^{15} - 1$ (i.e., −32,768 to 32,767); `unsigned short int` or `unsigned short`, a two-byte integer taking nonnegative values from 0 to $2^{16} - 1$ (i.e., 0 to 65,535); `long int`, `long`, `signed long`, or `signed long int`, often consisting of eight bytes and representing values from $-(2^{63})$ to $2^{63} - 1$; and `unsigned long int` or `unsigned long`, an eight-byte integer taking nonnegative values from 0 to $2^{64} - 1$. A `long long int` data type may also be available.
**Example definition:**

```
float x;
```

This syntax defines the variable `x` to be a four-byte “single-precision” floating point number.

**Example definition:**

```
double x;
```

This syntax defines the variable `x` to be an eight-byte “double-precision” floating point number. The data type `long double` (quadruple precision) may also be available, using 16 bytes (128 bits). These types allow the representation of larger numbers, to more decimal places, than single-precision `float`s.

The sizes of the data types, both on a typical x86_64 computer with gcc and on the PIC32, are summarized in Table A.1. Note the differences; C lets the compiler determine these details. The C99 standard introduces data types such as `int32_t` (32-bit signed integer) and `unit8_t` (8-bit unsigned integer) which are guaranteed to be the specified size across platforms and compilers.

<table>
<thead>
<tr>
<th>Type</th>
<th># bytes on x86_64</th>
<th># bytes on PIC32</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>long double</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

Using the data types

If your program requires floating point calculations, you can choose between `float`, `double`, and `long double` data types. The advantages of smaller types are that they use less memory and computations with them (e.g., multiplies, square roots, etc.) may be faster. The advantage of the larger types is the greater precision in the representation (e.g., smaller roundoff error).

If your program needs integer calculations, you should use integer rather than floating point data types due to the higher speed of integer math and the ability to represent a larger range of
integers for the same number of bytes. You can decide whether to use signed or unsigned chars, or \{signed/unsigned\} \{short/long\} ints. The considerations are memory usage, possibly the time of the computations, and whether the type can represent a sufficient range of integer values. For example, if you decide to use unsigned chars for integer math to save memory, and you add two of them with values 100 and 240 and assign to a third unsigned char, you will get a result of 84 due to integer overflow. This example is illustrated in the program overflow.c in Section A.4.

Representations of data types

A simple representation for integers is the sign and magnitude representation. In this representation, the msb represents the sign of the number (0 = positive, 1 = negative), and the remaining bits represent the magnitude of the number. The sign and magnitude method represents zero twice (positive and negative zero) and is not often used.

A more common representation for integers is called two’s complement. This method also uses the msb as a sign bit, but it only has a single representation of zero. The two’s complement representation of an 8-bit char is given below:

<table>
<thead>
<tr>
<th>binary</th>
<th>signed char, base-10</th>
<th>unsigned char, base-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01111111</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>10000000</td>
<td>-128</td>
<td>128</td>
</tr>
<tr>
<td>10000001</td>
<td>-127</td>
<td>129</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111111</td>
<td>-1</td>
<td>255</td>
</tr>
</tbody>
</table>

As the binary representation is incremented, the two’s complement (signed) interpretation of the binary number also increments, until it “wraps around” to the most negative value when the msb becomes 1 and all other bits are 0. The signed value then resumes incrementing until

---

7 Just as a four-byte float can represent fractional values that a four-byte int cannot, a four-byte int can represent more integers than a four-byte float can. See the type conversion example program typecast.c in Section A.4 for an example.

8 Computations with smaller data types are not always faster than with larger data types. It depends on the architecture.
it reaches $-1$ when all bits are 1. To negate a number using two’s complement arithmetic, invert all of the bits and add one. For example, 1 (0b00000001) becomes $-1$ (0b11111111). What happens when you perform the negation procedure on zero?

Another representation choice is **endianness**. The little-endian representation of an int stores the least significant byte at the lowest address (ADDRESS) and the most significant byte at highest (ADDRESS+3) (remember, little-lowest-least), while the big-endian convention is the opposite, storing the most significant byte at the lowest address (ADDRESS) and the least significant byte at the highest address (ADDRESS+3). The convention used depends on the processor. For definiteness in this appendix, we always assume little-endian representation, which is used by x86_64 (most likely your computer’s architecture) and the PIC32.

**floats, doubles, and long doubles** are commonly represented in the IEEE 754 floating point format

$$\text{value} = (-1)^s \times b \times 2^c, \quad (A.1)$$

where one bit is used to represent the sign ($s = 0$ for positive, $s = 1$ for negative); $m = 23/52/112$ bits are used to represent the significand $b$ (also known as the mantissa) in the range $1$ to $2 - 2^{-m}$; and $n = 8/11/15$ bits are used to represent the exponent $c$ in the range $-(2^n-1) + 2$ to $2^n-1 - 1$, where $n$ and $m$ depend on whether the type uses 4/8/16 bytes. Certain exponent and significand combinations are reserved for representing special cases like positive and negative infinity and “not a number” ($\text{NaN}$).

Specifically for a four-byte float, the 32 bits of the IEEE 754 representation are

$$\begin{align*}
\underline{s} & \quad \underline{\text{8 bits of exponent } c} \quad \underline{\text{23 bits of significand } b} \\
\text{sign bit} & \quad e_7 e_6 e_5 e_4 e_3 e_2 e_1 e_0 & \quad \underline{f_{23} f_{22} \cdots f_2 f_1}
\end{align*}$$

The exponent $c$ in (A.1) is equal to $e - 127$, where $e$ is the unsigned integer value of the eight bits of $e$, ranging from 0 to $2^8 - 1 = 255$. (The values $e = 0$ and $e = 255$ are reserved to represent special cases, like ±infinity and “not a number.”) The significand $b$ in (A.1) is given by

$$b = 1 + \sum_{i=1}^{23} f_i \ 2^{-i},$$

so $b$ ranges from 1 to $2 - 2^{-23}$.

---

9 These phrases come from *Gulliver’s Travels* by Jonathan Swift, where Lilliputians fanatically divide themselves according to which end of a soft-boiled egg they crack open.

For more information visit www.nu32.org
See Exercise 14 to experiment with two’s complement and IEEE 754 floating point representations.

Rarely do you need to worry about the specific bit-level representation of the different data types: endianness, two’s complement, IEEE 754, etc. You tell the compiler to store values and retrieve values, and it handles implementing the representations.

A.3.2 Memory, Addresses, and Pointers

Consider the following C syntax:

```c
int i;
int *ip;
```

These definitions may appear to define the variables `i` and `*ip` of type `int`; however, the character `*` cannot be part of a variable name. The variable name is actually `ip`, and the special character `*` means that `ip` is a **pointer** to something of type `int`. Pointers store the address of another variable; in other words, they “point” to another variable. We often use the words “address” and “pointer” interchangeably.

When the compiler sees the definition `int i`, it allocates four bytes of memory to hold the integer `i`. When the compiler sees the definition `int *ip`, it creates the variable `ip` and allocates to it whatever amount of memory is needed to hold an address, a platform-dependent quantity. The compiler also remembers the data type that `ip` points to, `int` in this case, so when you use `ip` in a context that requires a pointer to a different data type, the compiler may generate a warning or an error. Technically, the type of `ip` is “pointer to type `int`.”

**Important!** Defining a pointer only allocates memory to hold the pointer. It does not allocate memory for a pointee variable to be pointed at. Also, simply defining a pointer does not initialize it to point to anything valid. Do not use pointers without explicitly initializing them!

When we want the address of a variable, we apply the **address** (or **reference**) operator to the variable, which returns a pointer to the variable (its address). In C, the address operator is written `&`. Thus the following command assigns the address of `i` to the pointer `ip`:

```c
ip = &i;  // ip now holds the address of i
```

---

10 When computers switched from the 32-bit x86 to the 64-bit x86_64 architecture, code that relied on pointers being 32 bits long were in trouble; x86_64 uses 64-bit long pointers!
The address operator always returns the lowest address of a multi-byte type. For example, if the four-byte int i occupies addresses 0x0004 to 0x0007 in memory, &i will return 0x0004.\footnote{The address may actually be a “virtual” address rather than a physical location in memory. The computer automatically translates the value of &i to an actual physical address, when needed.}

If we have a pointer (an address) and we want the contents at that address, we apply the dereference operator to the pointer. In C, the dereference operator is written *. Thus the following command stores the value at the address pointed to by ip in i:

\[
i = *ip; \quad \text{// i now holds the contents at the address ip}
\]

However, you should never dereference a pointer until it has been initialized to point to something using a statement such as ip = &i.

As an analogy, consider the pages of a book. A page number can be considered a pointer, while the text on the page can be considered the contents of a variable. So the notation &text would return the page number (pointer or address) of the text, while *page_number would return the text on that page (but only after page_number is initialized to point at a page of text).

Even though we are focusing on the concept of pointers, and not C syntax, let us look at some sample C code, remembering that everything after // on the same line is a comment:

```c
int i,j; \quad \text{// define i, j as type int}
int *ip; \quad \text{// define ip as type "pointer to type int"}
ip = &i; \quad \text{// set ip to the address of i (& "references" i)}
i = 100; \quad \text{// put the value 100 in the location allocated by the compiler for i}
j = *ip; \quad \text{// set j to the contents of the address ip (* dereferences ip),}
\quad \text{// i.e., 100}
j = j+2; \quad \text{// add 2 to j, making j equal to 102}
i = *(&j); \quad \text{// & references j to get the address, then * gets contents; i is set}
\quad \text{// to 102}
*(&j) = 200; \quad \text{// content of the address of j (j itself) is set to 200; i is unchanged}
```

The use of pointers can be powerful, but also dangerous. For example, you may accidentally try to access an illegal memory location. The compiler is unlikely to recognize this during compilation, and you may end up with a “segmentation fault” when you execute the code.\footnote{A good name for a program like this is coredumper.c.}

This kind of bug can be difficult to find, and dealing with it is a C rite of passage. More on pointers in Section A.4.8.

For more information visit www.nu32.org
A.3.3 Compiling

The process loosely referred to as “compilation” actually consists of four steps:

1. **Preprocessing.** The preprocessor takes the `program.c` source code and produces an equivalent `.c` source code, performing operations such as removing comments. Section A.4.3 discusses the preprocessor in more detail.

2. **Compiling.** The compiler turns the preprocessed code into *assembly* code for the specific processor. The C code becomes a set of instructions that directly correspond to actions that the processor can perform. The compiler can be configured with several options that impact the assembly code generated. For example, the compiler can generate assembly code that increases execution time to reduce the amount of memory needed to store the code. Assembly code generated by a compiler can be inspected with a standard text editor. Coding directly in assembly is still a popular, if painful (or fun), way to program microcontrollers.

3. **Assembling.** The assembler converts the assembly instructions into processor-dependent machine-level binary *object* code. This code cannot be examined using a text editor. Object code is called *relocatable*, in that the exact memory addresses for the data and program statements are not specified.

4. **Linking.** The linker takes one or more object code files and produces a single executable file. For example, if your code includes pre-compiled libraries, such as the C standard library that allows you to print to the screen (described in Sections A.4.3 and A.4.14), this code is included in the final executable. The data and program statements in the various object code files are assigned to specific memory locations.

In our `HelloWorld.c` program, this entire process is initiated by the single command line statement:

```
> gcc HelloWorld.c -o HelloWorld
```

If our `HelloWorld.c` program used any mathematical functions in Section A.4.7, the compilation would be initiated by:

```
> gcc HelloWorld.c -o HelloWorld -lm
```

where the `-lm` flag tells the linker to link the math library, which may not be linked by default like other libraries are.

If you want to see the intermediate results of the preprocessing, compiling, and assembling steps, Exercise 42 provides an example.

For more complex projects requiring compilation of several files into a single executable or specifying various options to the compiler, it is common to create a *Makefile* that specifies

---

13 Well, you can view it in a text editor, but it will be incomprehensible.
how the compilation should be performed, and to then use the command `make` to actually execute the commands to create the executable. Details on the use of `Makefile` are beyond the scope of this appendix; however, we use one extensively when programming the PIC32. Section A.4.15 gives a simple example of compiling multiple C files into a single executable using a `Makefile`.

### A.4 C Syntax

So far we have seen only glimpses of C syntax. Let us begin our study of C syntax with a few simple programs. We then jump to a more complex program, `invest.c`, that demonstrates many of the major elements of C structure and syntax. If you can understand `invest.c` and can create programs using similar elements, you are well on your way to mastering C. We defer the more detailed descriptions of the syntax until after introducing `invest.c`.

**Printing to screen**

Because it is the simplest way to see the results of a program, as well as a useful tool for debugging, let us start with the function `printf` for printing to the screen.\(^\text{14}\) We have already seen it in `HelloWorld.c`. Here’s a slightly more interesting example. Let us call this program file `printout.c`.

```c
#include <stdio.h>

int main(void) {
    int i;
    float f;
    double d;
    char c;

    i = 32;
    f = 4.278;
    d = 4.278;
    c = 'k'; // or, by ASCII table, c = 107;

    printf("Formatted output:\n");
    printf("i = %4d c = '%c'\n",i,c);
    printf("f = %19.17f\n",f);
    printf("d = %19.17f\n",d);
    return 0;
}
```

\(^\text{14}\) Programs called debuggers (such as gdb) also help you debug, allowing you to step through your program line by line as it runs.
The first line of the program

```c
#include <stdio.h>
```

tells the preprocessor that the program will use functions from the “standard input and output” library, one of many code libraries provided in standard C installations that extend the power of the language. The `stdio.h` function used in `printout.c` is `printf`, covered in more detail in Section A.4.14.

The next line

```c
int main(void) {
```

starts the block of code that defines the `main` function. The `main` code block is closed by the final closing brace `}`. Each C program has exactly one `main` function, and program execution begins there. The type of `main` is `int`, meaning that the function should end by returning a value of type `int`. In our case, it returns a 0, which indicates to the operating system that the program has terminated successfully.

The next four lines define and allocate memory for four variables with four different types. The following lines assign values to those variables. The `printf` lines will be discussed after we look at the output.

Now that you have created `printout.c`, you can create the executable file `printout` and run it from the command line. Make sure you are in the directory containing `printout.c`, then type the following:

```sh
> gcc printout.c -o printout
> printout
```

(Again, you may have to use `./printout` to tell your computer to look in the current directory.)

Here is the output:

```
Formatted output:
i = 32 c = 'k'
f = 4.2779987792968750
d = 4.2779999999999958
```

The main purpose of this program is to demonstrate formatted output from the code

```c
printf("Formatted output:\n");
printf(" i = %4d c = '%c'\n",i,c);
printf(" f = %19.17f\n",f);
printf(" d = %19.17f\n",d);
```

Inside the function call to `printf`, everything inside the double quotation marks is printed to the screen, but some character sequences have special meaning. The `\n` sequence creates a newline. The `%` is a special character, indicating that some data will be printed, and for each `%
in the double quotes, there must be a variable or other expression in the comma-separated list at the end of the \texttt{printf} call. The \texttt{%4d} means that an \texttt{int} type variable is expected, and it will be displayed right-justified using four spaces. (If the number has more than four digits, it will take as much space as is needed.) The \texttt{%c} means that a \texttt{char} is expected. The \texttt{%19.17f} means that a \texttt{float} or \texttt{double} will be printed right-justified over 19 spaces with 17 spaces after the decimal point. If you did not care how many decimal places were displayed, you could have simply used \texttt{%f} and let the C implementation default make the choice for you.\footnote{\texttt{printf} does not distinguish between doubles and floats, so use \texttt{%f} for both.} More details on \texttt{printf} can be found in Section A.4.14.

The output of the program also shows that neither the \texttt{float f} nor the \texttt{double d} can represent 4.278 exactly, though the double-precision representation comes closer.

Data sizes

Since we have focused on data types, our next program measures how much memory is used by different data types. Create a file called \texttt{datasizes.c} that looks like the following:

\begin{verbatim}
#include <stdio.h>

int main(void) {
    char a;
    char *bp;
    short c;
    int d;
    long e;
    float f;
    double g;
    long double h;
    long double *ip;

    printf("Size of char: %2ld bytes\n", sizeof(a)); // "% 2 ell d"
    printf("Size of char pointer: %2ld bytes\n", sizeof(bp));
    printf("Size of short int: %2ld bytes\n", sizeof(c));
    printf("Size of int: %2ld bytes\n", sizeof(d));
    printf("Size of long int: %2ld bytes\n", sizeof(e));
    printf("Size of float: %2ld bytes\n", sizeof(f));
    printf("Size of double: %2ld bytes\n", sizeof(g));
    printf("Size of long double: %2ld bytes\n", sizeof(h));
    printf("Size of long double pointer: %2ld bytes\n", sizeof(ip));
    return 0;
}
\end{verbatim}

The first lines in the \texttt{main} function define nine variables, telling the compiler to allocate space for these variables. Two of these variables are pointers. The \texttt{sizeof()} operator returns the number of bytes allocated in memory for its argument. You can use \texttt{sizeof()} on either a variable or a type (i.e., \texttt{sizeof(int)}); here we use it exclusively on variables.
Here is the output of the program:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 bytes</td>
</tr>
<tr>
<td>char pointer</td>
<td>8 bytes</td>
</tr>
<tr>
<td>short int</td>
<td>2 bytes</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
</tr>
<tr>
<td>long int</td>
<td>8 bytes</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
</tr>
<tr>
<td>long double</td>
<td>16 bytes</td>
</tr>
<tr>
<td>long double pointer</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

We see that, on an x86_64 computer with gcc, ints and floats use four bytes, short ints two bytes, long ints and doubles eight bytes, and long doubles 16 bytes. Regardless of whether it points to a char or a long double, a pointer (address) uses eight bytes, meaning we can address a maximum of \((2^8)^8 = 256^8\) bytes of memory. Considering that corresponds to almost 18 quintillion bytes, or 18 billion gigabytes, we should have enough available addresses (at least for the time-being)!

## Overflow

Now let us try the program `overflow.c`, which demonstrates the issue of integer overflow mentioned in Section A.3.1.

```c
#include <stdio.h>

int main(void) {
    char i = 100, j = 240, sum;
    unsigned char iu = 100, ju = 240, sumu;
    signed char is = 100, js = 240, sums;
    sum = i+j;
    sumu = iu+ju;
    sums = is+js;
    printf("char: \%d + \%d = \%d or ASCII \%c
",i,j,sum,sum);
    printf("unsigned char: \%d + \%d = \%d or ASCII \%c
",iu,ju,sumu,sumu);
    printf("signed char: \%d + \%d = \%d or ASCII \%c
",is,js,sums,sums);
    return 0;
}
```

In this program we initialize the values of some of the variables when they are defined. You might also notice that we are assigning a signed char a value of 240, even though the range for that data type is \(-128\) to 127. So something fishy is happening. The program outputs:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>100 + -16 = 84 or ASCII T</td>
</tr>
<tr>
<td>unsigned char</td>
<td>100 + 240 = 84 or ASCII T</td>
</tr>
<tr>
<td>signed char</td>
<td>100 + -16 = 84 or ASCII T</td>
</tr>
</tbody>
</table>
Notice that, with our C compiler, `char` are the same as `signed char`. Even though we assigned the value of 240 to `js` and `j`, they contain the value \(-16\) because the binary representation of 240 has a 1 in the \(2^7\) column. For the two's complement representation of a `signed char`, this column indicates whether the value is positive or negative. Finally, we notice that the `unsigned char ju` is successfully assigned the value 240 since its range is 0 to 255, but the addition of `iu` and `ju` leads to an overflow. The correct sum, 340, has a 1 in the \(2^8\) (or 256) column, but this column is not included in the 8 bits of the `unsigned char`. Therefore we see only the remainder of the number, 84. The number 84 is assigned the character T in the standard ASCII table.

Type conversion

Continuing our focus on data types, we try another simple program that illustrates what happens when you mix data types in mathematical expressions. This program uses a helper function in addition to the `main` function. We name this program `typecast.c`.

```c
#include <stdio.h>

void printRatio(int numer, int denom) { // printRatio is a helper function
    double ratio;
    ratio = numer/denom;
    printf("Ratio, %d/%d: %5.2f\n",numer,denom,ratio);
    ratio = numer/((double) denom);
    printf("Ratio, %d/((double) %d): %5.2f\n",numer,denom,ratio);
    ratio = ((double) numer)/((double) denom);
    printf("Ratio, ((double) %d)/((double) %d): %5.2f\n",numer,denom,ratio);
}

int main(void) {
    int num = 5, den = 2;
    printRatio(num,den); // call the helper function
    return(0);
}
```

The helper function `printRatio` “returns” type `void` since it does not return a value. It takes two `int`s as arguments and calculates their ratio in three different ways. In the first, the two `int`s are divided and the result is assigned to a `double`. In the second, the integer `denom` is `typecast` or `cast` to `double` before the division occurs, so an `int` is divided by a `double` and the result is assigned to a `double`.\(^{16}\) In the third, both the numerator and denominator are cast as `doubles` before the division, so two `doubles` are divided and the result is assigned to a `double`.

\(^{16}\) The typecasting does not change the variable `denom` itself; it simply creates a temporary `double` version of `denom` which is lost as soon as the division is complete.
The `main` function defines two `ints`, `num` and `den`, and passes their values to `printRatio`, where those values are copied to `numer` and `denom`, respectively. The variables `num` and `den` are only available to `main`, and the variables `numer` and `denom` are only available to `printRatio`, since they are defined inside those functions.

Execution of any C program always begins with the `main` function, regardless of where it appears in the file.

After compiling and running, we get the output

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio, 5/2:</td>
<td>2.00</td>
</tr>
<tr>
<td>Ratio, 5/((double) 2):</td>
<td>2.50</td>
</tr>
<tr>
<td>Ratio, ((double) 5)/((double) 2):</td>
<td>2.50</td>
</tr>
</tbody>
</table>

The first answer is “wrong,” while the other two answers are correct. Why?

The first division, `numer/denom`, is an *integer* division. When the compiler sees that there are `ints` on either side of the divide sign, it assumes you want integer math and produces a result that is an `int` by simply truncating any remainder (rounding toward zero). This value, 2, is then converted to the floating point number 2.0 so it can be assigned to the double-precision floating point variable `ratio`. On the other hand, the expression `numer/((double) denom)`, by virtue of the parentheses, first produces a double version of `denom` before performing the division. The compiler recognizes that you are dividing two different data types, so it temporarily *coerces* the `int` to a `double` so it can perform a floating point division. This is equivalent to the third and final division, except that the typecast of the numerator to `double` is explicit in the code for the third division.

Thus we have two kinds of type conversions:

- **Implicit** type conversion, or *coercion*. This occurs, for example, when a type has to be converted to carry out a variable assignment or to allow a mathematical operation. For example, dividing an `int` by a `double` will cause the compiler to treat the `int` as a `double` before carrying out the division.

- **Explicit** type conversion. An explicit type conversion is coded using a casting operator, e.g., `(double) <expression>` or `(char) <expression>`, where `<expression>` may be a variable or mathematical expression.

Certain type conversions may result in a change of value. For example, assigning the value of a `float` to an `int` results in truncation of the fractional portion; assigning a `double` to a `float` may result in roundoff error; and assigning an `int` to a `char` may result in overflow. Here’s a less obvious example:

```c
float f;
int i = 16777217;
f = i;  // f now has the value 16,777,216.0, not 16,777,217!
```

For more information
visit [www.nu32.org](http://www.nu32.org)
It turns out that $16,777,217 = 2^{24} + 1$ is the smallest positive integer that cannot be represented by a 32-bit float. On the other hand, a 32-bit int can represent all integers in the range $-2^{31}$ to $2^{31} - 1$.

Some type conversions, called promotions, never result in a change of value because the new type can represent all possible values of the original type. Examples include converting a char to an int or a float to a long double.

As with pointers, typecasts are dangerous and should be used sparingly. Knowing where type coercion occurs, however, can be crucial. In the example below, the first line performs integer division and then converts the result to a double, whereas the second line performs floating point division.

```c
double f = 3/2; // yields 1.0!
double g = 3.0/2.0; // yields 1.5
```

We will see more on use of parentheses (Section A.4.1), the scope of variables (Section A.4.5), and defining and calling helper functions (Section A.4.6).

---

**Advanced:** Pointers can be used in conjunction with typecasts to view the same data in different ways. For example, the declaration `unsigned short s = 0xAB12` stores 0xAB12 in memory as two consecutive bytes: 0x12 0xAB (remember, the LSB is in the lowest address on a little-endian processor). Performing `(*&s)` dereferences the address &s and treats the memory location as an unsigned short because &s has type unsigned short *; thus, the expression yields 0xAB12. Performing `*(unsigned char *)&s` yields 0x12 because the typecast converts the pointer &s into a pointer to an unsigned char *. Dereferencing such a pointer yields an unsigned char, which is only one byte long. Pointers always refer to the lowest address of the variable, so the result is 0x12 not 0xAB. On a big-endian system, however, the result would be 0xAB.

A more complete example: `invest.c`

Until now we have been dipping our toes in the C pool. Now let us dive in headfirst.

Our next program is called `invest.c`, which takes an initial investment amount, an expected annual return rate, and a number of years, and returns the growth of the investment over the years. After performing one set of calculations, it prompts the user for another scenario, and continues this way until the data entered is invalid. The data is invalid if, for example, the initial investment is negative or the number of years to track is outside the allowed range.

The real purpose of `invest.c`, however, is to demonstrate the syntax and several useful features of C.

Here’s an example of compiling and running the program. The only data entered by the user are the three numbers corresponding to the initial investment, the growth rate, and the number of years.

For more information visit [www.nu32.org](http://www.nu32.org)
Before we look at the full `invest.c` program, let us review two principles that should be adhered to when writing a longer program: modularity and readability.

- **Modularity.** You should break your program into a set of functions that perform specific, well-defined tasks, with a small number of inputs and outputs. As a rule of thumb, no function should be longer than about 20 lines. (Experienced programmers often break this rule of thumb, but if you are a novice and are regularly breaking this rule, you are likely not thinking modularly.) Almost all variables you define should be “local” to (i.e., only recognizable by) their particular function. Global variables, which can be accessed by all functions, should be minimized or avoided altogether, since they break modularity, allowing one function to affect the operation of another without the information passing through the well-defined “pipes” (input arguments to a function or its returned results). If you find yourself typing the same (or similar) code more than once, that’s a good sign you should determine how to write a single function and just call that function from multiple places. Modularity makes it much easier to develop large programs and track down the inevitable bugs.

- **Readability.** You should use comments to help other programmers, and even yourself, understand the purpose of the code you have written. Variable and function names should be chosen to indicate their purpose. Be consistent in how you name variables and functions. Any “magic number” (constant) used in your code should be given a name and defined at the beginning of the program, so if you ever want to change this number, you can just change it at one place in the program instead of every place it is used. Global variables and constants should be written in a way that easily distinguishes them from more common local variables; for example, you could WRITE CONSTANTS IN UPPERCASE and Capitalize Globals. You should use whitespace (blank lines, spaces, tabbing, etc.) consistently to make it easy to read the program. Use a fixed-width font.
(e.g., Courier) so that the spacing/tabbing is consistent. Modularity (above) also improves readability.

The program `invest.c` demonstrates readable modular code using the structure and syntax of a typical C program. In the program’s comments, you will see references of the form `==SecA.4.3==` that indicate where you can find more information in the review of syntax that follows the program.

```c
/******************************************************************************
 * PROGRAM COMMENTS (PURPOSE, HISTORY)
 * ***************************************************************************/

/*
 * invest.c
 *
 * This program takes an initial investment amount, an expected annual
 * return rate, and the number of years, and calculates the growth of
 * the investment. The main point of this program, though, is to
 * demonstrate some C syntax.
 *
 * References to further reading are indicated by ==SecA.B.C==
 */

/******************************************************************************
 * PREPROCESSOR COMMANDS ==SecA.4.3==
 * ***************************************************************************/
#include <stdio.h> // input/output library
#define MAX_YEARS 100 // constant indicating max number of years to track

/******************************************************************************
 * DATA TYPE DEFINITIONS (HERE, A STRUCT) ==SecA.4.4==
 * ***************************************************************************/
typedef struct {
    double inv0; // initial investment
    double growth; // growth rate, where 1.0 = zero growth
    int years; // number of years to track
    double invarray[MAX_YEARS+1]; // investment array ==SecA.4.9==
} Investment; // the new data type is called Investment

/******************************************************************************
 * GLOBAL VARIABLES ==SecA.4.2, A.4.5==
 * ***************************************************************************/
// no global variables in this program

/******************************************************************************
 * HELPER FUNCTION PROTOTYPES ==SecA.4.2==
 * ***************************************************************************/
int getUserInput(Investment *invp); // invp is a pointer to type ...
```
void calculateGrowth(Investment *invp); // ... Investment ==SecA.4.6, A.4.8==
void sendOutput(double *arr, int years);

/***************************************************************************/
* MAIN FUNCTION ==SecA.4.2==
/***************************************************************************/
int main(void) {
Investment inv; // variable definition. ==SecA.4.5==

while(getUserInput(&inv)) { // while loop ==SecA.4.13==
    inv.invarray[0] = inv.inv0; // struct access ==SecA.4.4==
    calculateGrowth(&inv); // & referencing (pointers) ==SecA.4.6, A.4.8==
    sendOutput(inv.invarray, // passing a pointer to an array ==SecA.4.9==
               inv.years); // passing a value, not a pointer ==SecA.4.6==
}
return 0; // return value of main ==SecA.4.6==
} // ***** END main *****

/***************************************************************************/
* HELPER FUNCTIONS ==SecA.4.2==
/***************************************************************************/
/* calculateGrowth
* This optimistically-named function fills the array with the investment
* value over the years, given the parameters in *invp.
*/
void calculateGrowth(Investment *invp) {
    int i;

    // for loop ==SecA.4.13==
    for (i = 1; i <= invp->years; i = i + 1) { // relational operators ==SecA.4.10==
        // struct access ==SecA.4.4==
        invp->invarray[i] = invp->growth * invp->invarray[i-1];
    }
} // ***** END calculateGrowth *****

/* getUserInput
* This reads the user’s input into the struct pointed at by invp.
* and returns TRUE (1) if the input is valid, FALSE (0) if not.
*/
int getUserInput(Investment *invp) {
    int valid; // int used as a boolean ==SecA.4.10==

    // I/O functions in stdio.h ==SecA.4.14==
    printf("Enter investment, growth rate, number of yrs (up to %d): ",MAX_YEARS);
    scanf("%lf %lf %d", &(invp->inv0), &(invp->growth), &(invp->years));

    // logical operators ==SecA.4.11==

valid = (invp->inv0 > 0) && (invp->growth > 0) &&
   (invp->years > 0) && (invp->years <= MAX_YEARS);
printf("Valid input? %d\n", valid);

// if-else ==SecA.4.12==
if (!valid) { // ! is logical NOT ==SecA.4.11==
    printf("Invalid input; exiting.\n");
    return(valid);
} // ***** END getUserInput *****

/* sendOutput
 *
* This function takes the array of investment values (a pointer to the first
* element, which is a double) and the number of years (an int). We could
* have just passed a pointer to the entire investment record, but we decided
* to demonstrate some different syntax.
*/
void sendOutput(double *arr, int yrs) {

    int i;
    char outstring[100]; // defining a string ==SecA.4.9==

    printf("\nRESULTS:\n\n");  
    for (i=0; i<=yrs; i++) { // ++, +=, math in ==SecA.4.7==
        sprintf(outstring,"Year %3d: %10.2f\n",i,arr[i]);
        printf("%s",outstring);
    }
    printf("\n");  
} // ***** END sendOutput *****

A.4.1 Basic Syntax

Comments

Everything after a /* and before the next */ is a comment. Comments are removed during the
preprocessing step of compilation. They help make the purpose of the program, function,
loop, or statement clear to yourself or other programmers.17 Keep the comments neat and
concise for program readability. Some programmers use extra asterisks or other characters to
make the comments stand out (see the examples in invest.c), but all that matters is that
/* starts the comment and the next */ ends it.

If your comment is short, you can use // instead. Everything after // and before the next
carriage return will be ignored. The // style comments originate from C++ but most modern C
compilers support them.

17Reading your own code after several months away from it is often like reading someone else’s code!

For more information
visit www.nu32.org
Semicolons

A code statement must be completed by a semicolon. Some exceptions to this rule include preprocessor commands (see PREPROCESSOR COMMANDS in the program and Section A.4.3) and statements that end with blocks of code enclosed by braces {}. A single code statement may extend over multiple lines of the program listing until it is terminated by a semicolon (see, for example, the assignment to valid in the function getUserInput).

Braces and blocks of code

Blocks of code are enclosed in braces {}. Examples include entire functions (see the definition of the main function and the helper functions), blocks of code executed inside of a while loop (in the main function) or for loop (in the calculateGrowth and sendOutput functions), as well as other examples. In invest.c, braces are placed as shown here

```c
while (<expression>) { /* block of code */
}
```

but this style is equivalent

```c
while (<expression>)  {
    /* block of code */
}
```

as is this

```c
while (<expression>) { /* block of code */ }
```

Which brings us to...

Whitespace

Whitespace, such as spaces, tabs, and carriage returns, is only required where it is needed to recognize keywords and other syntax. The whole program invest.c could be written on a single line, for example. Indentations and line breaks should be used consistently, however, to make the program readable. Insert line breaks after each semicolon. Statements within the same code block should be left-justified with each other and statements in a code block nested within another code block should be indented with respect to the parent code block. Text editors should use a fixed-width font so that alignment is clear. Most editors provide fixed-width fonts and automatic indentation to enhance readability.

For more information
visit www.nu32.org
Parentheses

C has rules defining the order in which operations in an expression are evaluated, much like standard math rules that say \(3 + 5 \times 2\) evaluates to \(3 + (10) = 13\), not \((8) \times 2 = 16\). If uncertain about the default order of operations, use parentheses \((\quad)\) to enclose sub-expressions. For example, \(3 + (40/(4 \times (3 + 2)))\) evaluates to \(3 + (40/(4 \times 5)) = 3 + (40/20) = 3 + 2 = 5\), whereas \(3 + 40/4 \times 3 + 2\) evaluates to \(3 + 30 + 2 = 35\).

**A.4.2 Program Structure**

`invest.c` demonstrates a typical structure for a program written in one `.c` file. When you write larger programs, you may wish to divide your program into multiple files, to increase modularity. Section A.4.15 discusses C programs that consist of multiple source code files.

Let us consider the seven major sections of the program in order of appearance. **PROGRAM COMMENTS** describe the purpose of the program. **PREPROCESSOR COMMANDS** define constants and “header” files that should be included, giving the program access to library functions that extend the power of the C language. This section is described in more detail in Section A.4.3. In some programs, it may be helpful to define a new data type, as shown in **DATA TYPE DEFINITIONS**. In `invest.c`, several variables are packaged together in a single record or struct data type, as described in Section A.4.4. Any **GLOBAL VARIABLES** are then defined. These are variables that are available for use by all functions in the program. Because of this special status, the names of global variables could be Capitalized or otherwise written in a way to remind the programmer that they are not local variables (Section A.4.5). Generally, global variables should be avoided because they violate modularity.

The next section of the program contains the **HELPER FUNCTION PROTOTYPES** of the various helper functions. A prototype of a function declares the name, argument types, and return types of a function that will be defined later. Prototypes are used to allow code to call functions that have not yet been fully defined or are defined elsewhere (perhaps in another source file). For example, the function `printRatio` has a return type of `void`, meaning that it does not return a value. It takes two arguments, each of type `int`. The function `getUserInput` returns an `int` and takes a single argument: a pointer to a variable of type `Investment`, a data type defined a few lines above the `getUserInput` prototype.

The next section of the program, **MAIN FUNCTION**, is where the `main` function is defined. Every program has exactly one `main` function, where the program starts execution. The `main` function returns an `int`. By convention, it returns 0 if it executes successfully, and otherwise returns a nonzero value. In `invest.c`, `main` takes no arguments, hence the `void` in the argument list. Some programs accept arguments on the command line; these can be passed as arguments to `main`. For example, we could have written `invest.c` to run with a command such as this:
To allow this, `main` would have been defined with the following syntax:

```c
int main(int argc, char *argv[]) {
```

Then when the program is invoked as above, the integer `argc` would be set to four, the number of whitespace-separated strings on the command line, and `argv` would point to a array of four strings, where the string `argv[0]` is ’invest’, `argv[1]` is ’100.0’, etc. You can learn more about arrays and strings in Section A.4.9.

Finally, the last section of the program is the definition of the HELPER FUNCTIONS whose prototypes were given earlier. It is not strictly necessary that the helper functions have prototypes, but if not, every function should be defined before it is used by any other function. For example, none of the helper functions uses another helper function, so they could have all been defined before the `main` function, in any order, and their function prototypes eliminated. The names of the variables in a function prototype and in the actual definition of the function need not be the same; for example, the prototype of `sendOutput` uses variables named `arr` and `years`, whereas the actual function definition uses `arr` and `yrs`. What matters is that the prototype and actual function definition have the same number of arguments, of the same types, and in the same order. In fact, in the arguments of the function prototypes, you can leave out variable names altogether, and just keep the comma separated list of argument data types; however, including the names serves as additional documentation and is generally a good practice.

### A.4.3 Preprocessor Commands

In the preprocessing stage of compilation, all comments are removed from the program. Additionally, the preprocessor performs actions when encountering the following preprocessor commands, recognizable by the `#` character:

```c
#include <stdio.h> // input/output header
#define MAX_YEARS 100 // constant indicating max number of years to track
```

Include files

The first preprocessor command in `invest.c` indicates that the program will use standard C input/output functions. The file `stdio.h` is called a header file for the library. This file is readable by a text editor and contains constants, function prototypes, and other included headers that are made available to the program. The preprocessor replaces the
#include <stdio.h> command with the contents of the header file stdio.h. Examples of function prototypes that are included are

```c
int printf(const char *Format, ...);
int sprintf(char *Buffer, const char *Format, ...);
int scanf(const char *Format, ...);
```

Each of these three functions is used in invest.c. If the program were compiled without including stdio.h, the compiler would generate a warning or an error due to the lack of function prototypes. See Section A.4.14 for more information on using thestdio input and output functions.

During the linking stage, the object code of invest.c is linked with the object code for printf, sprintf, and scanf in your C installation. Libraries like the C standard library provide access to functions beyond the basic C syntax. Other useful libraries (and header files for the C standard library) are briefly described in Section A.4.14.

Constants

The second line defines the constant `MAX_YEARS` to be equal to 100. The preprocessor searches for each instance of `MAX_YEARS` in the program and replaces it with 100. If we later decide that the maximum number of years to track investments should be 200, we can change the definition of this constant in one place, instead of in several places. Since `MAX_YEARS` is constant, not a variable, it can never be assigned another value somewhere else in the program. To indicate that it is not a variable, a common convention is to write constants in UPPERCASE. This is not required by C, however. We should emphasize that the preprocessor performs a text substitution, as if you used the “find and replace” feature of your text editor. So, in this example, the preprocessor literally replaces every occurrence of `MAX_YEARS` with the number 100.

Macros

One more use of the preprocessor is to define simple function-like macros that you may use in more than one place in your program. Constants, as described above, are technically a simple macro. Here’s an example that converts radians to degrees:

```c
#define RAD_TO_DEG(x) ((x) * 57.29578)
```

---

18 The preprocessor searches for header files in directories specified by the “include path.” If the header file header.h sits in the same directory as invest.c, we would write `#include "header.h"` instead of `#include <header.h>`. 

For more information
visit www.nu32.org
The preprocessor searches for any instance of `RAD_TO_DEG(x)` in the program, where `x` can be any text, and replaces it with `((x) * 57.29578)`. For example, the initial code

```c
angle_deg = RAD_TO_DEG(angle_rad);
```

is replaced by

```c
angle_deg = ((angle_rad) * 57.29578);
```

Note the importance of the outer parentheses in the macro definition. If we had instead used the preprocessor command

```c
#define RAD_TO_DEG(x) (x) * 57.29578 // don't do this!
```
then the code

```c
answer = 1.0 / RAD_TO_DEG(3.14);
```

would be replaced by

```c
answer = 1.0 / (3.14) * 57.29578;
```

which is very different from

```c
answer = 1.0 / ((3.14) * 57.29578);
```

Moral: if the expression you are defining is anything other than a single constant, enclose it in parentheses, to tell the compiler to evaluate the expression first.

As a second example, the macro

```c
#define MAX(A,B) ((A) > (B) ? (A):(B))
```

returns the maximum of two arguments. The `?` is the ternary operator in C, which has the form

```c
<test> ? return_value_if_test_is_true : return_value_if_test_is_false
```

The preprocessor replaces

```c
maxval = MAX(13+7,val2);
```
with

\[
\text{maxval} = ((13+7) > (val2)) ? (13+7):(val2));
\]

Why define a macro instead of just writing a function? One reason is that the macro may execute slightly faster, since no passing of control to another function and no passing of variables is needed. Most of the time, you should use functions.

### A.4.4 Typedefs, Structs, and Enums

In simple programs, you will do just fine with the data types `int`, `char`, `float`, `double`, and variations. Sometimes you may find it useful to create an alias for a data type, using the following syntax:

```c
typedef <type> newtype;
```

where `<type>` is an existing data type and `newtype` is its alias. Then, you can define a new variable `x` of type `newtype` by

```c
newtype x;
```

For example, you could write

```c
typedef int days_of_the_month;
days_of_the_month day;
```

You might find it satisfying that your variable `day` (taking values 1 to 31) is of type `days_of_the_month`, but the compiler will still treat it as an `int`. However, if you use this type a lot and later want to change it, using the `typedef` provides one location to make the change rather than needing to go through your whole program: you can think of a `typedef` as a constant but for data types.

In addition to aliasing existing data types, you can also create new types that combine several variables into a single record or struct. We gather investment information into a single struct in `invest.c`:

```c
typedef struct {
    double inv0;     // initial investment
    double growth;  // growth rate, where 1.0 = zero growth
    int years;      // number of years to track
    double inarray[MAX_YEARS+1]; // investment values
} Investment;     // the new data type is called Investment
```
Notice how the \texttt{struct} (...) replaces the data type \texttt{int} in our previous \texttt{typedef} example. This syntax creates a new data type \texttt{Investment} with a record structure, with \texttt{fields} named \texttt{inv0} and \texttt{growth} of type \texttt{double}, \texttt{years} of type \texttt{int}, and \texttt{invarray}, an array of \texttt{doubles}.\footnote{The \texttt{typedef} is actually aliasing an anonymous struct with the name \texttt{Investment}. You can omit the \texttt{typedef}, but then you create a type that must be referred to as \texttt{struct Investment} rather than \texttt{Investment}. The \texttt{typedef} provides a more convenient syntax when you use the type.} (Arrays are discussed in Section A.4.9.) With this new type definition, we can define a variable named \texttt{inv} of type \texttt{Investment}:

\begin{verbatim}
Investment inv;
\end{verbatim}

This definition allocates sufficient memory to hold the two \texttt{doubles}, the \texttt{int}, and the array of \texttt{doubles}. We can access the contents of the \texttt{struct} using the “.” operator:

\begin{verbatim}
int yrs;
yrs = inv.years;
inv.growth = 1.1;
\end{verbatim}

An example of this kind of usage is seen in \texttt{main}.

Referring to the discussion of pointers in Sections A.3.2 and A.4.8, if we are working with a pointer \texttt{invp} that points to \texttt{inv}, we can use the “->” operator to access the contents of the record \texttt{inv}:

\begin{verbatim}
Investment inv; // allocate memory for inv, an investment record
Investment *invp; // invp will point to something of type Investment
int yrs;
invp = &inv; // invp points to inv
inv.years = 5; // setting one of the fields of inv
yrs = invp->years; // inv.years, (*invp).years, and invp->years are all identical
invp->growth = 1.1;
\end{verbatim}

Examples of this usage are seen in \texttt{calculateGrowth()} and \texttt{getUserInput()}. Using the operator \texttt{a->b} is equivalent to doing \texttt{(*a).b}, dereferencing the \texttt{struct} pointer and accessing a specific field.

Another data type you can create is called an \texttt{enum}, short for “enumeration.” Although \texttt{invest.c} does not use an enumeration, they can be useful for describing a type that can take one of a limited set of values. For example, if you wanted a function to use a cardinal direction you could define an enumeration as follows:

\begin{verbatim}
typedef enum {NORTH, SOUTH, EAST, WEST} Direction;
\end{verbatim}

For more information
visit www.nu32.org
Each item in the `enum` gets assigned a constant numerical value. You can explicitly state this value, or use the default compiler-provided values, which start at zero and increment by one for each element. For example, the declaration above is equivalent to

```c
typedef enum {NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3} Direction;
```

You can use an `enum` as you would any other data type.

### A.4.5 Defining Variables

#### Variable names

Variable names can consist of uppercase and lowercase letters, numbers, and underscore characters ‘_’. You should generally use a letter as the first character; `var`, `Var2`, and `Global_Var` are all valid names, but `2var` is not. C is case sensitive, so the variable names `var` and `VAR` are different. A variable name cannot conflict with a reserved keyword in C, like `int` or `for`. Names should be succinct but descriptive. The variable names `i`, `j`, and `k` are often used for integer counters in `for` loops, and pointers often begin with `ptr_`, such as `ptr_var`, or end with `p`, such as `varp`, to remind you that they are pointers. Regardless of how you choose to name your variables, adopting a consistent naming convention throughout a program aids readability.

#### Scope

The **scope** of a variable refers to where it can be used in the program. A variable may be **global**, i.e., usable by any function, or **local** to a specific function or piece of a function. A global variable is one that is defined in the `GLOBAL VARIABLES` section, outside of and before any function that uses it. Such variables can be referred to or altered in any function.\(^{20}\) Because of this special status, global variables are often Capitalized. Global variable usage should be minimized for program modularity and readability.

A local variable is one that is defined in a function. Such a variable is only usable inside that function, after the definition.\(^{21}\) If you choose a local variable name `var` that is also the name of a global variable, inside that function `var` will refer to the local variable, and the global variable will not be available. It is not good practice to choose local variable names to be the

---

\(^{20}\) You could also define a variable outside of any function definition but *after* some of the function definitions. This quasi-global variable would be available to all functions defined after the variable is defined, but not those before. This practice is discouraged, as it makes the code harder to read.

\(^{21}\) Since we recommend that each function be brief, you can define all local variables in that function at the beginning of the function, so we can see in one place what local variables the function uses. Some programmers prefer instead to define variables just before their first use, to minimize their scope. Older C specifications required that all local variables be defined at the beginning of a code block enclosed by braces `{ }`.

For more information
visit [www.nu32.org](http://www.nu32.org)
same as global variable names, as it makes the program confusing to understand and is often the source of bugs.

The parameters to a function are local to that function’s definition, as in sendOutput at the end of invest.c:

```c
void sendOutput(double *arr, int yrs) { // ...
```

The variables `arr` and `yrs` are local to `sendOutput`.

Otherwise, local variables are defined at the beginning of the function code block by syntax similar to that shown in the function `main`.

```c
int main(void) {
    Investment inv; // Investment is a variable type we defined
    // ... rest of the main function ...
```

Since this definition appears within the function, `inv` is local to `main`. Had this definition appeared before any function definition, `inv` would be a global variable.

A global variable can be declared as `static`:

```c
static int i;
```

The `static` specifier means that the global variable can only be used from within the given `.c` file; other files cannot use the variable. If you must use a global variable, you should declare it `static` if possible. Preventing other `.c` files in a multi-file C program from accessing the variable helps increase modularity and reduce bugs.

Variables can also have `qualifiers` attached to their types. There are two main qualifiers in C, `const` and `volatile`. The `const` qualifier prevents the “variable” from being modified. The definition

```c
const int i = 3;
```

sets `i` to 3, and `i` can never be changed after that. The `volatile` qualifier is used extensively in embedded programming, and indicates that the variable may change outside of the normal flow of the program (i.e., due to interrupts, Chapter 6) that the compiler cannot know about in advance. Therefore, the compiler should not assume anything about the current value of a `volatile` variable. Here is an example usage:

```c
volatile int i;
```
We discuss `volatile` more fully in Chapter 6.

Qualifiers can also be applied to pointers. The syntax, however, can be a bit tricky. The first two definitions below make `cp` a `const` pointer, meaning that the content pointed to by `cp` (i.e., `*cp`) is constant: the value of `cp`, however, may change. The third definition makes the pointer `cp` itself constant, but the contents pointed to by `cp` may change. The fourth line makes both the pointer and the data that the pointer references constant.

```c
const int * cp; // pointer to const data
int const * cp; // pointer to const data
int * const cp; // pointer to data, value of cp is const
const int * const cp; // pointer and data are const
```

Definition and initialization

When a variable is defined, memory for the variable is allocated. In general, you cannot assume anything about the contents of the variable until you have initialized it. For example, if you want to define a `float x` with value 0.0, the command

```c
float x;
```

is insufficient. The memory allocated may have random 0’s and 1’s already in it, and the allocation of memory does not generally change the current contents of the memory. Instead, you can use

```c
float x = 0.0;
```

to initialize the value of `x` when you define it. Equivalently, you could use

```c
float x;
x = 0.0;
```

but, when possible, it is better to initialize a variable when you define it, so you do not accidentally use an uninitialized value.

Static local variables

Each time a function is called, its local variables are allocated space in memory. When the function completes, its local variables are discarded, freeing memory. If you want to keep the results of some calculation by the function after the function completes, you could return the results from the function or store them in a global variable. Sometimes, a better alternative is to use the `static` modifier in the local variable definition, as in the following program:

```c
#include <stdio.h>
```

For more information
visit www.nu32.org
void myFunc(void) {
    static char ch='d'; // this local variable is static, allocated and initialized
    // only once during the entire program
    printf("ch value is %d, ASCII character %c\n",ch,ch);
    ch = ch+1;
}

int main(void) {
    myFunc();
    myFunc();
    myFunc();
    return 0;
}

The static modifier in the definition of ch in myFunc means that ch is only allocated, and initialized to 'd', the first time myFunc is called during the execution of the program. This allocation persists after the function returns, and the value of ch is remembered. The output of this program is

ch value is 100, ASCII character d
ch value is 101, ASCII character e
ch value is 102, ASCII character f

Numerical values

Just as you can assign an integer a base-10 value using commands like ch=100, you can assign a number written in hexadecimal notation by putting “0x” at the beginning of the digit sequence, e.g.,

unsigned char ch = 0x64; // ch now has the base-10 value 100

This form may be convenient when you want to directly control bit values. This is often useful in microcontroller applications. Some C compilers, including the PIC32 C compiler, allow specifying bits directly using the following syntax:

unsigned char ch = 0b1100100; // ch now has the base-10 value 100

A.4.6 Defining and Calling Functions

A function definition consists of the function’s return type, function name, argument list, and body (a block of code). Allowable function names follow the same rules as variable names. The function name should make the purpose of the function clear; for example, getUserInput (which gets input from the user) in invest.c.

If the function does not return a value, it has return type void, as with calculateGrowth. If it does return a value, such as getUserInput which returns an int, the function should end with the command
return val;

where val is a variable with the same type as the function’s return type. The main function returns an int and should return 0 upon successful completion.

The function definition

```c
void sendOutput(double *arr, int yrs) { // ...
```

indicates that sendOutput returns nothing and takes two arguments, a pointer to type double and an int. When the function is called with the statement

```c
sendOutput(inv.invarray, inv.years);
```

the invarray and years fields of the inv structure in main are copied to sendOutput, which now has its own local copies of these variables, stored in arr and yrs. The difference between the two is that yrs is just data, while arr is a pointer, holding the address of the first element of invarray, i.e., &inv.invarray[0]). (Arrays will be discussed in more detail in Section A.4.9.) Since sendOutput now has the memory address of the beginning of this array, it can directly access, and potentially change, the original array seen by main. On the other hand, sendOutput cannot, by itself change the value of inv.years in main, since it only has a copy of that value, not the actual memory address of main’s inv.years. sendOutput takes advantage of its direct access to the inv.invarray to print out all the values stored there, eliminating the need to copy all the values of the array from main to sendOutput. To prevent the function from changing the contents of arr we could have (and probably should have) declared the parameter const, as in double const * arr.

The function calculateGrowth, which is called with a pointer to main’s inv data structure, takes advantage of its direct access to the invarray field to change the values stored there.

When a function is passed a pointer argument, it is called a pass by reference; the argument is a reference (address, or pointer) to data. When a function is passed non-pointer data, it is called a pass by value; data is copied but not an address.

If a function takes no arguments and returns no value, we can define it as void myFunc(void). The function is called using

```c
myFunc();
```

A.4.7 Math

Standard binary math operators (operators on two operands) include +, -, *, and /. These operators take two operands and return a result, as in

For more information
visit www.nu32.org
ratio = a/b;

If the operands are the same type, then the CPU carries out a division (or add, subtract, multiply) specific for that type and produces a result of the same type. In particular, if the operands are integers, the result will be an integer, even for division (fractions are rounded toward zero). If one operand is an integer type and the other is a floating point type, the integer type will generally be coerced to a floating point to allow the operation (see the typecast.c program description of Section A.4).

The modulo operator % takes two integers and returns the remainder of their division, i.e.,

```
int i;
i = 16 % 7; // i is now equal to 2
```

C also provides +=, -=, *=, /=, %= to simplify some expressions, as shown below:

```
x = x * 2; // these two lines
x *= 2; // are equivalent

y = y + 7; // these two lines
y += 7; // are equivalent
```

Since adding one to an integer or subtracting one from an integer are common operations in loops, these have a further simplification. For an integer \( i \), we can write

```
++i; // adds 1 to i, equivalent to i = i+1;
--i; // equivalent to i = i-1;
```

In fact we also have the syntax \( i++ \) and \( i-- \). If the ++ or -- come in front of the variable, the variable is modified before it is used in the rest of the expression. If they come after, the variable is modified after the expression has been evaluated. So

```
int i = 5, j;
j = (++i)*2; // after this line, i is 6 and j is 12
```

but

```
int i = 5, j;
j = (i++)*2; // after this line, i is 6 and j is 10
```

But your code would be much more readable if you just wrote \( i++ \) before or after the \( j = i*2 \) line.
If your program includes the C math library with the preprocessor command 
#include <math.h>, you have access to a much larger set of mathematical operations, some of which are 
listed here:

```
int abs (int x); // integer absolute value
double fabs (double x); // floating point absolute value
double cos (double x); // all trig functions work in radians, not degrees
double sin (double x);
double tan (double x);
double acos (double x); // inverse cosine
double asin (double x);
double atan (double x);
double atan2 (double y, double x); // two-argument arctangent
double exp (double x); // base e exponential
double log (double x); // natural logarithm
double log2 (double x); // base 2 logarithm
double log10 (double x); // base 10 logarithm
double pow (double x, double y); // raise x to the power of y
double sqrt (double x); // square root of x
```

These functions also have versions for floats. The names of those functions are identical, except with an 'f' appended to the end, e.g., cosf.

When compiling programs using math.h, you may need to include the linker flag -lm, e.g.,

```
gcc myprog.c -o myprog -lm
```

to tell the linker to link with the math library.

**A.4.8 Pointers**

It’s a good idea to review the introduction to pointers in Section A.3.2 and the discussion of call by reference in Section A.4.6. In summary, the operator & references a variable, returning a pointer to (the address of) that variable, and the operator * dereferences a pointer, returning the contents of the address.

These statements define a variable `x` of type `float` and a pointer `ptr` to a variable of type `float`:

```
float x;
float *ptr;
```

At this point, the assignment

```
*ptr = 10.3;
```
would result in undefined behavior, because the pointer `ptr` does not currently point to anything. The following code would be valid:

```c
ptr = &x;  // assign ptr to the address of x; x is the "pointee" of ptr
*ptr = 10.3;  // set the contents at address ptr to 10.3; now x is equal to 10.3
*(&x) = 4 + *ptr;  // the * and & on the left cancel each other; x is set to 14.3
```

Since `ptr` is an address, it is an integer (technically the type is “pointer to type float”), and we can add and subtract integers from it. For example, say that `ptr` contains the value `n`, and then we execute the statement

```c
ptr = ptr + 1;  // equivalent to ptr++;
```

If we now examined `ptr`, we would find that it has the value `n + 4`. Why? Because the compiler knows that `ptr` points to the type `float`, so when we add 1 to `ptr`, the assumption is that we want to increment by one `float` in memory, not one byte. Since a `float` occupies four bytes, the address `ptr` must increase by 4 to point to the next `float`. The ability to increment a pointer in this way can be useful when dealing with arrays, next.

### A.4.9 Arrays and Strings

One-dimensional arrays

An array of five `floats` can be defined by

```c
float arr[5];
```

We could also initialize the array at the time we define it:

```c
float arr[5] = {0.0, 10.0, 20.0, 30.0, 40.0};
```

Each of these definitions allocates five `floats` in memory, accessed by `arr[0]` (initialized to 0.0 above) through `arr[4]` (initialized to 40.0). The elements are stored consecutively, as per Figure A.2. The assignment

```c
arr[5] = 3.2;
```

![Figure A.2](https://via.placeholder.com/400)

A float array with five elements, as stored in memory. The dashed lines separate bytes and the solid lines separate floats. Each float has four bytes.
is a mistake, since only \texttt{arr[0..4]} have been allocated. This statement will compile successfully because compilers do not check for indexing arrays out of bounds. The best result at this point would be for your program to crash, to alert you to the fact that you are overwriting memory that may be allocated for another purpose. More insidiously, the program could seem to run just fine, but with difficult-to-debug erratic behavior. \footnote{The mistake could potentially be exploited as a security flaw.} Bottom line: never access arrays out of bounds!

In the expression \texttt{arr[i]}, \texttt{i} is an integer called the \textit{index}, and \texttt{arr[i]} is of type \texttt{float}. The variable \texttt{arr} by itself points to the first element of the array, and is equivalent to \texttt{&(arr[0])}. The address \texttt{&(arr[i])} is at \texttt{arr} plus \texttt{i*4} bytes, since the elements of the array are stored consecutively, and a \texttt{float} uses four bytes. Both \texttt{arr[i]} and \texttt{*(arr+i)} are correct syntax to access the \texttt{i}th element of the array. Since the compiler knows that \texttt{arr} is a pointer to the four-byte type \texttt{float}, the address represented by \texttt{(arr+i)} is \texttt{i*4} bytes higher than the address \texttt{arr}.

Consider the following code:

\begin{verbatim}
float arr[5] = {0.0, 10.0, 20.0, 30.0, 40.0};
float *ptr;
ptr = arr + 3;
// arr[0] contains 0.0 and ptr[0] = arr[3] = 30.0
// arr[0] is equivalent to *arr; ptr[0] is equivalent to *ptr and *(arr+3);
// ptr is equivalent to &(arr[3])
\end{verbatim}

If we would like to pass the array \texttt{arr} to a function that initializes each element of the array, we could call

\begin{verbatim}
arrayInit(arr,5);
\end{verbatim}

or

\begin{verbatim}
arrayInit(&arr[0],5);
\end{verbatim}

The function definition for \texttt{arrayInit} might look like

\begin{verbatim}
void arrayInit(float *vals, int length) {
    int i;
    for (i=0; i<length; i++) vals[i] = i*10.0;
    // equivalently, we could substitute the line below for the line above
    // for (i=0; i<length; i++) {*vals = i*10.0; vals++;}
}
\end{verbatim}
The pointer \texttt{vals} in \texttt{arrayInit} is set to point to the same location as \texttt{arr} in the calling function. Therefore \texttt{vals[i]} refers to the same memory contents that \texttt{arr[i]} does.

Note that \texttt{arr} does not carry any information on the length of the array, so we must send the length separately to \texttt{arrayInit}.

Strings

A string is an array of \texttt{chars}. The definition

\begin{verbatim}
char s[100];
\end{verbatim}

allocates memory for 100 \texttt{chars}, \texttt{s[0]} to \texttt{s[99]}. We could initialize the array with

\begin{verbatim}
char s[100] = "cat": // note the double quotes
\end{verbatim}

This places a 'c' (integer value 99) in \texttt{s[0]}, an 'a' (integer value 97) in \texttt{s[1]}, a 't' (integer value 116) in \texttt{s[2]}, and a value of 0 in \texttt{s[3]}, corresponding to the NULL ('\0') character and indicating the end of the string. (You could also do this, less elegantly, by initializing just those four elements using braces as we did with the \texttt{float} array above.)

You notice that we allocated more memory than was needed to hold "cat." Perhaps we will append something to the string in future, so we might want to allocate that extra space just in case. But if not, we could have initialized the string using

\begin{verbatim}
char s[] = "cat";
\end{verbatim}

and the compiler would only assign the minimum memory needed (four bytes in this case, three for each character and one for the NULL character).

The function \texttt{sendOutput} in \texttt{invest.c} shows an example of constructing a string using \texttt{sprintf}, a function provided by \texttt{stdio.h}. Other functions for manipulating strings are provided in \texttt{string.h}. Both of these headers are described briefly in Section A.4.14.

Multi-dimensional arrays

The definition

\begin{verbatim}
int mat[2][3];
\end{verbatim}

allocates memory for 6 \texttt{ints}, \texttt{mat[0][0]} to \texttt{mat[1][2]}, which can be thought of as a two-dimensional array, or matrix. These occupy a contiguous region of memory, with \texttt{mat[0][0]} at the lowest memory location, followed by \texttt{mat[0][1]}, \texttt{mat[0][2]}, \texttt{mat[1][0]}, \texttt{mat[1][1]}, and \texttt{mat[1][2]}. This matrix can be initialized using nested braces,
int mat[2][3] = {{0, 1, 2}, {3, 4, 5}};

Higher-dimensional arrays can be created by simply adding more indexes. In memory, a “row” of the rightmost index is completed before incrementing the next index to the left.

Static vs. dynamic memory allocation

A command of the form float arr[5] is called *static memory allocation*, meaning that the size of the array is known at compile time. Another option is *dynamic memory allocation*, where the size of the array can be chosen at run time. With the C standard library header `stdlib.h` included using the preprocessor command `#include <stdlib.h>`, the syntax

```c
float *arr; // arr is a pointer to float, but no memory has been allocated for the array
int i=5;
arr = (float *) malloc(i * sizeof(float)); // allocate the memory
```

allocates arr[0..4], and

```c
free(arr);
```

releases the memory when it is no longer needed. It is crucial to remember to *free* memory allocated with `malloc` when you are finished with it, so you do not run out of memory if you repeatedly allocate memory. If `malloc` cannot allocate the requested memory, perhaps because the computer is out of memory, it returns a NULL pointer (i.e., `arr` will have value 0). You must *always* check that the result of `malloc` is valid before continuing with your program.

A.4.10 Relational Operators and TRUE/FALSE Expressions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>equal</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal</td>
</tr>
<tr>
<td><code>&gt;</code>, <code>&gt;=</code></td>
<td>greater than, greater than or equal to</td>
</tr>
<tr>
<td><code>&lt;</code>, <code>&lt;=</code></td>
<td>less than, less than or equal to</td>
</tr>
</tbody>
</table>

Relational operators operate on two values and evaluate to 0 or 1. A 0 indicates that the expression is FALSE and a 1 indicates that the expression is TRUE. For example, the expression `(3>=2)` is TRUE, so it evaluates to 1, while `(3<2)` evaluates to 0, or FALSE.

---

23 Dynamic memory is allocated from the *heap*, a portion of memory set aside for dynamic allocation (and therefore is not available for statically allocated variables and program code). You may have to adjust linker options setting the size of the heap. See Chapter 5.3.

24 `malloc` tracks the size of the block associated with the address `arr`, so you do not need to tell `free` how much memory to release.

For more information visit [www.nu32.org](http://www.nu32.org)
The most common mistake with relational operators is using = to test for equality instead of ==. For example, using the if conditional syntax (Section A.4.12), the test

```c
int i = 2;
if (i = 3) {  // error: this is an assignment, not a test!!
    printf("Test is true.");
}
```

always evaluates to TRUE, because the expression (i=3) assigns the value of 3 to i, and the expression evaluates to 3. Any nonzero value is treated as logical TRUE. If the condition is written (i==3), it will operate as intended, evaluating to 0 (FALSE).

Be aware of potential pitfalls in checking equality of floating point numbers. Consider the following program:

```c
#include <stdio.h>
#define VALUE 3.1
int main(void) {
    float x = VALUE;
    double y = VALUE;
    if (x == VALUE) {
        printf("x is equal to %f.\n",VALUE);
    } else {
        printf("x is not equal to %f!\n",VALUE);
    }
    if (y == VALUE) {
        printf("y is equal to %f.\n",VALUE);
    } else {
        printf("y is not equal to %f!\n",VALUE);
    }
    return 0;
}
```

You might be surprised to see that your program says that x is not equal while y is! In fact, neither x nor y are exactly 3.1 due to roundoff error in the floating point representation. However, by default, the constant 3.1 is treated as a double, so the double y carries the identical (wrong) value. If you want a constant to be treated explicitly as a float, you can write it as 3.1F, and if you want it to be treated as a long double, you can write it as 3.1L.

### A.4.11 Logical and Bitwise Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift bits to the right (shifting in 0’s from the left)</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift bits to the left (shifting in 0’s from the right)</td>
</tr>
</tbody>
</table>

For more information
visit www.nu32.org
Bitwise operators act directly on the bits of the operand(s), as in the following example:

```
unsigned char a=0xC, b=0x6, c; // in binary, a is 0b00001100 and b is 0b00000110
  c = ~a; // NOT; c is 0xF3 or 0b11110011
  c = a & b; // AND; c is 0x04 or 0b00000100
  c = a | b; // OR; c is 0x0E or 0b00001110
  c = a ^ b; // XOR; c is 0x0A or 0b00001010
  c = a >> 3; // SHIFT RT 3; c is 0x01 or 0b00000001, one 1 is shifted off the right end
  c = a << 3; // SHIFT LT 3; c is 0x60 or 0b01100000, 1's shifted to more significant digits
```

Much like the math operators, we also have the assignment expressions &=, |=, ^=, >>=, and <<=, so a &= b is equivalent to a = a&b.

**A.4.12 Conditional Statements**

**if-else**

The basic if-else construct takes this form:

```
if (<expression>) {
  // execute this code block if <expression> is TRUE, then exit
}
else {
  // execute this code block if <expression> is FALSE
}
```

If the code block is a single statement, the braces are not necessary; however, good practice dictates that you should always use braces, in case you want to add additional statements later. The else and the block after it can be eliminated if no action needs to be taken when <expression> is FALSE.

if-else statements can be made into arbitrarily long chains:

```
if (<expression1>) {
  // execute this code block if <expression1> is TRUE, then exit this if-else chain
} else if (<expression2>) {
  // execute this code block if <expression2> is TRUE, then exit this if-else chain
} else {
  // execute this code block if both expressions above are FALSE
}
```

An example if statement is in getUserInput.
If you would like to check if the value of a single expression is one of several possibilities, a
switch may be simpler, clearer, and faster than a chain of if-else statements. Here is an example:

```c
char ch;
// ... omitting code that sets the value of ch ...
switch (ch) {
    case 'a': // execute these statements if ch has value 'a'
        <statement>;
        <statement>;
        break; // exit the switch statement
    case 'b':
        // ... some statements
        break;
    case 'c':
        // ... some statements
        break;
    default: // execute this code if none of the previous cases applied
        // ... some statements
}
```

Notice the break; statement after each case. These statements are required to prevent the code from “falling through” to the next case, which is usually undesirable.

### A.4.13 Loops

#### for loop

A for loop has the following syntax:

```c
for (<initialization>; <test>; <update>) {
    // code block
}
```

If the code block consists of only one statement, the surrounding braces can be eliminated, but it is a good idea to use them anyway.

The sequence is as follows: at the beginning of the loop, the <initialization> statement is executed. Then the <test> is evaluated. If it is TRUE, the code block is executed, the <update> is performed, and we return to the <test>. If it is FALSE, the for loop exits.

The following for loop is in calculateGrowth:

```c
for (i = 1; i <= invp->years; i = i + 1) {
    invp->invarray[i] = invp->growth*invp->invarray[i-1];
}
```
The `<initialization>` step sets $i = 1$. The `<test>` is TRUE if $i$ is less than or equal to the number of years we will calculate growth in the investment. If it is TRUE, the value of the investment in year $i$ is calculated from the value in year $i-1$ and the growth rate. The `<update>` adds 1 to $i$. In this example, the code block is executed for $i$ values of 1 to `invp->years`.

It is possible to perform more than one statement in the `<initialization>` and `<update>` steps by separating the statements by commas. For example, we could write

```c
for (i = 1, j = 10; i <= 10; i++, j--) { /* code */);
```

if we want $i$ to count up and $j$ to count down.

**while loop**

A `while` loop has the following syntax:

```c
while (<test>) {
    // code block
}
```

First, the `<test>` is evaluated, and if it is FALSE, the while loop exits. If the test is TRUE, the code block is executed and we return to the `<test>`.

In `main` of `invest.c`, the `while` loop executes until the function `getUserInput` returns 0, i.e., FALSE. `getUserInput` collects the user’s input and returns an `int` that is 0 if the user’s input is invalid and 1 if it is valid.

**do-while loop**

This is similar to a `while` loop, except the `<test>` is executed at the end of the code block, guaranteeing that the loop is executed at least once.

```c
do {
    // code block
} while (<test>);
```

**break and continue**

If anywhere in the loop’s code block the command `break` is encountered, the program will exit the loop. If the command `continue` is encountered, the rest of the commands in the code block will be skipped, and control will return to the `<update>` in a `for` loop or the `<test>` in a `while` or `do-while` loop. Examples:

```c
while (<test1>) {
    if (<test2>) {
        break; // jump out of the while loop
    }
    // ...
}
```
while (<test1>) {
if (<test2>) {
    continue; // skip the rest of the loop and go back to <test1>
} 
    x = x + 3;
}

Use break and continue judiciously; they can make your code difficult to read. If you find yourself relying on numerous break and continue statements in a single loop, you may want to rethink your approach.

A.4.14 The C Standard Library

C comes with a standard library, aspects of which you can use by including the appropriate .h header file. The header file provides data types, function prototypes and macros required for part of the library. We have already seen examples of standard header files such as stdio.h, which contains input/output functions; math.h in Section A.4.7; and stdlib.h in Section A.4.9. Third-party libraries can also be included by including their header files and linking with them.

It is well beyond our scope to provide details on the C standard library. Here we highlight a few particularly useful functions in stdio.h, string.h, and stdlib.h.

Input and output: stdio.h

```c
int printf(const char *Format, ...);
```

The function printf is used to print to the “standard output,” which, for a PC, is typically the screen. It takes a formatting string Format and a variable number of extra arguments, determined by the formatting string, as indicated by the ... notation. The keyword const means that printf cannot change the string Format.

An example comes from our program printout.c:

```c
int i;
float f;
double d;
char c;
```

Reminder: if you include <math.h>, you should also compile your program with the -lm flag, so the math library is linked during the linking stage. The math library is logically part of the C standard library, it just happens to be in a different file.
i = 32;
f = 4.278;
d = 4.278;
c = 'k'; // or, by ASCII table, c = 107:

printf("Formatted output:\n");
printf(" i = %4d c = '%c'\n",i,c);
printf(" f = %19.17f\n",f);
printf(" d = %19.17f\n",d);

which produces the output

Formatted output:
i = 32 c = 'k'
f = 4.27799987792968750
d = 4.2779999999999958

The formatting strings consist of plain text, the special character \n that prints a newline, and directives of the form %4d and %19.17f. Each directive indicates that printf will be looking for a corresponding variable in the argument list to insert into the output. A non-exhaustive list of directives is given here:

%4d  Print an int. Corresponding argument should be an int.
%u  Print an unsigned int. Corresponding argument should be an integer data type.
%ld  Print a long int.
%f  Print a double or a float. Corresponding argument should be a float or a double.
%c  Print a character according to the ASCII table. Argument should be char.
%s  Print a string. Argument should be a pointer to a char (first element of a string), terminated with a NULL character ('\0').
%x  Print an unsigned int as a hexadecimal number.

The directive %d can be written instead as %4d, for example, meaning that four spaces are allocated to write the integer, which will be right-justified in that space with unused spaces blank. The directive %f can be written instead as %6.3f, indicating that six spaces are reserved to write out the variable, with one of those spaces being the decimal point and three of the spaces after the decimal point.

int sprintf(char *str, const char *Format, ...):

Instead of printing to the screen, sprintf prints to the string str. An example of this is in sendOutput. The string str must have enough memory allocated to fit the results.

int scanf(const char *Format, ...):

The function scanf is a formatted read from the “standard input,” which is typically the keyboard. Arguments to scanf consist of a formatting string and pointers to variables where

For more information
visit www.nu32.org
the input should be stored. Typically the formatting string consists of directives like \%d, \%f, etc., separated by whitespace. The directives are similar to those for \printf, except they do not accept spacing modifiers (like the 5 in \%5d).

One notable difference between formatting strings is that, unlike \printf, \scanf does distinguish between floats and doubles. To read a double with \scanf use \%lf rather than \%f.

For each directive, \scanf expects a pointer to a variable of that type in the argument list. A very common mistake is the following:

```c
int i;
scanf("\%d",i);  // WRONG! We need a pointer to the variable.
scanf("\%d",\&i);  // RIGHT.
```

The pointer allows \scanf to put the input into the right place in memory.

getUserInput uses the statement

```c
scanf("\%lf \%lf \%d", (invp->inv0), (invp->growth), (invp->years));
```

to read in two doubles and an int and place them into the appropriate spots in the investment data structure. \scanf ignores the whitespace (tabs, newlines, spaces, etc.) between the inputs.

```
int sscanf(char *str, const char *Format, ...);
```

Instead of scanning from the keyboard, \scanf scans the string pointed to by \str.

```
FILE* fopen(const char *Path, const char *Mode);
int fclose(FILE *Stream);
int fscanf(FILE *Stream, const char *Format, ...);
int fprintf(FILE *Stream, const char *Format, ...);
```

These commands are for reading from and writing to files. Say you have a file named \inputfile, sitting in the same directory as the program, with information your program needs. The following code would read from it and then write to the file \outputfile.

```c
int i;
double x;
FILE *input, *output;
input = fopen("inputfile","r");  // "r" means you will read from this file
output = fopen("outputfile","w");  // "w" means you will write to this file
fscanf(input,"\%d \%lf\n",\&i,\&x);
fprintf(output,"I read in an integer %d and a double %lf.\n",i,x);
fclose(input);  // these streams should be closed ...
fclose(output);  // ... at the end of the program
```

Normally, you would check the return value of \fopen. If it returns NULL, than it failed to open the file.

For more information visit www.nu32.org
int fputc(int character, FILE *stream);
int fputs(const char *str, FILE *stream);
int fgetc(FILE *stream);
char* fgets(char *str, int num, FILE *stream);
int puts(const char *str);

These commands write (put) a character or string to a file, get a character or string from a file, or write a string to the screen. The variable stdin is a FILE * that corresponds to keyboard input, and stdout is a FILE * that corresponds to screen output.

String manipulation: string.h

char* strcpy(char *destination, const char *source);

Given two strings, char destination[100], source[100], we cannot simply copy one to the other using the assignment destination = source. Instead we use strcpy(destination, source), which copies the string source (until reaching the string terminator character, integer value 0) to destination. The string destination must have enough memory allocated to hold the source string.

char* strcat(char *destination, const char *source);

Appends the string in source to the end of the string destination, which must be large enough to hold the concatenated string.

int strcmp(const char *s1, const char *s2);

Returns 0 if the two strings are identical, a positive integer if the first unequal character in s1 is greater than s2, and a negative integer if the first unequal character in s1 is less than s2.

size_t strlen(const char *s);

The type size_t is an unsigned integer type. strlen returns the length of the string s, where the end of the string is indicated by the NULL character (’\0’).

void* memset(void *s, int c, size_t len);

memset writes len bytes of the value c (converted to an unsigned char) starting at the beginning of the string s. So

For more information visit www.nu32.org
char s[10];
memset(s,'c',5);

would fill the first five characters of the string s with the character ‘c’ (or integer value 99). This can be a convenient way to initialize a string.

**General purpose functions in stdlib.h**

```c
void* malloc(size_t objectSize)
```

`malloc` is used for dynamic memory allocation. An example use is in Section A.4.9.

```c
void free(void *objptr)
```

`free` is used to release memory allocated by `malloc`. An example use is in Section A.4.9.

```c
int rand()
```

It is sometimes useful to generate random numbers, particularly for games. The code

```c
int i;
i = rand();
```

places in i a pseudo-random number between 0 and `RAND_MAX`, a constant which is defined in `stdlib.h` (2,147,483,647 in our gcc installation). To convert this to an integer between 1 and 10, you could follow with

```c
i = 1 + (int)((10.0*i)/(RAND_MAX+1.0));
```

One drawback of the code above is that calling `rand` multiple times will lead to the same sequence of random numbers every time the program runs. The usual solution is to “seed” the random number algorithm with a different number each time, and this different number is often taken from a system clock. The `srand` function is used to seed `rand`, as in the example below:

```c
#include <stdio.h>   // allows use of printf()
#include <stdlib.h>  // allows use of rand() and srand()
#include <time.h>    // allows use of time()

int main(void) {
    int i:
```

For more information
visit www.nu32.org
srand(time(NULL)); // seed the random number generator with the current time
for (i=0; i<10; i++) printf("Random number: %d\n",rand());
return 0;
}

If we take out the line with srand, this program produces the same ten “random” numbers every time we run it. Note that this program includes the time.h header to allow the use of the time function.

```c
void exit(int status)
```

When exit is invoked, the program exits with the exit code status. stdlib.h defines EXIT_SUCCESS with value 0 and EXIT_FAILURE with value −1, so that a typical call to exit might look like
```
exit(EXIT_SUCCESS);
```

A.4.15 Multiple File Programs and Libraries

So far our programs have used the C Standard Library, which provides several functions, including printf and scanf. Accessing these functions is possible because:

1. The preprocessor command `#include <stdio.h>` inserts the header file stdio.h into the current compilation unit, providing function prototypes for library functions.
2. The linker links the pre-compiled library object code in the C Standard Library with your program.

Thus, using a library usually requires header files and object code. We could also loosely define a library to consist of a header file and a C file (without a main function) containing source code for the library functions.

The purpose of a library is to collect functions that are likely to be useful in multiple programs so you do not have to rewrite the code for each program. The same principles apply when dividing your project into multiple source files. Think about what functions may be generally useful yet related and put them into their own file. Having code in multiple files not only promotes reuse, but it also isolates your project’s components making them easier to test and debug. Many libraries evolve from a collection of source files that were designed for one particular project, but prove more generally useful.

Let us look at an example.

**A simple example: The rad2volume library**

Pretend you want to write several programs that need to calculate the volume of a sphere given its radius. You could copy and paste the formula into all of your programs. If, however, you
made some mistake and wanted to fix it, you would then need to find where you used the formula in every program and change it. By placing the formula in a function, in its own file, you only need to make one correction to fix everything.

In this example, you decide to write one helper C file, `rad2volume.c`, with a function `double radius2Volume(double r)` that can be used by other C files. For good measure, you decide to make the constant `MY_PI` available also. To test your new `rad2volume` library consisting of `rad2volume.c` and `rad2volume.h`, you create a `main.c` file that uses it. The three files are given below.

```c
// ***** file: rad2volume.h *****
#ifndef RAD2VOLUME_H // "include guard"; don't include twice in one compilation
#define RAD2VOLUME_H // second line of the "include guard"

#define MY_PI 3.1415926 // constant available to files including rad2Volume.h
double radius2Volume(double r); // prototype available to files including rad2Volume.h
#endif // third line, and end, of "include guard"

// ***** file: rad2volume.c *****
#include <math.h> // for the function pow
#include "rad2volume.h" // if the header is in the same directory, use "quotes"

static double cuber(double x) { // this function is not available externally
    return pow(x, 3.0);
}

double radius2Volume(double rad) { // function definition
    return (4.0/3.0)*MY_PI*cuber(rad);
}

// ***** file: main.c *****
#include <stdio.h>
#include "rad2volume.h"

int main(void) {
    double radius = 3.0, volume;
    volume = radius2Volume(radius);
    printf("Pi is approximated as %25.23lf.\n", MY_PI);
    printf("The volume of the sphere is %8.4lf.\n", volume);
    return 0;
}
```

The C file `rad2volume.c` contains two functions, `cuber` and `radius2Volume`. The function `cuber` is only meant for internal, private use by `rad2volume.c`, so there is no prototype in
rad2volume.h and it is also declared static so it is not visible to other source files. The function radius2Volume is meant for public use by other C files, so a prototype for radius2Volume is included in the library header file rad2volume.h. The constant MY_PI is also meant for public use, so it is defined in rad2volume.h. Now radius2Volume and MY_PI are available to any file that includes rad2volume.h. In this case, they are available to main.c and rad2volume.c. Typically, it is good practice for the implementation file to #include its own header; this prevents problems relating to a mismatch between function prototypes and function definitions.

Each of main.c and rad2volume.c is compiled independently to create the object code files main.o and rad2volume.o. The linker combines these files into the final executable. main.c compiles successfully because it has a prototype for radius2Volume from including rad2volume.h, and it expects that, during the linking stage, radius2Volume will be present. If no object code passed to the linker defines radius2Volume then a linker error occurs.

Note the three lines making up the include guard in rad2volume.h. During preprocessing of a C file, if rad2volume.h is included, the macro RAD2VOLUME_H is defined. If the same C file tries to include rad2volume.h again, the include guard will recognize that RAD2VOLUME_H already exists and therefore skip the prototype and constant definition, down to the #endif. Without include guards, if we wrote a .c file including both header1.h and header2.h, for example, not knowing that header2.h already includes header1.h, header1.h would be included twice, possibly causing errors.

The two C files, rad2volume.c and main.c, can be compiled into object code using the commands

```bash
gcc -c rad2volume.c -o rad2volume.o
gcc -c main.c -o main.o
```

where the -c flag indicates that the source code should be compiled and assembled, but not linked. The result is the object files rad2volume.o and main.o. The two object files can be linked into a final executable using

```bash
gcc rad2volume.o main.o -o myprog
```

Instead of typing these three lines, the single command

```bash
gcc rad2volume.c main.c -o myprog
```

will automatically compile and link the files.

Executing myprog, the output is

```
Pi is approximated as 3.14159260000000006840537.
The volume of the sphere is 113.0973.
```
More general multi-file projects

Generalizing from the previous example, a header file declares constants, macros, new data types, and function prototypes that are needed by the files that \#include them. A header file can be included by C source files or other header files. Figure A.3 illustrates a project consisting of one C source file with a \texttt{main} function and two helper C source files without a \texttt{main} function. (Every C program has exactly one .c file with a \texttt{main} function.) Each of the helper C files has its own header file. This project also has another header file, \texttt{general.h}, without an associated C file. This header contains general constant, macro, and data type definitions that are not specific to either helper source file or the \texttt{main} C file. The arrows indicate that the pointed-to file \#includes the pointed-from header file.

Assuming that all the files are in the same directory, the project in Figure A.3 can be built by the following four commands, which create three object files (one for each source file) and link them together into \texttt{myprog}:

\begin{verbatim}
gcc -c main.c -o main.o
gcc -c helper1.c -o helper1.o
gcc -c helper2.c -o helper2.o
gcc main.o helper1.o helper2.o -o myprog
\end{verbatim}

The build is illustrated in Figure A.4. Each C file is compiled independently and requires the constants, macros, data types, and function prototypes needed to successfully compile and assemble into an object file. During compilation of a single C file, the compiler neither has nor needs access to the source code for functions in other C files. If \texttt{main.c} uses a function in \texttt{helper1.c}, for example, it needs only a prototype of the function, provided by \texttt{helper1.h}. The prototype tells the compiler the return type of the function and what arguments it takes, allowing the compiler to check if the code in \texttt{main.c} uses the function properly. Calls to the function from \texttt{main.o} are linked to the actual function in \texttt{helper1.o} at the linker stage.

![Figure A.3](image)

An example project consisting of three C files and three header files. Arrows point from header files to files that include them.

For more information
visit www.nu32.org
Another benefit of splitting your project comes during compilation. If a .c file and the header files it includes do not change, then the .c file need not be compiled every time you build your project; the existing .o file may be used during the linking stage.

According to Figures A.3 and A.4, main.c has the following preprocessor commands:

```c
#include "general.h"
#include "helper1.h"
#include "helper2.h"
```

The preprocessor replaces these commands with copies of the files general.h, helper1.h, and helper2.h. But when it includes helper1.h, it finds that helper1.h tries to include a second copy of general.h (see Figure A.3; helper1.h has a #include "general.h" command). Since general.h has already been copied in, it should not be copied again; otherwise we would have multiple copies of the same function prototypes, constant definitions, etc. The include guards prevent this duplication from happening.

In summary, the general.h, helper1.h, and helper2.h header files contain definitions that are made public to files including them. We might see the following items in the helper1.h header file, for example:

For more information visit www.nu32.org
• an include guard
• other include files
• constants and macros defined with `#define` made public (and which may also be used by `helper1.c`)
• new data types (which may also be used by `helper1.c`)
• function prototypes of those functions in `helper1.c` which are meant to be used by other files

If a variable, function prototype, or constant is private to one C file, you should define it with the `static` keyword in the C file and not include it in the header file.

A header file like `helper1.h` could also have the declaration

```c
extern int Helper1_Global_Var; // no space is allocated by this declaration
```

where `helper1.c` has the global variable definition

```c
int Helper1_Global_Var;       // space is allocated by this definition
```

Then any file including `helper1.h` would have access to the global variable `Helper1_Global_Var` allocated by `helper1.c`. Global variables defined in `helper1.c` with the `static` keyword are private to `helper1.c` and cannot be accessed by other files. If you have to use global variables (which should be avoided generally), declare them `static` whenever possible. Only in rare circumstances should you need to use `extern`.

**Makefiles**

When you are ready to build your executable, you can type the `gcc` commands at the command line, as we have seen previously. A Makefile simplifies the process, particularly for multi-file projects, by specifying the dependencies and commands needed to build the project. A Makefile for our rad2volume example is shown below, where everything after a `#` is a comment.

```makefile
# ***** file: Makefile *****
# Comment: This is the simplest of Makefiles!

# Here is a template:
# [target]: [dependencies]
# [tab] [command to execute]

# The thing to the left of the colon in the first line is what is created.
```

For more information
visit www.nu32.org
With this Makefile in the same directory as your other files, you should be able to type the command `make [target]`, where `[target]` is `myprog`, `main.o`, `rad2volume.o`, or `clean`. If the target depends on other files, make will make sure those are up to date first, and if not, it will call the commands needed to create them. For example, `make myprog` triggers a check of `main.o`, which triggers a check of `main.c` and `rad2volume.h`. If either of those have changed since the last time `main.o` was made, then `main.c` is compiled and assembled to create a new `main.o` before the linking step.

The command `make` with no target specified will make the first target (which is `myprog` in this case).

Ensure that your Makefile is saved without any extensions (e.g., `.txt`) and that the commands are preceded by a tab (not spaces).

There are many more sophisticated uses of Makefiles which you can learn about from other sources.

### A.5 Exercises

1. Install C, create the `HelloWorld.c` program, and compile and run it.
2. Explain what a pointer variable is, and how it is different from a non-pointer variable.
3. Explain the difference between interpreted and compiled code.

---

26 In some C installations `make` is named differently, like `nmake` for Visual Studio or `mingw32-make`. If you can find no version of `make`, you may not have selected the `make` tools installation option when you performed the C installation.
4. Write the following hexadecimal (base-16) numbers in eight-digit binary (base-2) and three-digit decimal (base-10). Also, for each of the eight-digit binary representations, give the value of the most significant bit. (a) 0x1E. (b) 0x32. (c) 0xFE. (d) 0xC4.

5. What is 333\(_{10}\) in binary and 1011110111\(_{2}\) in hexadecimal? What is the maximum value, in decimal, that a 12-bit number can hold?

6. Assume that each byte of memory can be addressed by a 16-bit address, and every 16-bit address has a corresponding byte in memory. How many total bits of memory do you have?

7. (Consult the ASCII table.) Let ch be of type char. (a) The assignment ch = 'k' can be written equivalently using a number on the right side. What is that number? (b) The number for '5'? (c) For '='? (d) For '?'?

8. What is the range of values for an unsigned char, short, and double data type?

9. How many bits are used to store a char, short, int, float, and double?

10. Explain the difference between unsigned and signed integers.

11. (a) For integer math, give the pros and cons of using chars vs. ints. (b) For floating point math, give the pros and cons of using floats vs. doubles. (c) For integer math, give the pros and cons of using chars vs. floats.

12. The following signed short int's, written in decimal, are stored in two bytes of memory using the two’s complement representation. For each, give the four-digit hexadecimal representation of those two bytes. (a) 13. (b) 27. (c) −10. (d) −17.

13. The smallest positive integer that cannot be exactly represented by a four-byte IEEE 754 float is 2\(^{24}\) + 1, or 16,777,217. Explain why.

14. Give the four bytes, in hex, that represent the following decimal values: (a) 20 as an unsigned int. (b) −20 as a two’s complement signed int. (c) 1.5 as an IEEE 754 float. (d) 0 as an IEEE 754 float.

To verify your answers, use the program typereps.c below. This program allows the user to enter four bytes as eight hex characters, then prints the value of those four bytes when they are interpreted as an unsigned int, a two’s complement signed int, an IEEE 754 float, or four consecutive chars. To do this, the program creates a new data type four_types_t consisting of four bytes, or 32 bits. These same 32 bits are interpreted as either an unsigned int, int, float, or four chars depending on whether we reference the bits using the fields u, i, f, or char0-char3 in the union.

```c
#include <stdio.h>

typedef union { // a new data type consisting of four bytes
    unsigned int u; // the 32 bits interpreted as an unsigned int
    int i; // the same 32 bits interpreted as a two’s complement int
    float f; // the same 32 bits interpreted as an IEEE 754 single prec float
    struct {
        char char0:8; // bits 0 - 7 interpreted as char, called char0
        char char1:8; // bits 8 - 15 interpreted as char, called char1
        char char2:8; // bits 16 - 23 interpreted as char, called char2
        char char3:8; // bits 24 - 31 interpreted as char, called char3
    };
} four_types_t;
```

For more information
visit www.nu32.org
char char3:8; // bits 24 - 31 interpreted as char, called char3
);
} four_types_t; // the new type is called four_types_t

int main(void) {
 four_types_t val;

 while (1) { // exit the infinite loop using ctrl-c or similar
   printf("Enter four bytes as eight hex characters 0-f, e.g., abcd0123: ");
   scanf("%x",&val.u);
   printf("The 32 bits in hex: %x
",val.u);
   printf("The 32 bits as an unsigned int, in decimal: %u\n",val.u);
   printf("The 32 bits as a signed int, in decimal: %d\n",val.i);
   printf("The 32 bits as a float: %.20f\n",val.f);
   printf("The 32 bits as 4 chars: %c %c %c %c

", val.char3, val.char2, val.char1, val.char0);
 } return 0;
}

Below is a sample output. Note that only the ASCII values 32-126 have a visible printed representation, so the printouts as chars are meaningless in the first two examples.

Enter four bytes as eight hex characters 0-f, e.g., abcd0123: c0000000
The 32 bits in hex: c0000000
The 32 bits as an unsigned int, in decimal: 3221225472
The 32 bits as a signed int, in decimal: -1073741824
The 32 bits as a float: -2.00000000000000000000
The 32 bits as 4 chars: ?

Enter four bytes as eight hex characters 0-f, e.g., abcd0123: ff800000
The 32 bits in hex: ff800000
The 32 bits as an unsigned int, in decimal: 4286578688
The 32 bits as a signed int, in decimal: -8388608
The 32 bits as a float: -inf
The 32 bits as 4 chars: ? ?

Enter four bytes as eight hex characters 0-f, e.g., abcd0123: 48494a4b
The 32 bits in hex: 48494a4b
The 32 bits as an unsigned int, in decimal: 1212762699
The 32 bits as a signed int, in decimal: 1212762699
The 32 bits as a float: 206121.1718750000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
17. To keep things simple, let us assume we have a microcontroller with only \(2^8 = 256\) bytes of RAM, so each address is given by a single byte. Now consider the following code defining four global variables:

\[
\text{unsigned int } i, j, *kp, *np;
\]

Let us assume that the linker places \(i\) in addresses 0xB0..0xB3, \(j\) in 0xB4..0xB7, \(kp\) in 0xB8, and \(np\) in 0xB9. The code continues as follows:

\[
\begin{align*}
// \text{(a) the initial conditions, all memory contents unknown} \\
kp &= & & \text{(b)} \\
j &= *kp; & & \text{(c)} \\
i &= 0xAE; & & \text{(d)} \\
np &= kp; & & \text{(e)} \\
*np &= 0x12; & & \text{(f)} \\
j &= *kp; & & \text{(g)}
\end{align*}
\]

For each of the comments (a)-(g) above, give the contents (in hexadecimal) at the address ranges 0xB0..0xB3 (the \texttt{unsigned int }\ i), 0xB4..0xB7 (the \texttt{unsigned int }\ j), 0xB8 (the pointer \texttt{kp}), and 0xB9 (the pointer \texttt{np}), at that point in the program, after executing the line containing the comment. The contents of all memory addresses are initially unknown or random, so your answer to (a) is “unknown” for all memory locations. If it matters, assume little-endian representation.

18. Invoking the \texttt{gcc} compiler with a command like \texttt{gcc myprog.c -o myprog} actually initiates four steps. What are the four steps called, and what is the output of each step?

19. What is \texttt{main’s} return type, and what is the meaning of its return value?

20. Give the \texttt{printf} statement that will print out a \texttt{double }\ d \text{ with eight digits to the right of the decimal point and four spaces to the left.}

21. Consider three \texttt{unsigned char}s, \(i, j, \text{ and } k\), with values 60, 80, and 200, respectively. Let \texttt{sum} also be an \texttt{unsigned char}. For each of the following, give the value of \texttt{sum} after performing the addition. (a) \texttt{sum = i+j}; (b) \texttt{sum = i+k}; (c) \texttt{sum = j+k};

22. For the variables defined as

\[
\begin{align*}
\text{int } a=2, b=3, c; \\
\text{float } d=1.0, e=3.5, f;
\end{align*}
\]

give the values of the following expressions. (a) \(f = a/b\); (b) \(f = ((\text{float}) a)/b\); (c) \(f = (\text{float}) (a/b)\); (d) \(c = e/d\); (e) \(c = (\text{int}) (e/d)\); (f) \(f = ((\text{int}) e)/d\);

23. In each snippet of code in (a)-(d), there is an arithmetic error in the final assignment of \texttt{ans}. What is the final value of \texttt{ans} in each case?

\[
\begin{align*}
a. \quad & \text{char } c = 17; \\
& \text{float } ans = (1 / 2) * c; \\
& \text{unsigned int } ans = -4294967295; \\
b. \quad & \text{double } d = \text{pow}(2, 16); \\
& \text{short } ans = (\text{short}) d; \\
c. \quad & \text{double } ans = ((\text{double}) -15 * 7) / (16 / 17) + 2.0;
\end{align*}
\]

For more information
visit www.nu32.org
24. Truncation is not always bad. Say you wanted to store a list of percentages rounded down to the nearest percent, but you were tight on memory and cleverly used an array of chars to store the values. For example, pretend you already had the following snippet of code:

```c
char percent(int a, int b) {
    // assume a <= b
    char c;
    c = ???;
    return c;
}
```

You cannot simply write `c = a / b`. If \( \frac{a}{b} = 0.77426 \) or \( \frac{a}{b} = 0.778 \), then the correct return value is \( c = 77 \). Finish the function definition by writing a one-line statement to replace `c = ???`.

25. Explain why global variables work against modularity.

26. What are the seven sections of a typical C program?

27. You have written a large program with many functions. Your program compiles without errors, but when you run the program with input for which you know the correct output, you discover that your program returns the wrong result. What do you do next? Describe your systematic strategy for debugging.

28. Erase all the comments in `invest.c`, recompile, and run the program to make sure it still functions correctly. You should be able to recognize what is a comment and what is not. Turn in your modified `invest.c` code.

29. The following problems refer to the program `invest.c`. For all problems, you should modify the original code (or the code without comments from the previous problem) and run it to make sure you get the expected behavior. For each problem, turn in the modified portion of the code only.

   a. **Using if, break and exit.** Include the header file `stdlib.h` so we have access to the `exit` function (Section A.4.14). Change the `while` loop in `main` to be an infinite loop by inserting an expression `<expr> in while(<expr>)` that always evaluates to 1 (TRUE). (What is the simplest expression that evaluates to 1?) Now the first command inside the `while` loop gets the user’s input. If the input is not valid, `exit` the program; otherwise continue. Next, change the `exit` command to a `break` command, and see the different behavior.

   b. **Accessing fields of a struct.** Alter `main` and `getUserInput` to set `inv.invarray[0]` in `getUserInput`, not `main`.

   c. **Using printf.** In `main`, before `sendOutput`, echo the user’s input to the screen. For example, the program could print out `You entered 100.00, 1.05, and 5.`

   d. **Altering a string.** After the `sprintf` command of `sendOutput`, try setting an element of `outstring` to 0 before the `printf` command. For example, try setting the third element of `outstring` to 0. What happens to the output when you run the program? Now try setting it to '0' instead and see the behavior.

For more information
visit www.nu32.org
e. **Relational operators.** In calculateGrowth, eliminate the use of $<=$ in favor of an equivalent expression that uses $!=$.

f. **Math.** In calculateGrowth, replace $i=i+1$ with an equivalent statement using $+=$. 

g. **Data types.** Change the fields $inv0$, $growth$, and $inarray[]$ to be float instead of double in the definition of the Investment data type. Make sure you make the correct changes everywhere else in the program.

h. **Pointers.** Change sendOutput so that the second argument is of type int *, i.e., a pointer to an integer, instead of an integer. Make sure you make the correct changes everywhere else in the program.

i. **Conditional statements.** Use an `else` statement in getUserInput to print `Input is valid` if the input is valid.

j. **Loops.** Change the `for` loop in sendOutput to an equivalent `while` loop.

k. **Logical operators.** Change the assignment of `valid` to an equivalent statement using `||` and `!`, and `no &&`.

30. Consider this array definition and initialization:

```c
int x[4] = {4, 3, 2, 1};
```

For each of the following, give the value or write “error/unknown” if the compiler will generate an error or the value is unknown. (a) $x[1]$ (b) $x$ (c) $(x+2)$ (d) $(x)+2$ (e) $x[3]$ (f) $x[4]$ (g) $(&&(x[1]) + 1)$

31. For the (strange) code below, what is the final value of $i$? Explain why.

```c
int i,k=6;
i = 3*(5>1) + (k=2) + (k==6);
```

32. As the code below is executed, give the value of $c$ in hex at the seven break points indicated, (a)-(g).

```c
unsigned char a=0x0D, b=0x03, c;
c = ~a;  // (a)
c = a & b;  // (b)
c = a | b;  // (c)
c = a << b;  // (d)
c = a >> 3;  // (e)
c = a << 3;  // (f)
c &= b;  // (g)
```

33. In your C installation, or by searching on the web, find a listing of the header file `stdio.h`. Find the function prototype for one function provided by the library, but not mentioned in this appendix, and describe what that function does.

34. Write a program to generate the ASCII table for values 33 to 127. The output should be two columns: the left side with the number and the right side with the corresponding character. Turn in your code and the output of the program.

35. We will write a simple bubble sort program to sort a string of text in ascending order according to the ASCII table values of the characters. A bubble sort works as follows.
Given an array of \( n \) elements with indexes 0 to \( n - 1 \), we start by comparing elements 0 and 1. If element 0 is greater than element 1, we swap them. If not, leave them where they are. Then we move on to elements 1 and 2 and do the same thing, etc., until finally we compare elements \( n - 2 \) and \( n - 1 \). After this, the largest value in the array has “bubbled” to the last position. We now go back and do the whole thing again, but this time only comparing elements 0 up to \( n - 2 \). The next time, elements 0 to \( n - 3 \), etc., until the last time through we only compare elements 0 and 1.

Although this simple program `bubble.c` could be written in one function (main), we are going to break it into some helper functions to get used to using them. The function `getString` will get the input from the user; the function `printResult` will print the sorted result; the function `greaterThan` will check if one element is greater than another; and the function `swap` will swap two elements in the array. With these choices, we start with an outline of the program that looks like this.

```c
#include <stdio.h>
#include <string.h>
#define MAXLENGTH 100 // max length of string input

void getString(char *str); // helper prototypes
void printResult(char *str);
int greaterThan(char ch1, char ch2);
void swap(char *str, int index1, int index2);

int main(void) {
  int len; // length of the entered string
  char str[MAXLENGTH]; // input should be no longer than MAXLENGTH
  // here, any other variables you need

  len = strlen(str); // get length of the string, from string.h
  // put nested loops here to put the string in sorted order
  printResult(str);
  return(0);
}

// helper functions go here
```

Here’s an example of the program running. Everything after the first colon is entered by the user. Blank spaces are written using an underscore character, since `scanf` assumes that the string ends at the first whitespace.

Enter the string you would like to sort: This_is_a_cool_program!
Here is the sorted string: !T____aacghiiimoooprss

Complete the following steps in order. Do not move to the next step until the current step is successful.

a. Write the helper function `getString` to ask the user for a string and place it in the array passed to `getString`. You can use `scanf` to read in the string. Write a simple call in `main` to verify that `getString` works as you expect before moving on.
b. Write the helper function `printResult` and verify that it works correctly.
c. Write the helper function `greaterThan` and verify that it works correctly.
d. Write the helper function `swap` and verify that it works correctly.
e. Now define the other variables you need in `main` and write the nested loops to
   perform the sort. Verify that the whole program works as it should.

Turn in your final documented code and an example of the output of the program.

36. A more useful sorting program would take a series of names (e.g., Doe_John) and scores
   associated with them (e.g., 98) and then list the names and scores in two columns in
descending order. Modify your bubble sort program to do this. The user enters a name
   string and a number at each prompt. The user indicates that there are no more names by
   entering 0 0.

   Your program should define a constant `MAXRECORDS` which contains the maximum number
   of records allowable. You should define an array, `MAXRECORDS` long, of `struct` variables,
   where each `struct` has two fields: the name string and the score. Write your program
   modularly so that there is at least a `sort` function and a `readInput` function of type `int`
   that returns the number of records entered.

   Turn in your code and example output.

37. Modify the previous program to read the data in from a file using `fscanf` and write the
   results out to another file using `fprintf`. Turn in your code and example output.

38. Consider the following lines of code:

   ```c
   int i, tmp, *ptr, arr[7] = {10, 20, 30, 40, 50, 60, 70};
   ptr = &arr[6];
   for(i = 0; i < 4; i++) {
       tmp = arr[i];
       arr[i] = *ptr;
       *ptr = tmp;
       ptr--;
   }
   ```

   a. How many elements does the array `arr` have?
   b. How would you access the middle element of `arr` and assign its value to the variable
      `tmp`? Do this two ways, once indexing into the array using `[]` and the other with the
derreferencing operator and some pointer arithmetic. Your answer should only be in
   terms of the variables `arr` and `tmp`.
   c. What are the contents of the array `arr` before and after the loop?

39. The following questions pertain to the code below. For your responses, you only need to
   write down the changes you would make using valid C code. You should verify that your
   modifications actually compile and run correctly. Do not submit a full C program for this
   question. Only write the changes you would make using legitimate C syntax.

   ```c
   #include <stdio.h>
   #define MAX 10
   void MyFcn(int max);
   ```
int main(void) {
    MyFcn(5);
    return(0);
}

void MyFcn(int max) {
    int i;
    double arr[MAX];
    if (max > MAX) {
        printf("The range requested is too large. Max is %d.\n", MAX);
        return;
    }
    for (i = 0; i < max; i++) {
        arr[i] = 0.5 * i;
        printf("The value of i is %d and %d/2 is %f.\n", i, i, arr[i]);
    }
}

a. while loops and for loops are essentially the same thing. How would you write an equivalent while loop that replicates the behavior of the for loop?

b. How would you modify the main function so that it reads in an integer value from the keyboard and then passes the result to MyFcn? (This replaces the statement MyFcn(5);.) If you need to use extra variables, make sure to define them before you use them in your snippet of code.

c. Change main so that if the input value from the keyboard is between −MAX and MAX, you call MyFcn with the absolute value of the input. If the input is outside this range, then you simply call MyFcn with the value MAX. How would you make these changes using conditional statements?

d. In C, you will often find yourself writing nested loops (a loop inside a loop) to accomplish a task. Modify the for loop to use nested loops to set the ith element in the array arr to half the sum of the first i − 1 integers, i.e., \( \text{arr}[i] = \frac{1}{2} \sum_{j=0}^{i-1} j \). (You can easily find a formula for this that does not require the inner loop, but you should use nested loops for this problem.) The same loops should print the value of each arr[i] to 2 decimal places using the %f formatting directive.

40. If there are \( n \) people in a room, what is the chance that two of them have the same birthday? If \( n = 1 \), the chance is zero, of course. If \( n > 366 \), the chance is 100%. Under the assumption that births are distributed uniformly over the days of the year, write a program that calculates the chances for values of \( n = 2 \) to 100. What is the lowest value \( n^* \) such that the chance is greater than 50%? (The surprising result is sometimes called the “birthday paradox.”) If the distribution of births on days of the year is not uniform, will \( n^* \) increase or decrease? Turn in your answer to the questions as well as your C code and the output.

41. In this problem you will write a C program that solves a “puzzler” that was presented on NPR’s CarTalk radio program. In a direct quote of their radio transcript, found here
http://www.cartalk.com/content/hall-lights?question, the problem is described as follows:

RAY: This puzzler is from my “ceiling light” series. Imagine, if you will, that you have a long, long corridor that stretches out as far as the eye can see. In that corridor, attached to the ceiling are lights that are operated with a pull cord.

There are gazillions of them, as far as the eye can see. Let us say there are 20,000 lights in a row.

They’re all off. Somebody comes along and pulls on each of the chains, turning on each one of the lights. Another person comes right behind, and pulls the chain on every second light.

TOM: Thereby turning off lights 2, 4, 6, 8 and so on.

RAY: Right. Now, a third person comes along and pulls the cord on every third light. That is, lights number 3, 6, 9, 12, 15, etc. Another person comes along and pulls the cord on lights number 4, 8, 12, 16 and so on. Of course, each person is turning on some lights and turning other lights off.

If there are 20,000 lights, at some point someone is going to come skipping along and pull every 20,000th chain.

When that happens, some lights will be on, and some will be off. Can you predict which lights will be on?

You will write a C program that asks the user the number of lights \( n \) and then prints out which of the lights are on, and the total number of lights on, after the last \( n \)th person goes by. Here’s an example of what the output might look like if the user enters 200:

How many lights are there? 200

You said 200 lights.

Here are the results:
  Light number 1 is on.
  Light number 4 is on.
  ...
  Light number 196 is on.
  There are 14 total lights on!

Your program lights.c should follow the template outlined below. Turn in your code and example output.

/****************************************************************************
* lights.c
* *
* This program solves the light puzzler. It uses one main function
* and two helper functions: one that calculates which lights are on,
* and one that prints the results.
* *
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>  // allows the use of the "exit()" function
```c
#define MAX_LIGHTS 1000000 // maximum number of lights allowed

// here's a prototype for the light toggling function
// here's a prototype for the results printing function

int main(void) {
    // Define any variables you need, including for the lights' states
    // Get the user's input.
    // If it is not valid, say so and use "exit()" (stdlib.h, Sec 1.2.16).
    // If it is valid, echo the entry to the user.
    // Call the function that toggles the lights.
    // Call the function that prints the results.

    return(0);
}
```

// definition of the light toggling function
// definition of the results printing function

42. We have been preprocessing, compiling, assembling, and linking programs with commands like

```bash
gcc HelloWorld.c -o HelloWorld
```

The `gcc` command recognizes the first argument, `HelloWorld.c`, is a C file based on its `.c` extension. It knows you want to create an output file called `HelloWorld` because of the `-o` option. And since you did not specify any other options, it knows you want that output to be an executable. So it performs all four of the steps to take the C file to an executable. We could have used options to stop after each step if we wanted to see the intermediate files produced. Below is a sequence of commands you could try, starting with your `HelloWorld.c` code. Do not type the “comments” to the right of the commands!

```bash
> gcc HelloWorld.c -E > HW.i // stop after preprocessing, dump into file HW.i
> gcc HW.i -S -o HW.s // compile HW.i to assembly file HW.s and stop
> gcc HW.s -c -o HW.o // assemble HW.s to object code HW.o and stop
> gcc HW.o -o HW // link with stdio printf code, make executable HW
```

At the end of this process you have `HW.i`, the C code after preprocessing (`.i` is a standard extension for C code that should not be preprocessed); `HW.s`, the assembly code corresponding to `HelloWorld.c`; `HW.o`, the unreadable object code; and finally the executable code `HW`. The executable is created from linking your `HW.o` object code with object code from the `stdio` (standard input and output) library, specifically object code for `printf`.

Try this and verify that you see all the intermediate files, and that the final executable works as expected. (An easier way to generate the intermediate files is to use `gcc` for more information visit www.nu32.org
HelloWorld.c -save-temps -o HelloWorld, where the -save-temps option saves the intermediate files.)

If our program used any math functions, the final linker command would be

```
> gcc HW.o -o HW -lm  // link with stdio and math libraries, make executable HW
```

The C standard library is linked automatically, but often the math library is not, requiring the extra -lm option.

The HW.i and HW.s files can be inspected with a text editor, but the object code HW.o and executable HW cannot. We can try the following commands to make viewable versions:

```
> xxd HW.o v1.txt  // can't read obj code; this makes viewable v1.txt
> xxd HW v2.txt  // can't read executable; make viewable v2.txt
```

The utility xxd just turns the first file’s string of 0’s and 1’s into a string of hex characters, represented as text-editor-readable ASCII characters 0..9, A..F. It also has an ASCII sidebar: when a byte (two consecutive hex characters) has a value corresponding to a printable ASCII character, that character is printed. You can even see your message “Hello world!” buried there!

Take a quick look at the HW.i, HW.s, and v1.txt and v2.txt files. No need to understand these intermediate files any further. If you do not have the xxd utility, you could create your own program hexdump.c instead:

```c
#include <stdio.h>
#define BYTES_PER_LINE 16

int main(void) {
    FILE *inputp, *outputp; // ptrs to in and out files
    int c, count = 0;
    char asc[BYTES_PER_LINE+1], infile[100];

    printf("What binary file do you want the hex rep of? ");
    scanf("%s",infile);  // get name of input file
    inputp = fopen(infile,"r");  // open file as "read"
    outputp = fopen("hexdump.txt","w");  // output file is "write"

    asc[BYTES_PER_LINE] = 0;  // last char is end-string
    while ((c=fgetc(inputp)) != EOF) {  // get byte; end of file?
        fprintf(outputp,"%x%x ",(c >> 4),(c & 0xf));  // print hex rep of byte
        if ((c>=32) && (c<=126)) asc[count] = c;  // put printable chars in asc
        else asc[count] = '.';  // otherwise put a dot
        count++;
        if (count==BYTES_PER_LINE) {  // if BYTES_PER_LINE reached
            fprintf(outputp," %s\n",asc);
            count = 0;
        }
    }
    if (count!=0) {  // print last (short) line
        for (c=0; c<BYTES_PER_LINE-count; c++) // print extra spaces
            fprintf(outputp." ");
        fprintf(outputp,"%s\n",asc);
    }
}
```

For more information visit www.nu32.org
Further Reading

Circuits Review

This appendix is meant as a brief refresher on basic analysis of circuits with resistors, capacitors, inductors, diodes, bipolar junction transistors, and operational amplifiers, at the level they are used in this book. In particular, this appendix does not cover the general frequency response of circuits with complex impedance. It also does not cover digital circuit design; in this book, most logical operations are performed by the PIC32.

B.1 Basics

Primary quantities of interest in circuit analysis and design are voltage and current.

Voltage is an effort variable, analogous to force in mechanical systems. In fact, voltage is sometimes referred to as electromotive force. Just as a force causes a mass to move, a voltage causes electrons (and therefore current) to flow. The unit of voltage is a Volt (V). Voltage is measured across elements (e.g., the voltage, or potential, at the positive terminal of a 9 V battery is 9 V greater than at the negative terminal). By defining the voltage at a particular point in a circuit as 0 V, or ground (GND), it is possible to refer to the voltage at a point, implicitly comparing it to ground.

Current is a flow variable, analogous to velocity in mechanical systems. The unit of current is an Ampere (A), commonly shortened to amp. Current is measured as a flow through circuit elements. Current into a circuit element must equal the current coming out of the element, and therefore current can only flow around a closed loop. It cannot, for example, flow into an element and stop there.

Just as force times velocity is power in mechanical systems, voltage times current is power in electrical systems. The unit of power is the Watt (W), and 1 W = 1 A \times 1 V. For example, Figure B.1 shows a generic circuit element (perhaps a battery, resistor, capacitor, diode, etc.). The voltage V across the element is defined to be positive if the potential is higher at the end of the element labeled +; otherwise V is negative. The current I through the element is defined to be positive if it is in the direction of the arrow, from + to −; otherwise I is negative.\(^1\) With

\(^1\) Note: The labeling of the ends of the element as + and − does not necessarily indicate which end has higher potential. It just indicates the convention chosen to call the voltage positive or negative. Similarly, the arrow does not necessarily indicate which direction the current actually flows.
Figure B.1
Defining positive voltage $V$ across, and current $I$ through, a circuit element.

Figure B.2
The same circuit drawn two different ways, with a battery of voltage $V$ and four generic elements, A through D.

these conventions, the power consumed by the element is $P = IV$. When $IV > 0$, the element is consuming electrical power, either dissipating it (as with a resistor) or storing it as energy (as with a capacitor or inductor). When $IV < 0$, the element is providing electrical power (e.g., a battery or a discharging capacitor). The unit of energy is the Joule (J), and $1 \text{ J} = 1 \text{ W} \times 1 \text{ s}$.

Figure B.2 shows a circuit with a battery of voltage $V$ and four generic elements, labeled A through D. The voltages across the elements are $V_A$ through $V_D$. The current $I_1$ flows through the battery, A, and B; $I_2$ flows through C; and $I_3$ flows through D. The same circuit is drawn in two different ways. In one, a battery is drawn explicitly, allowing a closed loop for current to be clearly visualized. In the other, which is more common in circuit schematics, the closed loop through the battery is left implicit. This circuit also introduces the ground symbol the voltage level defined as zero volts (at the negative terminal of the battery in this case).

To solve for voltages and currents in this circuit, we use Kirchhoff’s current law (KCL) and Kirchhoff’s voltage law (KVL). KCL says that current is preserved at any node: current into the node is equal to current out of the node. In Figure B.2, there are two nodes where currents...
come together, indicated by dots, and each provides the same equation:

\[ I_1 = I_2 + I_3. \]

KVL says that the sum of voltages around any closed loop must be zero. As you step around a loop, add voltages from elements where you proceed from the \(-\) terminal to the \(+\) terminal, and subtract voltages from elements where you proceed from the \(+\) terminal to the \(-\) terminal. For example, there are three loops in Figure B.2: through the battery, A, B, and C; through the battery, A, B, and D; and through C and D. These yield the following equations, respectively:

\[
\begin{align*}
V - V_A - V_B - V_C &= 0 \\
V - V_A - V_B - V_D &= 0 \\
V_C - V_D &= 0.
\end{align*}
\]

Only two of these equations are independent. For example, the third equation can be used to show that the first two are equivalent.

We now have three independent equations (one from KCL and two from KVL) to solve for seven unknowns in the circuit: the three currents \(I_1, I_2,\) and \(I_3,\) and the four voltages across the elements, \(V_A, V_B, V_C,\) and \(V_D.\) To get four more equations, we need the constitutive laws of the elements, relating the voltages across the elements to the currents through them. Let us begin with the constitutive laws of the common linear circuit elements: resistors, capacitors, and inductors.

### B.2 Linear Elements: Resistors, Capacitors, and Inductors

Resistors, capacitors, and inductors are called linear circuit elements because the voltages across the elements are proportional to the current, time integral of the current, or derivative of the current, respectively. The symbols, units, constitutive laws, and information about power and energy are summarized in Table B.1. Resistors only dissipate power, as heat, while

<table>
<thead>
<tr>
<th>Element</th>
<th>Schematic</th>
<th>Symbol</th>
<th>Unit</th>
<th>Constitutive Law</th>
<th>Power (W)</th>
<th>Energy Stored (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resistor</td>
<td>(R) (\Omega)</td>
<td>(R)</td>
<td>Ohm ((\Omega))</td>
<td>(V = IR)</td>
<td>(I^2R) dissipated</td>
<td>0</td>
</tr>
<tr>
<td>Capacitor</td>
<td>(C) (F)</td>
<td>(C)</td>
<td>Farad (F)</td>
<td>(I = C \frac{dV}{dt})</td>
<td>(CV \frac{dV}{dt})</td>
<td>(\frac{1}{2}CV^2)</td>
</tr>
<tr>
<td>Inductor</td>
<td>(L) (H)</td>
<td>(L)</td>
<td>Henry (H)</td>
<td>(V = L \frac{di}{dt})</td>
<td>(LI \frac{di}{dt})</td>
<td>(\frac{1}{2}LI^2)</td>
</tr>
</tbody>
</table>

For more information visit www.nu32.org
capacitors and inductors do not dissipate any power, but either charge (consuming electrical power and storing it as energy) or discharge (providing electrical power).

Figure B.3 shows the circuit of Figure B.2 with the generic elements replaced by resistors. We can solve for the four voltages across the resistors and the three currents by simultaneously solving the seven KCL, KVL, and constitutive law equations:

KCL: $I_1 = I_2 + I_3$
KVL: $0 = V - V_A - V_B - V_D$

Constitutive laws:

$V_A = I_1 R_A$
$V_B = I_1 R_B$
$V_C = I_2 R_C$
$V_D = I_3 R_D$

Substituting the battery voltage $V = 5 \text{ V}$ and the resistances $R_A = 10 \Omega$, $R_B = 20 \Omega$, $R_C = 30 \Omega$, and $R_D = 40 \Omega$, the currents and voltages can be solved as

$I_1 = 0.106 \text{ A}, \quad I_2 = 0.061 \text{ A}, \quad I_3 = 0.045 \text{ A}$
$V_A = 1.061 \text{ V}, \quad V_B = 2.121 \text{ V}, \quad V_C = 1.818 \text{ V}, \quad V_D = 1.818 \text{ V}$. 

**Figure B.3**
A resistor network.
According to our sign convention, where \( I \) is defined as positive if it flows from the terminal labeled + to the terminal labeled −, the power consumed by the battery is \(-I_1 V = -(0.106 \, \text{A})(5 \, \text{V}) = -0.53 \, \text{W}\). Since the power consumed is negative, the battery is providing power. The power consumed by \( R_A \) is \( I_1 V_A = I_1^2 R_A = 0.112 \, \text{W}\). The power consumed by \( R_B, R_C, \) and \( R_D \) can be calculated as 0.225, 0.112, and 0.081 W, respectively, and the sum of the power dissipated by the resistors is 0.53 W, equal to the power provided by the battery, as we would expect.

If any of the elements were capacitors or inductors, those constitutive laws would relate the current through a capacitor to the rate of change of the voltage, or the voltage across an inductor to the rate of change of current. Instead of simply solving linear equations as above, we must now solve linear differential equations. In this book we do not delve into analysis of linear circuits with arbitrary combinations of resistors, inductors, and capacitors, but focus on circuits with resistors only, as above, as well as circuits with resistors and either a single capacitor or a single inductor (Section B.2.1). Such circuits cover many practical cases of interest in mechatronics.

In Figure B.3, the resistors \( R_A \) and \( R_B \) are said to be in series. The resistors \( R_C \) and \( R_D \) are said to be in parallel. A simple derivation shows that resistors in series act like a single resistor of greater resistance, \( R_{\text{series}} = R_A + R_B \), and resistors in parallel act like a single resistor of lesser resistance (since there are now two paths for the current to follow),
\[
R_{\text{parallel}} = \frac{R_C R_D}{R_C + R_D}
\]
Similar relationships can be derived for capacitors and inductors (Table B.2).

The last linear element we will use is the potentiometer, or pot for short (Figure B.4). A pot is a resistor with three connections: the terminals at either end, like a regular resistor, and a third connection called the wiper. The wiper is an electrical contact that can slide from one end of the resistor to the other, creating a variable resistance between the wiper and the end connections. If \( R_+ \) is the resistance between the + terminal of the resistor and the wiper, and \( R_- \) is the resistance between the − terminal and the wiper, then the sum of \( R_+ \) and \( R_- \) always equals \( R \), where \( R \) is the total resistance of the pot between the two ends. Pots often come packaged in rotary knobs, and turning the knob moves the wiper to allow \( R_+ \) and \( R_- \) to be varied from approximately 0 to \( R \).

For more information visit www.nu32.org
B.2.1 Time Response of RC and RL Circuits

Figure B.5(a)–(d) shows four circuits, each consisting of a resistor $R$ and either a capacitor $C$ or an inductor $L$. For mechatronics, circuits with resistors and a single inductor are important for understanding the behavior of motors, which have significant inductance. Circuits with resistors and a single capacitor are often used for signal filtering.

The circuits in Figure B.5(a)–(d) are powered by a time-varying voltage $V_{\text{in}}(t)$ that periodically switches between $V_{\text{hi}} > 0$ and 0 V relative to ground. Let us focus on the circuit in Figure B.5(a), where $V_{\text{out}} = V_C$ is the voltage across the capacitor.

Figure B.5
(a–d) RC circuits and RL circuits. (e) Response of the circuits in (a) and (d) to a changing $V_{\text{in}}(t)$. Note the discontinuity in time in the middle of the plot, to allow the response to reach steady state. (f) Response of the circuits in (b) and (c) to a changing $V_{\text{in}}(t)$.
KVL tells us that the circuit in Figure B.5(a) satisfies

\[ V_{in}(t) = V_R(t) + V_C(t) = I(t)R + \frac{1}{C} \int_0^t I(s) \, ds. \]

Assume that the input voltage \( V_{in}(t) \) is equal to \( V_{hi} \), and has been for a long time so that any transients have died out. Current flowing through the capacitor has charged it up until, in steady state, the capacitor is fully charged to a voltage \( V_C = V_{hi} \) and an energy \( \frac{1}{2}CV_{hi}^2 \). Therefore there is no voltage across the resistor, so \( I = 0 \). The capacitor is acting like an open circuit.

The key point is that the voltage across the capacitor \( V_C(t) \) cannot be discontinuous in time if current is finite. For example, \( V_C \) cannot change from 0 to 5 V instantaneously; it takes time for the current to integrate to develop a voltage across the capacitor.

Now consider what happens when \( V_{in}(t) \) instantly changes from \( V_{hi} \) to 0 V. Since the voltage across the capacitor cannot change instantaneously, just after the switch occurs, the voltage across the capacitor is still \( V_{hi} \). This means that the voltage across the resistor \( R \) is \( -V_{hi} \). Therefore current must be flowing from ground through the capacitor and resistor. By KVL, and the constitutive law of the resistor, we can calculate the current \( I \) just after the switch at time \( t = 0 \) (let us call this time \( 0^+ \)):

\[ V_{in}(0^+) = 0 = V_R(0^+) + V_C(0^+) = I(0^+)R + V_{hi} \quad \rightarrow \quad I(0^+) = -\frac{V_{hi}}{R}. \]

This negative current begins to discharge the energy stored in the capacitor, and therefore the voltage across the capacitor begins to drop. To solve for \( dV_C/dt \), the rate of change of the voltage across the capacitor, at time \( 0^+ \), we use the constitutive law of the capacitor:

\[ I(0^+) = -\frac{V_{hi}}{R} = C \frac{dV_C}{dt}(0^+) \quad \rightarrow \quad \frac{dV_C}{dt}(0^+) = -\frac{V_{hi}}{RC}. \]

If the capacitor continued to discharge at this rate, it would fully discharge in \( RC \) seconds.

Of course the capacitor does not continue to discharge at this rate; the rate slows as the voltage across the capacitor drops. To fully solve for \( V_C(t) \) using KVL and the constitutive law of the capacitor, we solve the first-order linear differential equation

\[ 0 = I(t)R + V_C(t) \]

\[ 0 = C \frac{dV_C}{dt}(t)R + V_C(t) \]

\[ \frac{dV_C}{dt}(t) = -\frac{1}{RC}V_C(t) \quad \text{(B.1)} \]
Circuits Review

where the initial voltage $V_0$ is $V_{hi}$. This is an exponential decay to zero as $t \to \infty$. The time constant of the decay is $\tau = RC$, in seconds. One time constant after the switch, the voltage is $V_C(\tau) = V_0 e^{-1} = 0.37V_0$; the voltage has decayed by 63%. After $3\tau$, the voltage has decayed to 5% of its initial value.

Note that the time constant $\tau = RC$ is large if $R$ is large, since the large resistance limits the current to charge or discharge the capacitor, or if $C$ is large, because it takes more time for the current to charge or discharge energy in the capacitor, which is $\frac{1}{2} CV_C^2$.

If instead $V_{in}(t)$ has been at 0 V for a long time and then switches to $V_{hi}$ at $t = 0$, a similar derivation yields

$$V_{out} = V_C(t) = V_{hi}(1 - e^{-t/RC}),$$

a rise from 0 V asymptoting at $V_{hi}$. The voltage across the capacitor rises to 63% (95%) of $V_{hi}$ after time $\tau$ ($3\tau$).

Figure B.5(e) shows a plot of the fall and rise of the voltage $V_{out}(t) = V_C(t)$ in the circuit in Figure B.5(a) in response to a $V_{in}(t)$ occasionally switching between $V_{hi}$ and 0.

In Figure B.5(b), the positions of the capacitors and the resistors are reversed, so $V_{out} = V_R = V_{in}(t) - V_C(t)$. The response of $V_{out}(t)$ to the switching $V_{in}(t)$ is shown in Figure B.5(f). In this case, $V_{out}$ spikes to $-V_{hi}$ on a falling edge of $V_{in}(t)$, then decays back to zero, and spikes to $V_{hi}$ on a rising edge of $V_{in}(t)$, then decays back to zero.

In summary, the output of the circuit in Figure B.5(a) is a smoothed version of $V_{in}(t)$, where the output gets smoother as $RC$ gets larger, while the output in circuit in Figure B.5(b) responds most strongly at the times of the switches of $V_{in}(t)$. Smoothing is characteristic of a low-pass filter, while strong response to signal changes is characteristic of a high-pass filter; see Section B.2.2.

The circuits in Figure B.5(c) and (d) can be analyzed similarly, now using the constitutive law $V_L = L dI/dt$ for the inductor instead of $I = C dV_C/dt$ for the capacitor. It is also important to realize that the inductor does not allow current to change discontinuously, as a discontinuous current implies an unbounded voltage $L dI/dt$ across the inductor. It takes time to charge or discharge the energy in the inductor, $\frac{1}{2} LI^2$, and therefore $I$ cannot change instantaneously.

Based on this analysis, we see that the response of the RL circuit in Figure B.5(c) is that shown in Figure B.5(f), but now with a time constant $\tau = L/R$. Similarly, the response of the RL circuit in Figure B.5(d) is shown in Figure B.5(e), again with a time constant $\tau = L/R$. For more information visit www.nu32.org
Table B.3: Summary of capacitor and inductor behavior

<table>
<thead>
<tr>
<th>Element</th>
<th>Rule Enforced</th>
<th>Discharged</th>
<th>Charged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacitor</td>
<td>Continuous voltage</td>
<td>Wire</td>
<td>Open circuit</td>
</tr>
<tr>
<td>Inductor</td>
<td>Continuous current</td>
<td>Open circuit</td>
<td>Wire</td>
</tr>
</tbody>
</table>

Note that the time constant is large if $R$ is small, since the low resistance does not dissipate much power for a given current, or if $L$ is large, since it takes longer to charge or discharge the inductor’s energy $\frac{1}{2}LI^2$.

Some rules of analyzing circuits with a capacitor or inductor are summarized in Table B.3. When a capacitor is initially discharged, it lets current flow freely (like a wire), and when it is fully charged, it behaves like an open circuit (no current flows). When an inductor is initially discharged, it behaves like an open circuit (it takes time for current to build up as initially all voltage is claimed by $L \frac{dI}{dt}$), and when it is fully charged and $\frac{dI}{dt} = 0$, it lets current flow freely with no voltage across it (like a wire).

**Application: Switch debouncing**

Figure B.6 illustrates a closing mechanical switch, nominally generating a clean falling edge from GND to $V$. In practice, mechanical switches tend to bounce; the two metal contacts impact and bounce before coming into steady contact. The result is a $V_0(t)$ that rapidly switches between $V$ and GND before settling at GND. Switch bounce is a common problem, and programs responding to button presses should not respond to the bounces.

To remedy the signal bounce, a debouncing circuit, as shown in Figure B.6, can be used. First, an RC filter is used to slow down the voltage variations, creating the signal $V_1(t)$. Because the

![Figure B.6](image)

(Left) Bounces on the closing of a mechanical switch generate the output signal $V_0(t)$ on the right. (Middle) A debouncing circuit. The bouncing signal is RC filtered, creating the signal $V_1(t)$. This signal then passes through a Schmitt trigger inverter, creating a single clean rising edge $V_2(t)$. (Right) The signals $V_0(t)$, $V_1(t)$, and $V_2(t)$. The Schmitt trigger hysteresis voltages $V_h$ and $V_\ell$ are shown on the signal $V_1(t)$. 

For more information visit [www.nu32.org](http://www.nu32.org)
bouncing transitions occur quickly, the signal \( V_1(t) \) changes little during the bouncing. Once the bouncing has ended, \( V_1(t) \) drops steadily to zero, according to the RC time constant.

The signal \( V_1 \) is then fed to a digital output Schmitt trigger chip. The purpose of a Schmitt trigger is to implement *hysteresis*: if the input is currently low, the output does not change until the input has risen past a voltage \( V_h \); if the input is high, the output does not change until the input has dropped below a voltage \( V_\ell \); and \( V_h > V_\ell \). This hysteresis means that small variations of the input signal should not change the output signal. Further, a Schmitt trigger *inverter* makes the digital output opposite the input. The 74HC14 chip has six Schmitt trigger inverters on it.

Because the Schmitt trigger inverter ignores the small voltage variations at \( V_1(t) \) during bouncing, it does not change its output until the switch contact is steady. The end result of the debouncing circuit, \( V_2(t) \), is a single clean rising edge, after the bounces have terminated.

Since it is unlikely that you will need to press and release a button in less than 10 ms, it is not unreasonable to choose \( RC \approx 10 \) ms.

There are other debouncing circuits, and debouncing can instead be performed in software. See Exercise 16 of Chapter 6.

### B.2.2 Frequency Response of RC and RL Circuits

In the previous section we focused on the time response of RC and RL circuits in response to step changes in voltage. The step response is helpful to understand, as microcontrollers and some sensors output digital signals. We should remember, however, that by Fourier decomposition, any periodic signal of frequency \( f \) can be represented by a sum of sinusoids of frequency \( f \), \( 2f \), \( 3f \), etc. For example, the 50% duty cycle square wave of amplitude 1 and frequency \( f \) in Figure B.7 can be represented by an infinite sum of sinusoids at frequencies \( f \), \( 3f \), \( 5f \), etc. Therefore it is useful to understand the behavior of circuits in response to sinusoidal inputs.

![Figure B.7](image)

*The lowest four frequency components of a Fourier decomposition of a square wave of amplitude 1 and frequency \( f \).*
If the input to a linear system, like an RCL circuit, is a sinusoid of the form
\[ V_{\text{in}}(t) = A \sin(2\pi ft) \], the output is a scaled, phase-shifted sinusoid of the same frequency,
\[ V_{\text{out}}(t) = G(f)A \sin(2\pi ft + \phi(f)) \], where the system’s gain \( G(f) \) and phase \( \phi(f) \) are a function of the frequency \( f \) of the input (Figure B.8). Collectively the gain \( G(f) \) and the phase \( \phi(f) \) are called the frequency response of the system. For periodic non-sinusoidal input signals like the square wave in Figure B.7, the output is the sum of the individually scaled and shifted sinusoids that constitute the Fourier decomposition of the input.

Each of the RC and RL circuits in Figure B.5 is a linear system, with \( V_{\text{in}}(t) \) as input and \( V_{\text{out}}(t) \) as output. Without derivation (take a linear systems or circuits course!), the frequency responses of the circuits are plotted in Figure B.9, where Figure B.9(a) is the frequency response \( G(f) \) and \( \phi(f) \) of the circuits in Figure B.5(a) and (d), and Figure B.9(b) is the frequency response of the circuits in Figure B.5(b) and (c). Note that the frequency and gain

![Figure B.8](image_url)

**Figure B.8**
An example linear system time response to a sinusoidal input of amplitude \( A \). The output is phase shifted by \( \phi = -45^\circ \) and scaled by a gain \( G = 1/\sqrt{2} \).

![Figure B.9](image_url)

**Figure B.9**
(a) The frequency response of a first-order low-pass filter, e.g., the circuits in Figure B.5(a) and (d). (b) The frequency response of a first-order high-pass filter, e.g., the circuits in Figure B.5(b) and (c).
are plotted on log scales, to cover larger ranges of values and to more clearly show the essential features of the response.

The frequency response in Figure B.9(a) corresponds to a low-pass filter (LPF). This name comes from the fact that low-frequency sinusoids are passed from the input to the output with little change: $G \approx 1$ and $\phi \approx 0$. As the frequency increases, the gain drops and the output signal begins to lag the input signal ($\phi < 0$). At the cutoff frequency $f_c = 1/(2\pi \tau)$, where $\tau = RC$ for the RC circuits and $\tau = L/R$ for the RL circuits, the gain has dropped to $G(f_c) = 1/\sqrt{2}$ and the phase has dropped to $\phi(f_c) = -45^\circ$. Beyond the cutoff frequency the gain drops by a factor of 10 for every increase in the frequency by a factor of 10, so $G(10f_c) \approx 0.1$ and $G(100f_c) \approx 0.01$. The phase $\phi$ continues to drop, asymptoting at $-90^\circ$ at high frequencies.

The frequency response in Figure B.9(b) corresponds to a high-pass filter (HPF). This name comes from the fact that high-frequency sinusoids are passed from the input to the output with little change: $G \approx 1$ and $\phi \approx 0$. As the frequency decreases, the gain drops and the output signal begins to lead the input signal ($\phi > 0$). At the cutoff frequency $f_c = 1/(2\pi \tau)$, the gain has dropped to $G(f_c) = 1/\sqrt{2}$ and the phase has risen to $\phi(f_c) = 45^\circ$. Beyond the cutoff frequency the gain drops by a factor of 10 for every decrease in the frequency by a factor of 10, so $G(0.1f_c) \approx 0.1$ and $G(0.01f_c) \approx 0.01$. This means that DC (constant) signals are completely suppressed by the filter. The phase $\phi$ continues to rise with decreasing frequency, asymptoting at $90^\circ$ at low frequencies.

Low-pass and high-pass filters are useful for isolating a signal of interest from other signals summed with it. For example, LPFs can be used to smooth and suppress high-frequency noise on a sensor line. HPFs can be used to suppress DC signals and only look for sudden changes in a sensor reading, just as the output depicted in Figure B.5(f) is largest when the input suddenly changes value and drops to zero when the signal is constant.

The LPFs and HPFs illustrated in Figure B.9 are called first order because the circuit response is described by a first-order differential equation, e.g., (B.1). First-order filters have relatively slow rolloff—in the cutoff frequencies, the filter gain drops by only a factor of 10 for every factor of 10 in frequency. By using more passive elements, it is possible to design second-order filters with a gain rolloff of 100 for every factor of 10 in frequency, which are better at suppressing signals at frequencies we want to eliminate with less effect on signals at frequencies we want to preserve. Higher-order filters, with even steeper rolloff, can be constructed by putting first- and second-order filters in series. LPFs and HPFs can also be combined to create bandpass filters, which suppress frequency components below some $f_{\text{min}}$ and above some $f_{\text{max}}$, or bandstop or notch filters, which suppress frequency components between $f_{\text{min}}$ and $f_{\text{max}}$.

Filters constructed purely with resistors, capacitors, and inductors are called passive filters, as these circuit elements do not generate power. More sophisticated active filters can be created.

For more information visit www.nu32.org
using operational amplifiers (Section B.4). These circuits have the advantage of having high input impedance (drawing very little current from $V_{\text{in}}$) and low output impedance (capable of supplying a lot of current at $V_{\text{out}}$ while maintaining the desired behavior as a function of $V_{\text{in}}$).

The gain (or magnitude) portion of the frequency response is often written in terms of decibels as $M_{\text{dB}}$, where

$$M_{\text{dB}} = 20 \log_{10} G.$$ 

So a gain of 1 corresponds to 0 dB, a gain of 0.1 corresponds to $-20$ dB, and a gain of 100 corresponds to 40 dB.

### B.3 Nonlinear Elements: Diodes and Transistors

Nonlinear circuit elements are critical for computing and nearly all modern circuits. Two common types of nonlinear elements are diodes and transistors. While there are many kinds of transistors, in this section we focus on bipolar junction transistors (BJTs), notably excluding field effect transistors (FETs), which are extraordinarily useful and come in many different varieties. In keeping with the spirit of this appendix, we do not get into the semiconductor physics of these nonlinear elements, but focus on simplified models that facilitate analysis.

Analyzing circuits with simplified models of nonlinear elements is quite different from circuits with only linear elements. We do not simply write a set of equations and solve them. Instead, a nonlinear element can operate in different regimes (two regimes for a diode: conducting and not conducting; and three regimes for a BJT: off, linear, and saturated), each regime with its own governing equations. In principle, we have to solve a complete set of circuit equations for each possible combination of regimes for the nonlinear elements in the circuit. All but one of these guesses at the operating regimes will be wrong, leading to equations and inequalities without valid solutions.

#### B.3.1 Diodes

Figure B.10 shows the circuit symbol and the simplified current-voltage behavior of a diode. When the voltage across the diode is less than the forward bias voltage $V_d \geq 0$, no current flows through the diode. When current flows, it is only allowed to flow in the direction indicated in Figure B.10, from anode to cathode, and the voltage across the diode is $V_d$. It is never possible to have a voltage greater than $V_d$ across the diode. A typical forward bias voltage for a diode is around 0.7 V, but other values are also possible.

Figure B.11(a) shows a simple resistor-diode circuit. Assume $V_{\text{in}} = 5$ V and $V_d = 0.7$ V. To solve for $V_{\text{out}}$, we analyze the circuit for the two possible cases of the diode: conducting or not conducting.
(Left) The circuit symbol for a diode, indicating positive current and positive voltage. (Right) The simplified current-voltage relationship for a diode with forward bias voltage $V_d$.

![Diagram](image)

Figure B.10

(a) A resistor-diode circuit. (b) The output voltage $V_{out}$ for a sawtooth $V_{in}$. The output voltage is equal to the input voltage for $V_{in} < V_d$, but the output voltage is capped at $V_d$ by the diode. Reversing the direction of the diode would cause the output to track the input for $V_{in} > -V_d$, and the voltage would never drop below $-V_d$.

![Diagram](image)

Figure B.11

- **Case 1: diode is not conducting.** In this case, we know $I = 0$, so there is no voltage drop across the resistor, so $V_{out} = V_{in} = 5$ V. But we know that the diode can never have more than $V_d = 0.7$ V across it. Therefore this regime is not valid.
- **Case 2: diode is conducting.** Since the diode is conducting in this case, we know that the voltage across the diode is the forward bias voltage $V_d = 0.7$ V. Therefore the current $I$ must be $(5$ V $- 0.7$ V$)/R$ from the constitutive law of the resistor. This current is flowing in the right direction (positive current) and therefore does not violate the current-voltage relationship of the diode, so this is a valid solution.

Figure B.11(b) illustrates $V_{out}$ as $V_{in}$ follows a sawtooth profile, showing that $V_{out}$ can never exceed $V_d$. The power dissipated by the diode when current $I$ flows is $IV_d$.

A *light-emitting diode* (LED) is just a diode that emits visible or invisible light when current flows. A typical forward bias voltage for an LED is 1.7 V.

If a large negative voltage is placed across a diode, it may break down, allowing negative current to flow. While this is a failure mode for most diodes, for *Zener* diodes it is the intended use. Zener diodes are designed to have specific (and often relatively small) negative
breakdown voltages and to allow current to flow easily while at that breakdown voltage. Zener diodes can be used in voltage regulator applications.

**B.3.2 Bipolar Junction Transistors**

Bipolar junction transistors come in two flavors: NPN and PNP. We will focus on the NPN BJT, then return to the PNP.

*Figure B.12* shows the circuit symbol for an NPN BJT. It has three connections: the collector (C), base (B), and emitter (E). Current flows into the collector and base, and the sum of those currents flows out of the emitter, $I_E = I_C + I_B$. The voltage drop from the base to the emitter is denoted $V_{BE}$ and the voltage drop from the collector to the emitter is written $V_{CE}$. In normal usage, $I_C$, $I_B$, $I_E$, and $V_{CE}$ are all nonnegative.

The basic function of the NPN BJT is to attempt to generate a collector current that amplifies the base current, $I_C = \beta I_B$, where $\beta$ is the gain of the transistor (also commonly referred to as $h_{FE}$). A typical value of $\beta$ is 100. Depending on the amount of base current flowing, the transistor can be in one of three modes—off, linear, or saturated—and each mode provides three equations governing the transistor voltages and currents:

- **Linear**: $I_B > 0$ and $V_{CE} > V_{CE,sat}$. In this mode, the collector current is $I_C = \beta I_B$, and the transistor is not yet saturated, so if $I_B$ increases, $I_C$ will also increase. Saturation occurs when $V_{CE}$ drops to the collector-emitter saturation voltage $V_{CE,sat}$, which is commonly around 0.2 V or so. In the linear mode, $V_{BE}$ is equal to $V_{BE,on}$, the PN junction diode voltage drop from the base to the emitter. A typical value is $V_{BE,on} = 0.7$ V. **Governing equations**: $I_C = \beta I_B$, $V_{BE} = V_{BE,on}$, and $I_E = I_C + I_B$.

For more information visit www.nu32.org
• **Off**: $I_B = 0$. This means that $I_C = 0$ and therefore $I_E = 0$. $V_{BE}$ is less than $V_{BE,\text{on}}$.

  **Governing equations**: $I_B = I_C = I_E = 0$.

• **Saturated**: $I_B > 0$ and $V_{CE} = V_{CE,\text{sat}}$. In this mode the collector cannot provide more current even if the base current increases; the transistor is saturated. This is because the voltage between the collector and emitter cannot drop below $V_{CE,\text{sat}}$. This means that the relationship $I_C = \beta I_B$ no longer holds.

  **Governing equations**: $V_{CE} = V_{CE,\text{sat}}$, $V_{BE} = V_{BE,\text{on}}$, and $I_E = I_C + I_B$.

When $I_E > 0$, the power dissipated by the transistor is $I_B V_{BE,\text{on}} + I_C V_{CE,\text{sat}}$.

**Figure B.12** shows an NPN BJT in a *common emitter* circuit, so called because the emitter is attached to ground (“common”). We can determine the transistor’s operating mode as a function of $V_{in}$:

• **Off**: $V_{in} \leq V_{BE,\text{on}}$. Input voltages in this range do not provide enough voltage to turn on the base-emitter PN junction while also providing a base current $I_B > 0$. Since $I_C = 0$, there is no voltage drop across $R_C$, and $V_{out} = V$.

• **Saturated**: $V_{in} \geq V_{BE,\text{on}} + R_B (V - V_{CE,\text{sat}})/(\beta R_C)$. When the transistor is saturated, the output voltage is $V_{out} = V_{CE,\text{sat}}$, and the voltage across $R_C$ is $V - V_{CE,\text{sat}}$. This means $I_C = (V - V_{CE,\text{sat}})/R_C$. At the boundary between the linear and saturated regions, the relationship $I_C = \beta I_B$ is still satisfied, so $I_B = I_C/\beta$. So the minimum $V_{in}$ for saturation is the sum of $V_{BE,\text{on}}$ and the voltage drop $I_B R_B$ across the base resistor.

• **Linear**: All $V_{in}$ between the off and saturated regimes. In this regime, $I_B = (V_{in} - V_{BE,\text{on}})/R_B$ and $I_C = \beta I_B$, so

$$V_{out} = V - I_C R_C = V - \frac{\beta R_C}{R_B} (V_{in} - V_{BE,\text{on}}).$$

To increase the gain of a transistor we can use two transistors, Q1 and Q2, as a *Darlington pair* (**Figure B.13(a)**). The two collectors are connected and the emitter of Q1 feeds the base of Q2. The two together act like a single transistor, with the base of Q1 as the base of the pair.
and the emitter of Q2 as the emitter of the pair, but now the overall gain is $\beta_1 \beta_2$. $V_{BE, on}$ for the Darlington pair is the sum of the individual base-emitter voltages.

Finally, Figure B.13(b) shows the circuit symbol for a PNP transistor. For a PNP BJT, $I_C = \beta I_B$ as with the NPN, but now $I_B$ and $I_C$ flow out of the transistor and $I_E = I_C + I_B$ flows into it. At saturation, $V_E$ is greater than $V_C$ (typically by about 0.2 V), and when the transistor is on, the voltage drop from $V_E$ to $V_B$ is approximately 0.7 V. The PNP BJT is off if $I_B = 0$; it is saturated if $I_B > 0$ and the voltage drop from $V_E$ to $V_C$ indicates saturation; and otherwise it is in the linear mode.

### B.4 Operational Amplifiers

The circuit symbol for an operational amplifier (op amp) is shown in Figure B.14(a). Apart from the power supply inputs, the op amp has two inputs, a noninverting input labeled $+$ and an inverting input labeled $-$, and one output. Figure B.14(b) shows a particular chip, the 8-pin Texas Instruments TLV272, which has two op amps.

An ideal op amp obeys the following rules:

1. Input impedance is infinite. No current flows in or out of the inputs.
2. Output impedance is zero. The op amp can produce any current necessary to satisfy the following rule.
3(a). If there is no feedback connection between the output and the inputs, then the output satisfies $V_{out} = G(V_{in+} - V_{in-})$, where the gain $G$ is very large, effectively infinite. (The output goes to its maximum positive or negative value if $V_{in+}$ and $V_{in-}$ are different.)

3(b). If there is a current path from the output to the inverting input (negative feedback), for example through a capacitor or resistor, then the voltage at the two inputs are equal. This is because the large gain $G$ of the op amp attempts to eliminate the voltage difference $V_{in+} - V_{in-}$.

Almost all useful op amp circuits have negative feedback, so rule 3(b) applies.

![Figure B.14](image)

(a) The op amp circuit symbol. (b) The 8-pin TLV272 integrated circuit, with two op amps.
Common op amp circuits. Of these circuits, note that only the unity gain buffer and the noninverting amplifier present the very high input impedance of the op amp at the \( V_{\text{in}} \) input; in the other circuits, the input impedance is dominated by external resistors or capacitors. See Section B.5 for a discussion of input impedance.

Figure B.15 shows several useful circuits built with op amps. To analyze these circuits, remember that no current flows in or out of the op amp inputs (current only flows through the external resistors and capacitors), and since there is negative feedback in each of them, the voltages at the two inputs are equal.

For example, to analyze the weighted summer circuit, we recognize that both inputs are at 0 V (ground). Therefore the currents flowing through \( R \), \( 2R \), and \( 4R \) are simply \( V_2/R \), \( V_1/(2R) \), and \( V_0/(4R) \). Since no current flows in or out of the \( - \) input, these currents sum to give the current \( I \) through the feedback resistor \( R_f \), and the output voltage is simply \( V_{\text{out}} = -IR_f \). If the input voltages \( V_i \) are binary, this circuit provides an analog voltage representation of the three-digit binary number \( V_2V_1V_0 \), where \( V_2 \) represents the most significant bit (since it provides the most current) and \( V_0 \) represents the least significant bit. If instead the three resistors \( R, 2R, 4R \) are replaced by variable resistances set by potentiometers, the weighted summer is similar to an audio mixer.

The response of the integrator circuit is obtained by recognizing that the current flowing from \( V_{\text{in}} \) is \( I = -V_{\text{in}}/R \) and that

\[
V_{\text{out}} = -V_C = -\frac{1}{C} \int I(t) \, dt = -\frac{1}{RC} \int V_{\text{in}}(t) \, dt.
\]

If \( V_{\text{in}} \) is constant at zero, ideally \( V_{\text{out}} \) should also be zero. In practice, however, the voltage across the capacitor is likely to drift due to nonidealities of the op amp (Section B.4.1),

For more information visit www.nu32.org
including slight input offset. For this reason, it is good practice to put another resistor in parallel with the capacitor. This resistor serves to slowly bleed off capacitor charge, counteracting voltage drift. This resistance should be much larger than $R$ to prevent significant impact on the circuit’s behavior at frequencies of interest.

The voltage follower circuit simply implements $V_{\text{out}} = V_{\text{in}}$, but it is quite useful because it draws essentially no current from the source providing $V_{\text{in}}$ (such as a sensor) while being able to provide significant current at the output. For this reason the circuit is sometimes called a unity gain buffer: the op amp provides a buffer that prevents the circuit connected to the output of the op amp from affecting the behavior of the circuit connected to the input of the op amp. This allows individual circuits to be designed and tested modularly, then cascaded using buffers in between (Section B.5).

One application of the unity gain buffer is to implement an RC LPF or HPF (Section B.2.2). By cascading a unity gain buffer, then a passive RC LPF or HPF, we get the ideal frequency response of the passive filter but with high input impedance, as opposed to the relatively low input impedance of the passive filter alone. There are many more sophisticated higher-order op amp filter designs that achieve better attenuation at frequencies to be suppressed; consult any text on the design of op amp filters. In particular, for LPFs, HPFs, bandpass, and notch filters, popular filters are Butterworth, Chebyshev, and Bessel filters, each with somewhat different properties. These names refer to the form of the mathematical transfer function from input to output. To implement these transfer functions using op amps, resistors, and capacitors, there are different types of circuit designs; popular choices are the Sallen-Key circuit topology and the multiple feedback circuit topology.

### B.4.1 Practical Op Amp Considerations

If you want to purchase an op amp chip, you will find that there are tens of thousands to choose from! How do you choose? Op amp data sheets can be bewildering to read, with many different characteristics, most of them depending on the particular operating condition (the power supply voltage, the load at the output, etc.). Here are a few characteristics to consider, along with the values for the flexible and inexpensive TLV272.

- **Supply voltage range.** An op amp has both a minimum and a maximum allowable voltage across the power supply lines. TLV272: 2.7-16 V.
- **Output voltage swing (rail-to-rail or not).** The maximum outputs of some op amps do not reach all the way to the power supply rails, falling short by 1 V or more. Other op amps are *rail-to-rail*, meaning that the output voltage can come close to the power supply rails. The TLV272 is rail-to-rail, with a maximum output voltage swing to within about 0.1 V of the rails.
• **Input voltage range.** Even if the output goes rail-to-rail, the differential inputs may not be allowed to approach the rails. TLV272: inputs can go rail-to-rail.

• **Output current.** The maximum amount of current that can be provided by a single output. TLV272: up to 100 mA.

• **Unity gain bandwidth.** This is a specification of how fast the op amp output can change. For an input signal at this frequency, the effective gain of the amplifier (which we assumed to be infinite) has dropped to one. TLV272: 3 MHz.

• **Slew rate.** This is another measure of how fast the output can change, in V/s. TLV272: 2.4 V/µs.

• **Input bias current.** There is actually a very small current at the inputs (we assumed it to be zero). This is the typical average of those input currents. TLV272: 1 pA.

• **Input offset current.** This is the typical difference between the two input currents. TLV272: 1 pA.

• **Input offset voltage.** Ideally zero voltage difference at the inputs would cause zero voltage at the open-loop output. In practice, the input levels may have to be slightly different to achieve a zero voltage output. TLV272: 0.5 mV.

• **Common-mode rejection ratio.** The ideal amplifier amplifies only the difference between the voltages at the + and − inputs, but there is actually a small amplification of the common voltage between them. For example, if the voltage $V_+$ is 5.1 V and the voltage $V_-$ is 5.0 V, the usual amplifier gain acts on the 0.1 V difference while the common-mode gain acts on the average, 5.05 V. The CMRR specifies the ratio of the differential gain to the common-mode gain. TLV272: 80 dB (or 10,000).

• **Number of op amps.** Some chips have more than one op amp. TLV272: two op amps.

• **Packaging.** Op amps come in different types of packaging. DIP packages are easiest to work with for breadboard prototyping. TLV272: available in a variety of packages, including an 8-pin DIP.

• **Price.** Price increases for higher bandwidth and slew rates, higher output current, lower offset voltage, higher common-mode rejection ratio, rail-to-rail operation, etc. TLV272: about one dollar.

---

### B.4.2 Single Supply Design and Virtual Ground

When using an op amp in microcontroller applications, often the only power supply available is a positive voltage rail and ground (no negative rail), and the positive voltage may be small (e.g., 3.3 V). First of all, this likely means that a rail-to-rail op amp should be used, to maximize the output voltage range, and it should be capable of being powered by the microcontroller voltage (e.g., 3.3 V). Secondly, notice that many of the standard op amp circuits (Figure B.15) provide an output voltage that has a sign opposite of the input voltage, which the op amp cannot produce if there is no negative rail.

For more information visit www.nu32.org
One way to handle this issue is to introduce a virtual ground at a voltage halfway between the positive supply rail and ground, effectively creating a bipolar supply about this virtual ground. Figure B.16 illustrates the idea. A unity gain buffer is used in combination with a voltage divider to create a virtual ground at $V/2$. The capacitor helps stabilize the reference voltage to any transients on the power supply line. This virtual ground is then used in place of ground in the inverting circuits in Figure B.15. Inverted voltages are now with respect to $V/2$ instead of 0.

Since the op amp likely sinks or sources less current than a typical power supply, care should be taken to make sure that these limits are never exceeded.

### B.4.3 Instrumentation Amps

An instrumentation amp is a specialized amplifier designed to precisely amplify the difference in voltage between two inputs, $V_{out} = G(V_{in+} - V_{in-})$. Like an op amp, it has inputs $V_{in+}$ and $V_{in-}$, but unlike an op amp, it is not used with a negative feedback path. Instead, an instrumentation amp like the Texas Instruments INA128 allows you to connect a single external resistor $R_G$ to determine the gain $G$, where

$$G = 1 + \frac{50 \text{k}\Omega}{R_G}.$$  

The INA128 is typically used to implement gains $G$ from 1 ($R_G = \infty$, i.e., no connection at the gain resistor inputs) to 10,000 ($R_G \approx 5 \text{\Omega}$).

Other instrumentation amps allow you to choose from a fixed set of very precise gains, not dependent on an external resistor (with its associated tolerance). These are sometimes called programmable-gain instrumentation amps, and an example is the TI PGA204, which uses two digital inputs to choose gains of 1, 10, 100, or 1000. A related design is the TI INA110, offering gains of 1, 10, 100, 200, or 500.

Instrumentation amps distinguish themselves in their very high common-mode rejection ratio (120 dB for the INA128). Instrumentation amps are typically more expensive than op amps,
e.g., in the range of 10 to 20 USD for quantities of one. If you are trying to save money, you can build your own difference-and-gain circuit using multiple op amps, but you will not achieve the same performance as an instrumentation amp.

**B.5 Modular Circuit Design: Input and Output Impedance**

One way to design a complex circuit is to design subcircuits, each with a specific function, and then cascade them so that the output of one circuit is the input of another. Modular circuit design is similar to modular code design: each subcircuit has a specific function and well-defined inputs and outputs.

Modularity requires that connecting the output of circuit A to the input of circuit B does not change the behavior of either circuit. Modularity is assured if circuit A has low output impedance (it can source or sink a lot of current with little change in the output voltage) and circuit B has high input impedance (it draws little current at the input).\(^2\) For constant (DC) voltages and currents, if a change $\Delta I$ in the current drawn from the output of a circuit causes a change of voltage $\Delta V$, then the DC output impedance (or simply the output resistance, since the voltage is DC) is $|\Delta V/\Delta I|$. A circuit’s input resistance can be measured similarly. High input impedance means that a change in input voltage gives a very small change in input current.

Input and output impedance are generally frequency dependent. For sinusoidal signals of any frequency $\omega = 2\pi f$, the impedance of a resistor is simply its resistance $R$. The magnitude of the impedance of an inductor is $\omega L$, meaning that the impedance increases linearly with $\omega$—the impedance is zero at DC and infinite at infinite frequency. The magnitude of the impedance of a capacitor is $1/(\omega C)$, indicating that the impedance magnitude is infinite at DC and zero at infinite frequency.

As a simple DC example, consider the following design problem. We want to provide a user the ability to choose an input voltage between 0 and 3 V by turning a potentiometer knob. So we decide the circuit B will be a 10 kΩ potentiometer with one end at 3 V and the other end at 0 V, with the wiper providing the user’s input signal. No 3 V supply is available, however; there is only a 6 V supply. So we decide to design a circuit A, a voltage divider consisting of two resistors of resistance $R$, to create the 3 V reference. The output of circuit A becomes the input for circuit B (see Figure B.17).

Let us say we choose $R = 100$ kΩ for the voltage divider. Then the currents in Figure B.17 can be calculated using

\[^2\text{It is actually the ratio of input impedance to output impedance that matters. This ratio should ideally be multiple orders of magnitude.}\]
to find $I_1 = 55 \mu A$, $I_2 = 5 \mu A$, and $I_3 = 50 \mu A$. This means the voltage divider actually creates a voltage at the output of A of $6 \text{ V} - (55 \mu A)(100 \text{ k}\Omega) = 0.5 \text{ V}$ instead of 3 V. Circuit B “loads” or “pulls down” the output of circuit A. The output impedance $R$ of circuit A is too high relative to the 10 k\Omega input impedance of circuit B, defeating circuit modularity. Our attempt to design circuits A and B independently and put them together has failed.

On the other hand, if we choose $R = 100 \text{ k}\Omega$ for the voltage divider, we find that $I_1 = 30.15 \text{ mA}$, so $V_A = 2.985 \text{ V}$, very close to our target of 3 V. The output impedance of circuit A is much lower, so modularity is more closely achieved. This comes at the cost of greater power dissipated by the voltage divider, $V^2/R = (6 \text{ V})^2/200 \text{ k}\Omega = 180 \text{ mW}$ vs. $(6 \text{ V})^2/200 \text{ k}\Omega = 0.18 \text{ mW}$.

Op amps, with their high input impedance and low output impedance, are quite useful in achieving circuit modularity. In particular, a unity gain buffer between the output A and input B in Figure B.17 would eliminate any loading of the circuit A by circuit B, allowing us to use higher resistances for $R$ and therefore wasting less power.

Further Reading

Other PIC32 Models

As of this writing, there are nearly 200 different PIC32 models, arranged in six major families: PIC32MX1xx/2xx, PIC32MX3xx/4xx, PIC32MX5xx/6xx/7xx (from which our PIC32MX795F512H was chosen), PIC32MX1xx/2xx/5xx (64-100 pins), PIC32MX330/350/370/430/450/470, and PIC32MZ. The five PIC32MX families all use the MIPS32 M4K processor as the CPU, at speeds of 40-120 MHz, while the MZ family has a different architecture and uses the MIPS32 microAptiv microprocessor as the CPU at up to 200 MHz. “MIPS32” refers to CPU architectures and associated assembly language instructions licensed by Microchip from Imagination Technologies.

The main differences between the families are the CPU architecture (MIPS32 M4K vs. microAptiv), CPU clock speeds, amount of RAM and flash, physical packaging, available peripherals, number of pins, and the extent to which the pins can be mapped to different functions. This appendix provides a brief introduction to the features of the different families.

C.1 The PIC32MX5xx/6xx/7xx Family

Devices in the PIC32MX5xx/6xx/7xx family have names of the form

<table>
<thead>
<tr>
<th>PIC32MX</th>
<th>5, 6, or 7</th>
<th>x x</th>
<th>F</th>
<th>yyy</th>
<th>H or L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>communication options</td>
<td>other model options</td>
<td>flash in KB</td>
<td>64 or 100 pins</td>
<td></td>
</tr>
</tbody>
</table>

The 5xx series has full-speed USB and CAN peripherals, the 6xx series has full-speed USB and Ethernet, and the 7xx series has full-speed USB, Ethernet, and CAN. The xx code can be 34, 64, 75, or 95, corresponding to other model options, but primarily indicating the amount of RAM available (16, 32, 64, and 128 KB, respectively). The yyy code indicates the amount of flash memory, in KB. (All devices in this family also have an additional 12 KB of boot flash.) Devices ending in H have 64 pins and devices ending in L have 100 pins. Therefore the PIC32MX795F512H has full-speed USB, CAN, and Ethernet; 128 KB of RAM; 512 KB of flash; and 64 pins.

The M4K CPU can operate at up to 80 MHz for all devices in the PIC32MX5xx/6xx/7xx family. All devices have full-speed USB, five 16-bit counter/timers with up to two 32-bit
counter/timers, five input capture devices, five output compare modules with 16-bit resolution, 16 10-bit ADC inputs, six UARTs, two comparators, DMA, and PMP. Devices with 64 pins have three SPI and four I\textsuperscript{2}C peripherals; devices with 100 pins have four SPI and five I\textsuperscript{2}C peripherals. All devices are available only in surface mount packages.

### C.2 PIC32MX3xx/4xx Family

PIC32MX3xx/4xx devices are the first to have appeared in the PIC32 line. The M4K CPU can operate at up to 80 MHz for most devices in this family, but a few devices are limited to 40 MHz. Devices in this family have up to 32 KB of RAM and 512 KB of flash and have names of the form

<table>
<thead>
<tr>
<th>PIC32MX</th>
<th>3 or 4</th>
<th>_xx</th>
<th>F</th>
<th>yyy</th>
<th>H or L</th>
</tr>
</thead>
<tbody>
<tr>
<td>3: no USB; 4: with USB</td>
<td>20, 40, or 60</td>
<td>flash in KB</td>
<td>64 or 100 pins</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thus the PIC32MX460F512H has full-speed USB, 32 KB of RAM (with the “60” option), 512 KB of flash, and 64 pins.

Devices in the PIC32MX3xx/4xx family have similar capabilities to those in the PIC32MX5xx/6xx/7xx family, except they do not offer Ethernet or CAN, have fewer UART, SPI, and I\textsuperscript{2}C peripherals, and have only one 32-bit counter/timer.

### C.3 PIC32MX1xx/2xx Family

PIC32MX1xx/2xx devices are more recent than the 3xx/4xx and 5xx/6xx/7xx families. They are smaller devices, coming in 28-, 36-, and 44-pin devices. The 28-pin devices are available in DIP (dual inline package), convenient for breadboarding. The maximum CPU clock speed for a 1xx/2xx device is 40 or 50 MHz, depending on the model. The 1xx/2xx devices do not have a prefetch cache module; they run at full speed pulling instructions from flash.

Devices in this family have up to 64 KB of RAM and 256 KB of flash and have names of the form

<table>
<thead>
<tr>
<th>PIC32MX</th>
<th>1 or 2</th>
<th>_xx</th>
<th>F</th>
<th>yyy</th>
<th>B, C, or D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: no USB; 2: with USB</td>
<td>other model options</td>
<td>flash in KB</td>
<td>28, 36, or 44 pins</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The xx code can be 10, 20, 30, 50, or 70, which correspond to 4, 8, 16, 32, or 64 KB of RAM, respectively. Thus the PIC32MX230F064B has full-speed USB, 16 KB of RAM, 64 KB of flash, and 28 pins.

Devices in this family differ from the 5xx/6xx/7xx family in that they have only 3 KB of boot flash; fewer ADC inputs; fewer UART, SPI, and I\textsuperscript{2}C peripherals; and no Ethernet or CAN.

For more information visit www.nu32.org
The 1xx/2xx devices have three comparator modules instead of two, programmable with up to 32 reference voltages as compared to the 16 of the 5xx/6xx/7xx family. Other interesting new features, which we have not seen in the previous models, include Peripheral Pin Select (PPS), audio interface by SPI, and charge-time measurement for capacitive touch sensing, as discussed below.

PPS allows a wide range of digital-only peripherals to be assigned flexibly to different device pins, a major feature for low-pin-count devices like PIC32MX1xx/2xx devices. While a pin of our PIC32MX795F512H can support several possible peripherals, devices with PPS have much more flexibility in mapping certain peripherals to different pins. A remappable peripheral does not have default I/O pins; SFRs must be configured to assign the peripheral to specific pins before it can be used. Examples of peripherals that can be remapped by PPS include UARTs, SPI modules, counter/timer inputs, input capture, output compare, and interrupt-on-change inputs. Some peripherals cannot be remapped, such as I²C peripherals and ADC inputs, because of special requirements on the I/O circuitry for those peripherals.

The 1xx/2xx devices’ SPI modules support audio interface protocols for 16-, 24-, and 32-bit audio data. One example is the Inter-IC Sound (I²S) protocol, which allows the transmission of two channels of digital audio data using the SPI peripheral. The I²S capability allows a 1xx/2xx device to communicate with digital audio equipment as either the master or slave.

Finally, the 1xx/2xx’s Charge-Time Measurement Unit (CTMU) provides a current source to interface with an external capacitive touch sensor, such as a capacitive on/off button or even an x-y touchpad. The CTMU is used with one or more ADC channels to measure the capacitance of one or more analog capacitive sensors.

### C.4 PIC32MX1xx/2xx/5xx 64-100 Pin Family

The PIC32MX1xx/2xx/5xx 64-100 pin family expands on the features of the PIC32MX1xx/2xx family, which includes PPS, CTMU, and audio interface protocols. The M4K CPU operates at speeds up to 50 MHz, and the devices have 64 or 100 pins and up to 64 KB of RAM and 512 KB of flash. These devices feature more analog input channels (up to 48), more UART and SPI peripherals, and some models feature USB and CAN. Device names have the form

\[
\text{PIC32MX}_{\begin{array}{c}1, 2, \text{or } 5 \\ \text{communication options}
\end{array}}_{\begin{array}{c}xx \\ \text{other model options}
\end{array}}_{\begin{array}{c}F \\ \text{flash in KB}
\end{array}}_{\begin{array}{c}yyy \\ 64 \text{ or } 100 \text{ pins}
\end{array}}_{\begin{array}{c}H \text{ or } L
\end{array}}
\]

The 1xx series has neither CAN nor USB, the 2xx series features full-speed USB but no CAN, and the 5xx series has both full-speed USB and CAN. The code xx can be 20, 30, 50, or 70,
corresponding to 8, 16, 32, or 64 KB of RAM, respectively. Thus the PIC32MX570F512H offers both USB and CAN and has 64 KB of RAM, 512 KB of flash, and 64 pins.

C.5 PIC32MX330/350/370/430/450/470 Family

The PIC32MX330/350/370/430/450/470 family differs from the PIC32MX1xx/2xx/5xx 64-100 pin family by offering 12 KB of boot flash, up to 128 KB of RAM, and CPU speeds up to 120 MHz, the fastest in the PIC32MX families. Devices in this family have two SPI channels (compared to the three or four of the PIC32MX1xx/2xx/5xx 64-100 pin family) and do not offer CAN. 4xx devices in this family offer full-speed USB while 3xx devices do not.

C.6 PIC32MZ Family

The most recent addition to the PIC32 line, the PIC32MZ family is the most advanced. Devices in this family have names of the form

```
PIC32MZ  xxxx  three-letter code  yyy
          flash in KB       number of pins
```

The number of pins (yyy) is either 064, 100, 124, or 144. The three-letter code indicates whether the PIC32 has CAN modules; an external bus interface (EBI), described below; a floating point unit (FPU) for fast floating point operations; and a Crypto Engine, a hardware module used to accelerate applications requiring encryption, decryption, and authentication. PIC32MZs currently have up to 512 KB of RAM.

All PIC32MZs have a much larger boot flash segment (160 KB), nine 16-bit counter/timers configurable to up to four 32-bit counter/timers, nine output compares with up to 32-bit resolution, nine input captures, six UARTs, up to five I²C modules, up to six SPI modules supporting audio interfaces, PPS for more flexible pin remapping, Ethernet, and high-speed USB, as opposed to the slower full-speed USB of the PIC32MX models. PIC32MZs do not currently have a CTMU.

Two new capabilities on the PIC32MZ are the 50 MHz External Bus Interface (EBI) and the 50 MHz Serial Quad Interface (SQI). SQI is similar to SPI, except it has four data lines and supports single lane, dual lane, and quad lane modes of operation. In single lane mode, it is identical to SPI. EBI allows a high-speed connection to external memory devices, like NOR flash devices or SRAM, allowing you to seamlessly address external memory in your C code.

The PIC32MZ family also has several different peripheral buses, each potentially clocked at different frequencies.

For more information visit www.nu32.org
Certainly the biggest difference of the PIC32MZ family from PIC32MX devices is its different microprocessor, the MIPS32 microAptiv core. The microAptiv CPU can be clocked at up to 200 MHz, and it has multiple shadow register sets, a larger number of interrupt sources, and new assembly instructions and hardware to accelerate digital signal processing calculations. Other capabilities of the microAptiv core can be found in Section 50 of the Reference Manual.

C.7 Conclusion

To learn more about a specific PIC32 model, first consult the Data Sheet for the appropriate PIC32 family to learn the specific capabilities of each model. After that, you can consult the sections of the Reference Manual for more information. You will find it helpful to be armed with the knowledge of the features that your PIC32 model has or does not have, since the Reference Manual is currently written to cover all PIC32 models. After that, you can modify the sample code provided in this book for your particular PIC32, or start with sample code provided by Microchip.
Index

Note: Page numbers followed by f indicate figures and t indicate tables.

A

Analog input. See Analog-to-digital converter (ADC)

Analog output

ADC_max_rate.c program, 153–154

Absolute encoders, 326–327, 326f

ADC_max_rate.c program, 153–154

Analog-to-digital converter (ADC)

AD1CHS, 151

ADC_max_rate.c program, 153–154

AD1CON1, 149

AD1CON2, 150

AD1CON3, 151

AD1CSSL, 151

AD1PCFG, 149

block diagram, 145, 146f

clock period, 148

current sensor, 480

data format, 148

DC analog output, 137–139, 138f

time-varying, 140

DC motor control, 480

dual buffer mode, 149

input scan and alternating modes, 149

input voltages, 145

interrupts, 148

manual sampling and conversion, 152

multiplexers, 147

sampling and conversion

initiation events, 148

sampling and conversion timing, 146–147

unipolar differential mode, 148

voltage reference, 148

Angular velocity

absolute encoders, 326–327, 326f

gyro, 330, 331f

incremental encoder, 324–326, 325f

magnetic encoders, 327, 333f

potentiometer, 323–324, 324f

resolver, 327–328, 327f

tachometer, 328

B

Battery, 588, 21f

Bipolar junction transistors (BJT), 601–603, 601f, 602f

Bipolar stepper motor, 498–499, 498f

Bootloaded programs, 51–53, 62–63

Brushed permanent magnet direct current (DC) motor, 415, 416f

back-emf, 405–406

brushless motors, 403

electrical constant $k_e$, 403–405, 416

electrical time constant $T_e$, 413–414, 418

factors, 403–405

friction, 410–412, 411f, 419

loop of wire, 400, 401, 401f, 402

Lorentz force law, 399, 400f

max continuous current $I_{cont}$, 417

max continuous torque $\tau_{cont}$, 417

max efficiency $\eta_{max}$, 420

max mechanical power $P_{max}$, 420

mechanical time constant $T_m$, 414, 419

motor constant $k_m$, 412–413, 417, 422f

motor efficiency, 410–412, 411f

motor windings, 412–413, 422f

no-load current $I_0$, 419

no-load speed $\omega_0$, 419

nominal voltage $V_{nom}$, 419

non-negative torque, 402, 402f

Pittman brushed DC motor, 403, 404f

power rating $P$, 419

rotor inertia $J$, 418

short-circuit damping $B$, 415, 417

speed constant $k_s$, 403–405, 417

speed-torque curve, 406–410, 407f, 408f, 409f, 410f

stall current $I_{stall}$, 420

stall torque $\tau_{stall}$, 420

terminal inductance $L$, 417–418, 417f

terminal resistance $R$, 415

torque constant $k_t$, 403–405, 415–416

voltage, 405

Brushless DC (BLDC) motor

advantages, 500

vs. brushed DC motors, 500

disadvantage, 500

Hall effect sensors, 501

Hall sensor feedback commutation, 507

library, 504–505

open-loop driving, 510

PIC32 interface, 504, 504f
Brushless DC (BLDC) motor
(Continued)
Pittman ELCOM SL
4443S013, 501–502, 502f,
503f
quadcopters, 509
three-phase motor driver,
502–503, 504f
working principle, 500–501,
500f

C
CAN. See Controller area network
(CAN)
Change notification interrupts,
118–119
Circular buffer, 168–169, 169f, 172
Clock stretching, 193
Comparator
analog output, 231
block diagram, 227, 228f
CMSTAT, 229
CMxCON, 227
CVRCON, 229
inverting (–) input, 227
noninverting (+) input, 227
vs. voltage, 230
Context save and restore, 97,
103–104
Continuous operating region,
408–409
Controller area network (CAN), 22
addresses, 257
bit rate, 250–251
bit stuffing, 251
CAN frame, 250–251
CiCFG, 253
CiCON, 253
CiFIFOBA, 255
CiFIFOCONn, 255
CiFIFOINTn, 256
CiFIFOUAn, 256
CiFLTCONn, 254
CiRXFn, 255
CiRXMr, 255
CiTREC, 254
dominant state, 249, 250
FIFOs, 252
interframe spacing, 251
interrupt vectors, 256–257
light control, 261–262
loopback mode, 260
message buffer, 252
receive messages, 258–260, 259f
recessive state, 249–250
SOF bit, 251
transceiver, 249, 250f
transmitting messages, 257–258,
258f
two wires bus, 249, 250f
Core timer interrupt
central processing unit, 101
configuration, 101
disabling interrupts, 101,
107–111
enabling interrupts, 101,
107–111
IFSx, 101
ISR, 100
priority and subpriority, 101
Counters/timers, 128–129
asynchronous counting, 124
block diagram, 123, 124f
duration, 125, 130
fixed frequency ISR, 128
interrupt flag, 127, 127f
PBCLK pulses, 124
period match, 123
period register, 123, 127
synchronous counting, 124
T1CONbits, 126
Timerx, 127
TxCONbits, 125, 126
Type A vs. Type B, 125
C programming language
blocks of code, 540
comments, 539
conditional statements, 559–560
data sizes, 531–532
Data Types, 518–521
dynamic memory allocation, 557
function definition, 550–551
integer overflow, 532–533
logical operators, 558
loops, 560–562
MATLAB/Python, 517
memory, addresses, and pointers,
526
multi-dimensional arrays,
556–557
one-dimensional arrays,
554–556, 554f
pointers, 553–554
preprocessor commands, 542
printing to screen, 529–531
program structure, 541–542
Quick start, 515–517
relational operators, 557
standard binary math operators,
551–553
standard library (see Library)
static memory allocation, 557
strings, 556
type conversion, 533–535
typedefs, structs, and enums,
545–547
variables, 547
whitespace, 540
Current sensor, 587
ADC, 480
Hall effect, 339
resistor and amplifier, 337–339,
338f

D
Data terminal equipment (DTE),
159, 161
Data Types
binary and hexadecimal,
518–519
bits and bytes, 519–521
compilation, 528–535
float, double, and long double,
523–524
sign and magnitude
representation, 524–526
DC motor control
ADC, 480
block diagram, 450–451, 450f
client code, 473
tom control with PWM, 445–447,
446f, 447f
Copley controls, 453–455, 453f,
454f
current control, 452–453
current sensor wiring and
 calibration, 478–480, 479f
data collection, 488
decisions, 470–471
EMI, 449
encoder menu options, 475–476, 475f
coder testing, 474–475
gain storage, 487–488
hardware components, 459–460, 460f
H-bridge, 480–482, 481f. see also (H-bridge) ITTEST mode, 482–483
LCD display, 488
model-based control, 488
motion control, 451–452, 452f
nested control loops, 450–451, 450f
operating mode, 477–478
optoisolators, 449, 450
PI current controller, 482–483
position control, 484
PWM, 480–482
real-time data, 488–489
regeneration, 447–449, 448f
software (see Software)
terminal emulator, 471
trajectory tracking, 484–485
trapezoidal moves, 489, 489f
Device Configuration Register, 97
DFT. See Discrete Fourier transform (DFT)
Digital inputs and outputs (DIO)
AD1PCFGbits, 117
buffered and open drain, 115
change notification, 118–119
external pull-up resistor, 115, 116f
LATxbits, 117
ODCxbits, 117
PORTxbits, 117
simplePIC.c, 119
TRISAbits, 117
Digital signal processing (DSP)
aliasing, 342–344, 342f, 343f
dsp_fft_fir.c code, 368
dsp library, 369, 370, 370f
filtering signals (see Finite impulse response (FIR) filter)
fixed-point format, 364, 369–370
frequency domain representation (see Discrete Fourier transform (DFT))
header file, 364
low-pass-filtered signal, 364, 365f
MATLAB client code, 366
MIPS functions, 364
nudsp_fft_1024 function, 371
nudsp_fir_1024 function, 371
nudsp_qform_scale function, 370–371
Python users, 365
sampled signals, 342–344, 342f, 343f
weighted sums, calculation (see Infinite impulse response (IIR) filters)
DIO. See Digital inputs and outputs (DIO)
Diodes, 599–601, 600f
Direct Memory Access controller (DMAC), 22
Disassembly file
bit manipulation version, 74
delay(), 74, 76
jump/branch, 75
LATAINV = 0x20 command, 73, 74
stack, 76
timing.c, 73
Discrete Fourier transform (DFT) complex numbers, 344
FFT, 345–346, 346f, 347, 348f, 349–353, 350f
frequency components, 344–345
normalized frequency, 344
N-periodic signal, 344
DMAC. See Direct Memory Access controller (DMAC)
Drivers
header files, 274
static and dynamic modes, 274
timers (see Timer (TMR) driver)
UART (see Universal asynchronous receiver/transmitter (UART))
DSP. See Digital signal processing (DSP)
DTE. See Data terminal equipment (DTE)
Dual buffer mode, 149

E
Electric actuators
linear brushless motors, 512
RC servos, 494–495, 495f
solenoids, 491–493, 492f
speakers, 493–494
stepper motors, 495–499, 496f, 497f, 498f
voice coil actuators, 493–494, 494f
Electromagnetic interference (EMI), 449
Electromotive force, 587

F
Fast Fourier transform (FFT)
analog signal, 345, 346f
DC component, 362–363
design tools, 363
inverse FFT, 349–353, 350f
lowest nonzero frequency, 345–346
MATLAB, 347, 348f
sample square wave, 362
sinusoidal component, 363, 364f
zero padding, 345
Feedforward control, 381, 382f
FIFOs. See First-in first-out queues (FIFOs)
Finite impulse response (FIR) filter convolution, 352f, 353
examples, 358–366, 358f, 359f, 360f, 361f
filter coefficients, 350–351, 351f
frequency response, 353
high-order filters, 357
impulse response, 351, 353
MATLAB, 357, 358
order of filter, 351
running average calculation (see Moving average filter (MAF))
scaled and time-shifted impulses, 351–353
Index

Finite state machines (FSMs)
callback function, 274
non-blocking function, 273
states and transitions, 272–273, 273f
task, 273–274
UART, 276
FIR filter. See Finite impulse response (FIR) filter
First-in first-out queues (FIFOs) receiving data, 160–161, 168–169, 252
transmitting data, 160–161, 168–169, 252
Flash memory
buffer declaration, 241
flashbasic.c, 243
flash_op function, 242
output, 243
RTSP, 239
SFRs, 240–241
Force sensor, 334–336, 335f, 336f
Frequency response
RC circuits, 596–599, 596f, 597f
RL circuits, 596–599, 596f, 597f
FSMs. See Finite state machines (FSMs)

G
Gated accumulation mode, 125
Gearing, 427–428, 428f
backdrivability, 429
backlash, 428–429
ball and lead screws, 430, 431f
bevel gears, 430, 431f
efficiency, 428
factors, 434
harmonic drive, 430, 431f
inertia matching, 434–435
input and output limits, 429
planetary gearhead, 429–430, 431f
rack and pinion, 430, 431f
reflected inertia, 432–434, 433f
speed-torque curve, 431–432, 432f
spur gearhead, 429, 431f
worm gear, 430, 431f

H
Hall effect sensors, 332–333, 333f, 339, 501
Hardware
configuration bits, 28
flow control, 161
NU32 development (see NU32 development board)
peripherals (see PIC32 architecture)
physical memory map, 27–28
pin functions, 18, 18f, 19f
SFR, 20
specifications, 17
Harmony
abstraction, 267–268
app.h file, 291
application implementation, 292
code generation, 268
drivers (see Drivers)
file and directory structure, 289–290
FSM, 272–274
installation directory, 268–269
main function, 291
Makefile, 290–291
middleware, 269
PLIB functions, 270–271
portable code, 268
setup, 270
SFR layer, 269, 269f
system_definitions.h file, 293
system_init.c file, 293
system_interrupt.c file, 294
system services, 269
system_tasks.c file, 295
USB devices (see Universal serial bus (USB) devices)
H-bridge
bidirectional operation, 440f, 441
current amplification, 439–440, 440f
flyback diode, 440–441, 440f
linear push-pull amplifier, 440f, 442–450
microcontroller pin, 439, 440f
PHASE/ENABLE mode, 444
switches, 444
Texas Instruments DRV8835, 444, 445, 445f
unipolar power supply, 443, 443f
High-speed sampling, 154
Human interface device (HID)
API function, 299
callbacks, 312
client code, 297
compiler command line, 297
configuration_descriptors, 311
device class definition, 301
device_descriptor, 311
documentation, 303
event handlers, 312
hid.h, 302, 303, 305
initialization and opening function, 313
macros, 301–302
Makefile and NU32bootloaded.ld, 299–300
string descriptor, 305, 312
system_config.h, 303
system_definitions.h, 305
talkingHID.c file, 300
variables, 312
vendor and product identifiers, 299
Hybrid stepper motor, 497, 497f

I
I2C communication.
See Inter-integrated circuit (I2C) communication
IEC. See Interrupt enable control (IEC)
IFSx. See Interrupt flag status (IFSx)
Incremental encoder, 324–326, 325f
Inertial measurement unit (IMU), 330–332
Infinite impulse response (IIR) filters
Chebyshev and Butterworth filters, 362

For more information
visit www.nu32.org
generalization, 361–362
roundoff errors, 362
sample signal, 358, 362, 363
Input capture (IC)
  ICxBUF, 223
  ICxCON, 222
modules, 221, 222
PWM signal, 223
Inter-integrated circuit (I²C)
  communication
  accelerometer/magnetometer, 203–204
bus collision, 193
clock stretching, 193
data transaction, 192–193
I²CxADD, 195
I²CxBRG, 195
I²CxCON, 194
I²CxMSK, 195
I²CxRCV, 195
I²CxSTAT, 194
interrupt-based master, 200–201
main program, 199
master code, 196
multiple devices, 208
OLED screen, 205, 205
slave code, 197–198, 200
timing diagram, 191–192, 192
two wires, 191
typical bus connection, 191, 192
Inter-integrated circuit (I²C)
  modules, 24
Interrupt enable control (IEC), 93
Interrupt flag status (IFSx), 93
Interrupt priority control (IPCy), 93
Interrupt request (IRQ), 25, 93
Interrupt service routine (ISR), 25
  central processing unit, 97
circular buffer, 168–169, 169,
    172
configuration, 99
clock save and restore, 92
core timer interrupt, 99–100
counters/timers, 128
decimation, 168, 169
decoding, 168
exception memory, 92
external interrupt inputs, 98–99,
    102–103
IECx and IFSx, 93
interrupt vector, 92, 93
IPCy, 93
IRQs, 93
multi-vector mode (see Core
timer interrupt)
real-time control applications, 91
SFRs, 98–99, 103–104
single vector mode, 93
SRS, 97
TMR driver, 285–287
volatile qualifier, 106
Interrupt vector, 92, 93
IPCy. See Interrupt priority control
(IPCy)
IRQ. See Interrupt request (IRQ)
ISR. See Interrupt service routine
(ISR)
ITEST mode, 482–483

K
Kirchhoff’s current law (KCL), 588–589
Kirchhoff’s voltage law (KVL), 588–589

L
LCD library
circuit diagram, 63, 64
HD44780, 63
header file, 64
LCDwrite.c program, 65
PMP, 215–216
LED. See Light-emitting diode
(LEDD)
Library
  bootloaded programs, 62–63
  definition, 59
  input and output library, 562
  LCD library, 63–64,
    64
Makefile, 572
middleware, 66
multi-file projects, 570–572
NU32 library, 59, 61
peripheral libraries, 66–67
printf and scanf, 567
rad2volume library, 567–568
stdlib.h functions, 566
string manipulation, 565
talkingPIC.c program, 60–61
Light-emitting diode (LED)
  ADC, 395
  block diagram, 388, 388
features, 396
MATLAB interface, 391–394
OC1, 389–390
open-loop PWM waveform, 390–391
phototransistor, 387–388, 388
  PI controller, 395–396
printGainsToLCD() function, 395
  wiring and testing, 389, 389
Light sensors
  photocell, 319, 320
  photodiode, 319–321, 320
  photointerruptor, 322, 322
  phototransistor, 321, 322
  reflective object sensor, 323, 323
Linear circuit elements, 589–599,
  19, 31, 29
Linear variable differential
  transformer (LVDT), 328, 329
Lorentz force law, 332
Low-pass filtering
  FIR filters, 357
  PWM signals, 137–139, 140

M
MAF. See Moving average filter
(MAF)
Magnetic encoders, 327, 333
Magnetic field sensing, 332–333, 333
Map file
  data memory report, 81
  heap, 78, 83
  kseg1 boot memory, 80
  kseg1 data memory usage, 82
  kseg0 program memory usage, 79, 80, 82
  my_cat_string, 82, 83
  out.map file, 79
  program flash, 78
  .sdata, 82
  stack, 78–79

For more information
visit www.nu32.org
**Index**

Microelectromechanical systems (MEMS)
- accelerometer, 329–330, 330f
- gyroscopes/gyros, 330, 331f
- IMU, 330–332

Middleware. See Universal serial bus (USB) devices

Modularity, 608–609

Motor sizing. See Gearing

Moving average filter (MAF)
- causal vs. acausal filters, 356–357
- frequency response, 356
- sensor signal, 353, 354f
- smoothed and delayed version, 353–354
- test frequencies, 354, 355f, 356, 356f

Multi-vector mode
- central processing unit, 101
- configuration, 101
- disabling interrupts, 101, 107–111
- enabling interrupts, 101, 107–111
- IFSx, 101
- ISR, 100
- priority and subpriority, 101

Non-volatile memory (NVM), 239, 240–241

NU32 development board, 4, 4f
- LEDs and USER button, 30, 31f
- PICkit 3 programmer, 30–31, 31f
- pinout, 28, 29f
- voltage regulators, 28–30

NU32 library, 61

NVM. See Non-volatile memory (NVM)

Ohmic heating, 408–409

OLED screen. See Organic light emitting diode (OLED) screen

Operational amplifiers
- characteristics, 605–606

PID controller. See Proportional-integral-derivative (PID) controller

PMP. See Parallel master port (PMP)

Polling mode, 284

Position-sensitive detector (PSD), 334, 334f

Potentiometer, 591, 31f

Power-saving modes
- idle and sleep modes, 234
- SFRs and configuration bits, 234–235

Prefetch cache module, 26, 77–78

Proportional integral (PI) controller, 395–396

Proportional-integral-derivative (PID) controller
- block diagram, 375, 376f
control saturation, 378
derivative, 377
everal gain tuning, 380–381, 381f
error difference and sum, 378
force proportional, 376
integer math vs. floating point
math, 378
integrator anti-windup
protection, 379
model-based control, 381–383, 382f
pseudocode, 377
reference signal, 376
sensor noise, quantization, and
filtering, 379
step error response, 375
time integral, 377
timestep and delays, 378
variants, 379–380, 380f
PSD. See Position-sensitive
detector (PSD)
Pulse width modulation (PWM)
signals, 445–447, 446f, 447f
DC analog output, 137–139,
138f
input capture, 223
pulse train, 134, 136
software, 463
time-varying analog output, 140
waveform, 134f, 136, 137f
Python programming language, 173

R
Random access memory (RAM), 26
context save and restore, 92
FIFOs, 252, 257
heap, 78, 83
physical memory map, 27–28
.sdata, 82
SRAM, 183–184
stack, 78–79
virtual memory map, 35–37, 36f
Real-time clock and calendar
(RTCC), 24
Receiving data (RX)
DTE, 159
FIFOs, 160–161, 168–169, 252
interrupts, 165
Run-time self-programming
(RTSP), 239, 240–241
Serial peripheral interface (SPI), 23–24
interrupt vector, 181
LSM303D accelerometer/
magnetometer, 178f, 186–187, 187f
SDI, SDO, and SCK pins, 177,
178f
spi_loop.c, 181–182
SPlxBRG, 181
SPIxBUF, 180
SPIxCON, 179
SPIxSTAT, 180
SRAM, 183–184
SFRs. See Special function
registers (SFRs)
Shadow register set (SRS), 97,
103–104
simplePIC.c program, 37
ADC, 49
archiver, 38
arguments, 53, 54
assembler, 38, 54
compilation, 38, 39f, 54
DIO, 119
extern variable, 46
Harmony, 49
hex file, 8, 54
include files, 46
installation, 54
libraries, 45
linker, 38, 54
LSM303D accelerometer/
magnetometer, 186
NU32bootloaded.ld linker
script, 50–51
pins RA4, RA5, and
RD13, 43
POSITION, LENGTH, and
MASK constants, 49
preprocessor, 38, 54
RTCALRM, 47
SFRs, 40–42, 41f
statements, 48
structs, 47
TCS/TCKP, 48
__TRISAbits_t data type, 47
type __RTCALRMbits_t, 47
USER button, 7
XC32 compiler, 44–46

Q
Quadcopters, 509
Quickstart
<COMportA>, 12
bootloader utility, 6–7
hardware, 4
Harmony’s version, 10
loading, 9–10
simplePIC.c, 7, 8
software, 4–6
talkingPIC.c, 11, 12
terminal emulator, 3
types, 3
USB port, 10
XC32 installation, 8

Index
simplePIC.c program
(Continued)
xc.h, 44–46
XC32 installation, 8
Software
angle, 464
average error, 460, 461f
bootloaded vs. standalone programs, 51–53
command line utilities, 55
command prompt, 5
cubic trajectory, 466
current controller, 464, 464f
current gains, 463
current sensor, 461, 462
data types, 470
debugging, 467
encoder, 462, 463
features, 461, 462f
FTDI Virtual COM Port Driver, 5
get mode, 466
idle mode, 466
integer math, 470
low-frequency position control loop, 460–461
Makefile, 5
Microchip XC32 compiler, 5, 6
modularity, 467–469
MPLAB Harmony, 5, 6
native C compiler, 5
nested high-frequency current control loop, 460–461
PIC32 quickstart code, 5, 6
position gains, 463
PWM, 463
reference trajectory, 465, 465f
simplePIC.c (see simplePIC.c program)
terminal emulator, 5
text editor, 5
trajectory execution, 466
variables, 469
virtual memory map, 35–37, 36f
Solenoids
BLDC (see Brushless DC (BLDC) motor)
electrical characteristics, 492
mechanical characteristics, 492
pontiac coil F041A, 491, 492f
push type, 492–493
TIP120 Darlington NPN bipolar junction transistor, 491–492, 492f
Special function registers (SFRs), 20
AD1CHS, 151
AD1CON1, 149
AD1CON2, 150
AD1CON3, 151
AD1CSSL, 151
AD1PCFG, 149
CAN (see Controller area network (CAN))
CLR, SET, and INV, 43–51
comparator, 227–230, 229f
data sheet, 41–42, 41f
flash memory, 240–241
hardware, 20
I2CxADD, 195
I2CxBRG, 195
I2CxCON, 194
I2CxMSK, 195
I2CxRCV, 195
I2CxSTAT, 194
input capture, 221–223
interrupt IRQ, vector, and bit location, 93, 98–99
I/O pins, 41–42
OCxCON, 134
OCxR, 136
OCxRS, 136
PMADD, 215
PMAEN, 215
PMDIN, 215
PMMODE, 214
power-saving modes, 234–235
PRx, 127
SPIxBRG, 181
SPIxBUF, 180
SPIxCON, 179
SPIxSTAT, 180
SRAM, 183–184
TMRx, 127
TRISA, LATA, and PORTD, 40–41, 42
WDT, 234–235
SPI. See Serial peripheral interface (SPI)
SRAM. See Satic random-accessmemory (SRAM)
SRS. See Shadow register set (SRS)
Stepper motors
applications, 498
bipolar stepper motor, 498–499, 498f
detent torque, 497
half-stepping, 496
holding torque, 497
hybrid stepper motor, 497, 497f
microstepping, 497
permanent magnet, 495–496, 496f
unipolar stepper motor, 499
variable reluctance, 496
Strain gauges, 334–335, 335f
Successive Approximation Register (SAR), 146
System clock (SCK) pin, 177, 178f
System services. See Timer (TMR) driver

T
talkingPIC.c program, 60–61
Time and space
coretimer, 71–72
disassembling (see Disassembly file)
map file (see Map file)
math, 78–89
prefetch cache module, 77–78
stopwatch, 71
XC32 compiler optimization, 70
Timer clock prescaler (TCKPS), 48
Timer (TMR) driver
APP_STATE_INIT and APP_STATE_RUN states, 287
callback and initialization structure, 282–283, 284, 285, 287
CLK and DEVCON system services, 284
interrupts, 285–287
project directory, 281
status, 289

For more information
visit www.nu32.org
Index

system_config.h file, 282, 286
system_definitions.h file, 281

Time response
RC circuits, 592, 592f, 593–594, 595, 595f
RL circuits, 592, 592f, 593–594, 595, 595f
switch debouncing, 595–596, 595f

Transmitting data (TX)
DTE, 159
FIFOs, 160–161, 168–169, 252
interrupts, 165

U
Unipolar differential mode, 148
Unipolar stepper motor, 499
Universal asynchronous receiver/transmitter (UART), 24
APP_STATE_ECHO states, 275–276
APP_STATE_QUERY states, 275–276
baud, 159–160

DRV_<drvname>_Open function, 275
FIFO, 160–161
FSMs, 276, 279
FTDI chip, 159
hardware flow control, 161
initialization struct, 278, 279
interrupts, 165
ISR, 168–169
MATLAB, 172
non-blocking mode, 279
NU32 library, 166
parameters, 159–160
project directory, 275
Python, 173

ReadUart and WriteUart functions, 279
struct controls, 278
system_config.h, 280
system_definitions.h, 279

transmission, 160, 160f
uart_loop.c, 163–164

UxBRG, 163
UxMODE, 161
UxRXREG, 163
UxSTA, 162
UxTXREG, 163

XBee radios, 174, 175f

Universal serial bus (USB) devices, 22
endpoints and descriptors, 296

HID (see Human interface device (HID))
host, 295–296
powering, 296–297
types, 296
wires, 296

V
Virtual memory map, 35–37, 36f
Voltage, 587

W
Watchdog timer (WDT), 23
SFRs and configuration bits, 234–235
sleep mode, 235–236

Wireless communication, 174, 175f

X
XBee radios, 174, 175f

Z
Zero padding, 345

For more information visit www.nu32.org
EMBEDDED COMPUTING AND MECHATRONICS
WITH THE PIC32 MICROCONTROLLER

An introduction to Microchip’s PIC32 microcontroller and its use in embedded computing and mechatronics

For the first time in a single reference, this book provides the beginner with a coherent and logical introduction to the hardware and software of the PIC32, bringing together key materials from the PIC32 Reference Manual, Data Sheets, XC32 C Compiler User’s Guide, Assembler and Linker Guide, MIPS32 CPU manuals, and Harmony documentation. This book also teaches you to use the Microchip documentation, allowing better life-long learning of the PIC32. The philosophy is to get you started quickly, but to emphasize fundamentals and to eliminate “magic steps” that prevent a deep understanding of how the software you write connects to the hardware.

Applications focus on mechatronics: microprocessor-controlled electromechanical systems incorporating sensors and actuators. To support a learn-by-doing approach, you can follow the examples throughout the book using the sample code and your PIC32 development board. The exercises at the end of each chapter help you put your new skills to practice.

Features include:
- Extensive, freely downloadable sample code for the NU32 development board incorporating the PIC32MX795F512H microcontroller
- Free online instructional videos to support many of the chapters

Coverage includes:
- Getting up and running quickly with the PIC32
- An exploration of the hardware architecture of the PIC32 and differences between PIC32 families
- Fundamentals of embedded computing with the PIC32, including the build process, time- and memory-efficient programming, and interrupts
- A peripheral reference, with extensive sample code covering digital input and output, counter/timers, PWM, analog input, input capture, watchdog timer, and communication by the parallel master port, SPI, I2C, CAN, USB, and UART
- An introduction to the Microchip Harmony programming framework
- Essential topics in mechatronics, including interfacing sensors to the PIC32, digital signal processing, theory of operation and control of brushed DC motors, motor sizing and gearing, and other actuators such as stepper motors, RC servos, and brushless DC motors
- Appendices introducing the C programming language and background material on circuit analysis

This book is ideal for beginners, as a reference for professionals, or as a textbook for a college course on mechatronics.